REHASH: A Flexible, Developer Focused, Heuristic Adaptation Platform for Intermittently Powered Computing

ABU BAKAR, Northwestern University, USA
ALEXANDER G. ROSS, Northwestern University, USA
KASIM SINAN YILDIRIM, University of Trento, Italy
JOSIAH HESTER, Northwestern University, USA

Battery-free sensing devices harvest energy from their surrounding environment to perform sensing, computation, and communication. This enables previously impossible applications in the Internet-of-Things. A core challenge for these devices is maintaining usefulness despite erratic, random or irregular energy availability; which causes inconsistent execution, loss of service and power failures. Adapting execution (degrading or upgrading) seems promising as a way to stave off power failures, meet deadlines, or increase throughput. However, because of constrained resources and limited local information, it is a challenge to decide *when* would be the best time to adapt, and *how* exactly to adapt execution. In this paper, we systematically explore the fundamental mechanisms of energy-aware adaptation, and propose *heuristic adaptation* as a method for modulating the performance of tasks to enable higher sensor coverage, completion rates, or throughput, depending on the application. We build a task based adaptive runtime system for intermittently powered sensors embodying this concept. We complement this runtime with a user facing simulator that enables programmers to conceptualize the tradeoffs they make when choosing what tasks to adapt, and how, relative to real world energy harvesting environment traces. While we target battery-free, intermittently powered sensors, we see general application to all energy harvesting devices. We explore heuristic adaptation with varied energy harvesting modalities and diverse applications: machine learning, activity recognition, and greenhouse monitoring, and find that the adaptive version of our ML app performs up to 46% more classifications with only a 5% drop in accuracy; the activity recognition app captures 76% more classifications with only nominal down-sampling; and find that heuristic adaptation leads to higher throughput versus non-adaptive in all cases.

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

Additional Key Words and Phrases: Energy Harvesting, Batteryless Platform, Intermittent Computing, Adaptation

ACM Reference Format:
Abu Bakar, Alexander G. Ross, Kasım Sinan Yıldırım, and Josiah Hester. 2021. REHASH: A Flexible, Developer Focused, Heuristic Adaptation Platform for Intermittently Powered Computing. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 5, 3, Article 87 (September 2021), 42 pages. https://doi.org/10.1145/3478077

1 INTRODUCTION

Miniaturization of energy harvesters and computation has recently enabled battery-free mobile sensing devices which promise longer lifetimes over their battery-powered predecessors. Battery-free sensors harvest energy from

Authors' addresses: Abu Bakar, Northwestern University, Evanston, IL, USA, abubakar@u.northwestern.edu; Alexander G. Ross, Northwestern University, Evanston, IL, USA, alexanderross2021@u.northwestern.edu; Kasım Sinan Yıldırım, University of Trento, Trento, Italy, kasimsinan.yildirim@unitn.it; Josiah Hester, Northwestern University, Evanston, IL, USA, josiah@northwestern.edu.

2474-9567/2021/9-ART87 $15.00
https://doi.org/10.1145/3478077

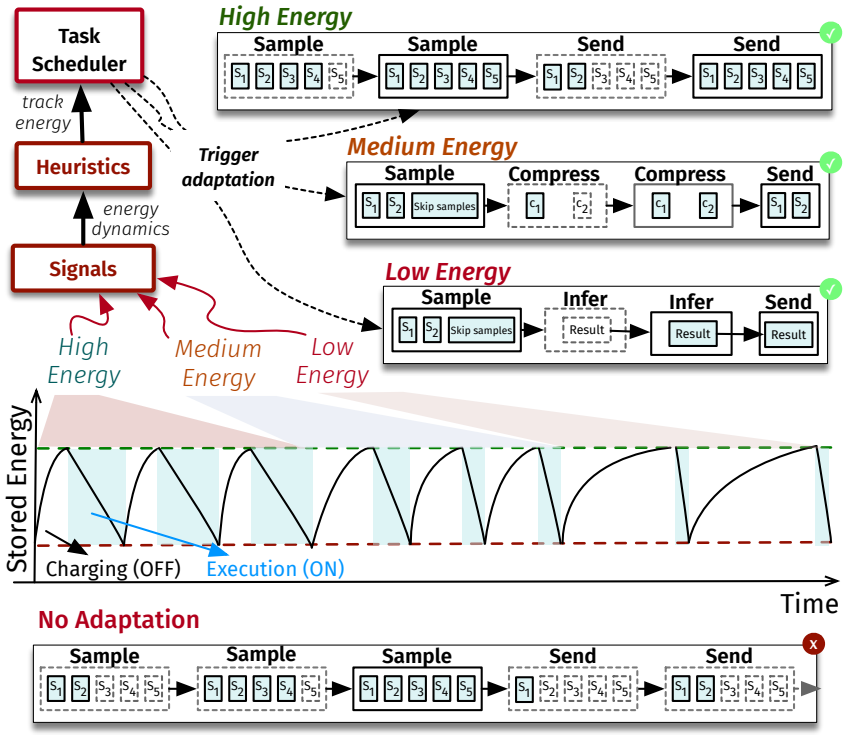


Fig. 1. A simple sense and send application samples the temperature sensor 5 times and sends it to a base station. Energy harvesting causes power scarcity and outages creating a *non-termination* state. Checkpoints or task divisions maintain forward progress but lead to a *greedy* execution that takes a much longer time. Adaptive execution strategies (top of the figure) quickly adapt to changing environmental conditions and finish off quickly—using harvested energy efficiently by gracefully modifying execution.

the ambient (solar, RF, and other sources), store it in tiny capacitors, and execute intermittently because of frequent power failures from energy scarcity. Despite their intermittent operation, these devices enable applications in the enormously hyped vision of *ubiquitous computing*, *smart dust* [44], and the *TerraSwarm* [49] because of their increased lifetime, reduced size and cost [33], and lower environmental impact [33, 54, 76]. However, frequent power failures where the stack frame is lost, CPU is reset, and peripherals are left in undefined states make development complex. Battery-free sensors (and energy harvesting sensor generally) cannot easily make use of a traditional embedded computing development stack, like TinyOS [51] Contiki [19], or mBed—making their large-scale deployment even more challenging.

Solving the intermittent computing problem has been challenging, with existing techniques split between two different approaches. In *checkpointing* approaches [5, 6, 8, 12, 36, 43, 46, 55, 57, 69, 69, 78], volatile content of the MCU, i.e., registers, stack, and the heap, is dumped into the non-volatile memory at specific points to ensure forward progress and memory consistency. In *task-based* approaches [12, 34, 56, 80], checkpointing is eliminated by asking programmers to decompose their program code into smaller idempotent units called *tasks*, which are atomic code blocks that can be executed completely once the capacitor is fully charged. Hardware platforms like Flicker [31], Capybara [14], and BOTOKS [17], and tools like Ekho [27] and EDB [15] have made development faster and more straightforward.

1.1 Core Challenge

The main difficulty in intermittent computing is maintaining performance and usefulness despite erratic, random, or irregular energy availability. These frequently failing systems are sensitive to the energy harvesting environment. Since tasks have rigid energy requirements, if the energy available in the environment is not enough to power a task (like calculating average motion), then the task will never complete. If the frequency of power failures and the length of power outages is higher than usual, these systems fail to produce useful outputs within a reasonable time (Fig. 1) and waste vital energy on checkpointing and non-useful computations. What is needed is a mechanism for *adaptation* so that tasks can be degraded or upgraded depending on the available energy to ensure that something useful happens. However, making this adaptation flexible, general, and most importantly understandable (in terms of understanding how the design decisions impact performance), is a challenge.

Works like Capybara, Flicker, and BOTOKS provide dynamic reconfigurability in *hardware* to change function in the face of energy harvesting. At the same time, runtime systems like InK [80] have notions of priority to provide some options when available energy is scarce. Recent work has tried to account for dynamism by automatically degrading performance to meet deadlines [58] or skipping task commit operations when energy is abundant to save time [59]. Finally, inference focused platforms like ePerceptive [65] and Zygarde [40] degrade machine learning tasks to meet deadlines or stay alive. These nascent efforts have not yet enabled generalized adaptive computation for intermittently powered devices or, perhaps more importantly, given the programmer flexibility and control on how and when to adapt. Without robust and programmer-specified adaptation, intermittent computing devices will be constrained in their potential deployment locations and reduced usefulness. Adaptation will increase the availability and responsiveness of battery-free devices, enabling numerous new applications. However, increased choices and flexibility lead to an increased tradeoff space. Programmers may become overwhelmed with the knobs and the variables, and possible configurations. For example, when choosing between degrading the sample rate or the communication rate when energy is low, the developer might be wondering what the impact of a certain decision will be on performance. The developer has little insight into how well the program will perform in the wild, or even if the program as written can get things done under the most likely energy harvesting environment.

1.2 Contributions: REHASH Runtime Framework and Toolchain

This paper describes a runtime system, framework, and complementary tool for developers to write robust intermittent applications that are highly **adaptive** to the energy available to the battery-free sensing devices and easily controllable/specifiable based on the application and energy environment constraints. Adaptation has emerged as a likely fundamental building block for intermittent computing and energy harvesting systems generally. However, the underlying mechanisms for deciding *when* to adapt, *how* to adapt, and how to *specify* and *control* adaptation for an arbitrary application and platform remain hidden or invisible in all prior work. Prior work has enabled rigid adaptation that is platform-dependent and out of the control of the programmer. All prior work employs only one adaptation policy based on an implicit *heuristic* (i.e., the voltage level on the capacitor) and knob (i.e., the checkpoint frequency). However, the choice of heuristics and knobs to adapt greatly depends on the application as well as the programmers' goals, not on the rigid policies employed by the underlying runtime. Finally, all adaptive systems to date hide the effect of adaptation on the application; basically, it is very difficult for a programmer to interpret the effect of a minor change in the heuristic, or runtime configuration, on the actual application performance. This work seeks to free the developer from these rigid constraints of other runtime systems and to elucidate how design decisions impact the application performance of these systems.

Our *key insights* are on the *when* and *how* of adaptation. We observe that easily gathered signals stemming from the intermittent operation of a sensor can provide a useful heuristic for estimating current and future energy availability. By taking these signals and applying simple transformations like taking into account history to understand the energy trend, these signals turn into fast, low overhead, and hardware-independent triggers

for *when* to adapt to degrade or improve execution. One example signal, the change in length of a power failure over time, is a direct marker of an environment's energy richness since energy-rich environments will have shorter off-times as they recharge the capacitor faster. When off-time increases, this would be a signal to degrade execution to maintain throughput since the environment is experiencing scarcity. When off-time decreases, an abundance of energy exists so that the execution quality can be upgraded.

The *adaptation strategy* refers to the actual mechanism used to reduce an application's energy consumption to provide higher throughput and availability. All previous adaptive runtimes define this implicitly. An example would be when an inference task on a low-resolution image must degrade performance to allow for continuous processing. One adaptation strategy could be to reduce the number of classes the task can recognize (for example, only recognize "cat", "dog", "null", and leave out "cow"). Another strategy would be to reduce the number of pixels looked at to make an inference. While the energy reduction is similar, the effect these different strategies have on the application quality is significant. Therefore the choice of the adaptation strategy employed must be decided by the developer, instead of the runtime. However, to make a a good decision, the programmer must be able to see what the impacts of these design choices are. We provide programming model and runtime support for developers to specify fine and coarse-grained adaptation of tasks from multiple heuristics, providing adaptation strategies executed by the runtime. Complementing the runtime, we provide a visual and online simulation tool that illuminates the cost or benefit of design decisions and configurations, enabling programmers to compare different configurations of an application quickly and without hardware. [1]

If adaptation is to become the modus operandi of making intermittent computing work for real-world applications, we must understand how it works, allow developers to configure and tune the mechanism of adaption for their application, and provide robust methods and tools for knowing when to adapt and understanding the performance impact of adaptation design decisions. We build the first heuristic adaptation runtime system and toolchain enabling these goals of flexibility, explainability, and generality. The core contributions of this paper are within the development of the REHASH framework and tool; we list the specific merits below:

(1) We outline *heuristic adaptation* as an approach to enable flexible, general, task-based intermittent computation that leverages easily gathered *signals* from changing environmental and physical phenomena, which are embedded in programmer defined *heuristic functions* that trigger short-term degradation or upgrading tasks.
(2) We identify and implement three lightweight, hardware-independent *signals* to assist this adaptation; *off-time*, *on-time*, and *task count*; and we design multiple *heuristic functions* that capture things like history, deadlines, and trends for general adaptation strategies.
(3) We develop a framework and runtime system (REHASH) for developers to specify how an application should adapt (or not) at the task level. For the first time, this enables highly flexible applications, finer-grained programmer control, and application-centered adaptation for intermittently powered systems.
(4) We develop a simulation tool, REHASH-`explorer`, with visual feedback that helps developers to conceptualize the impact of design decisions on energy-efficiency and application performance, building off a formal model of adaptation.

We release the source code of REHASH as a living artifact hosted publicly on Gitlab[2], to enhance replicability reproducibility for other researchers in this space, as well as to enable application developers who seek to use REHASH for their own intermittent computing systems. We built REHASH as a flexible, even plug-in-based framework, which we demonstrate in this paper. We anticipate that the tool and framework will be used and adapted for future runtime systems built by the community. These contributions make a task-based adaptive runtime system that i) is flexible and highly tunable to application requirements, ii) is hardware-independent

[1] A demo of the REHASH-`explorer` tool can be found at this link https://adaptationprofiler.github.io/adaptation-profiler
[2] We invite the reader to explore the source and documentation of REHASH at: https://gitlab.com/ka-moamoa/rehash

Table 1. A comparison of relevant intermittent runtimes with respect to their adaptable computing support. ✓and ✓indicate a good and a bad feature of the presented runtimes for the adaptation, respectively. Similarly, ✗indicates a good feature and ✗indicates a bad feature for adaptation which is missing in the runtime.

Intermittent Runtimes	Energy Prediction	Energy-aware Task Scheduling	Task Degradation	Task Starvation	Language Constructs	Configurable Adaptation	Multi-knob Tuning
Dewdrop [11], Mementos [69], Chain [12], DINO [55], Hibernus++ [5], QuickRecall [43], Ratchet [78], Clank [36], HarvOS [8], Alpaca [56], Mayfly [34], InK [80], Chinchilla [57], DICE [3]	**No Reaction to Energy Availability Dynamics**						
Dewdrop [11]	HW voltage meas.	✓	✗	✓	✗	✗ (scheduler driven)	✗no knobs
Coala [59]	task-count heuristic	✗	✗	✓	✗	✗ (runtime driven)	✗no knobs
Camaroptera [67]	HW voltage meas.	✗	code degradation	✗	✗	✗ (runtime driven)	✗no knobs
Celebi [41]	HW voltage meas.	✓	✗	✓	✗	✗ (scheduler driven)	✗no knobs
CatNap [58]	HW voltage meas.	✓	period, parameter and code degradation for events only	✓	✓	✗ (scheduler driven)	✗only one knob per event
ePerceptive [65], Zygarde [40]	HW voltage meas. Off-time heuristic	✓	only for inference	✓	✗	✗ (scheduler driven)	✗inference specific knobs
REHASH (this work)	Based-on many **SW-based** heuristics	✓	any degradation strategy (**many knobs**)	✗	✓	**programmer decides/configures at runtime** ✓(any adaptation strategy, any knobs)	✓multiple knobs (**simultaneously**)

since heuristic signals are easily gathered on commodity platforms, and iii) enables computation despite scarce energy. Importantly, we note that the presented heuristic adaptation methods generally apply to all energy harvesting systems (including energy-neutral, power neutral[72], and emergency reserve systems [42]). While the heuristics may change, the systematic adaptive framework presented is useful for all systems operating under energy scarcity and dynamism.

2 BACKGROUND: DEFINING ADAPTATION

Energy harvesting devices enable sustainable and affordable sensing across many applications from wearables to infrastructure monitoring in smart cities, to body implants. While energy is virtually limitless, actual availability at any point in time is effected by i) the efficiency of harvesters (source), ii) the sporadic availability and dynamic volume of ambient energy (environment), iii) and the increasing demand of computing for inference and learning (load) in the "TerraSwarm" as sensors become more intricate. This combination of factors leads to frequent energy scarcity, which leads to power outages and reduced throughput, sometimes necessitating intermittent execution through multiple power cycles, as many applications cannot be completely executed in one cycle.

2.1 Intermittent Computing Devices

Energy harvesting devices vary in approach; high powered devices like Signpost [1] which trickle charge a battery, power and energy neutral devices which trickle charge super capacitors [73], emergency reserve, ultra constrained devices like Permamote [42] which gather power for all operation from a solar panel but have

a battery as last resort, and finally intermittent computing devices which frequently fail and have minimal capacitance. All of these devices are at the whims of the energy environment, and benefit from adaptation.

The most constrained devices among them are the batteryless intermittently powered devices. Devices like the UMass Moo [81], and WISP [70] have been widely used for in-lab battery-free application development. They harvest ambient RF, buffer energy into small capacitors, and perform sensing. Other hardware platforms like Flicker [31], Capybara [14], and BOTOKS [17] have devised ways to intelligently manage harvested energy and keep track of time. A more recently developed platform, BFree [45] provides energy harvesting hardware that can be used with common hobbyist platforms, and lets novice developers write applications in Python. These platforms keep charging their capacitors without powering up the MCU, until an energy threshold is met to perform sensing and computing tasks. During execution, when the energy is fully drained from the capacitor, the device dies, resetting volatile state (register file, stack, program counter), preventing forward progress of computation and destroys memory consistency. This hinders the use of existing programming models designed for continuously-powered devices running correctly on batteryless sensors.

Software systems for intermittent computing preserve progress and ensure memory consistency by inserting checkpoints throughout program code—which store volatile content into a non-volatile storage (flash or FRAM [39]) when a power failure is approaching, and restore the execution from the same point when the power failure is over. Compiler based automatic checkpointing systems [3, 9, 18, 46, 57, 69, 78] either statically insert checkpoints at arbitrary points or place trigger calls that measure voltage level at different points in program code in order to determine checkpoint placement. On the other hand, dynamic systems [5, 6] keep monitoring supply voltage to make decisions on checkpointing. An alternative to checkpointing (which this paper approach extends) employs task-based programming abstraction models. In these systems [12, 56] developers are expected to manually define tasks that are guaranteed to either complete by committing their output to non-volatile memory, or to have no effect on program state. Finally, new programming language primitives as proposed in [34, 80] use remanence-based timekeeping techniques [17, 35] to meet data expiration deadlines and to keep track of time in failure, while others automatically degrade performance to meet task deadlines [58].

2.2 Need for Adaptation in Intermittent Computing

The design goal for all of the above approaches is to ensure forward progress while maintaining memory consistency. Some systems like COALA [59] or CatNap [58] provide automatic response to changes in energy; by skipping task commit operations or degrading performance to meet a deadline, respectively. But these system do not provide flexible methods for adapting arbitrary applications to changes in the energy availability. As an example of what adaptation could be, Fig. 2 shows the sampling activity and energy arrival for an Activity Recognition (AR) application—a machine-learning based human physical activity classification application that has been widely used for testing intermittent programming models [12, 25, 34, 55–57]. It collects several (a fixed number, typically 16) accelerometer samples into a sliding window, converts them into feature vectors, and classifies the window as moving or stationary. To power up the AR wearable, a capacitor may be used to store kinetic energy which is released when the person is moving. The stored energy can be utilized later when the person is at rest. This application is useful to demonstrates why the *when* and *how* of adaptation is important. When low energy causes execution to degrade, multiple strategies can be pursued; reducing sampling rates, degrading computation, reconfiguring the tasks, depending on the programmer needs.

Adaptation is not a new concept in mobile computer systems; in fact, it is a core factor of numerous classes of devices, in fact, concepts like imprecise computation and approximate computing in real time systems are highly related, and have been explored since at least the early 1990s [52]. Recent examples include throttling background apps [60] or adapting thermal load for android applications [68], to wearables devices that discard video frames or reduce resolution when battery life is low [66]. Adaptation in intermittent computing devices face different

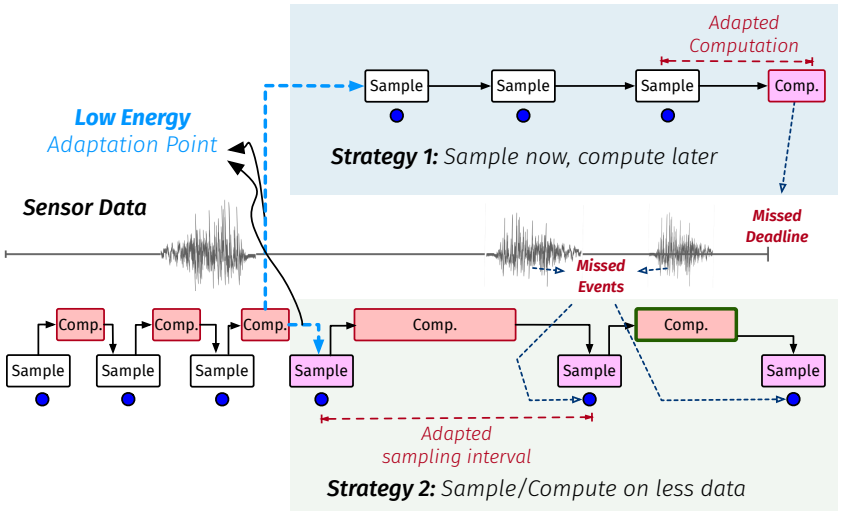


Fig. 2. For an activity recognition application, two different adaptation strategies are shown: one that prefers gathering samples consistently and waiting to compute over them, and another that gathers fewer samples and computes on them. In strategy 1; deadlines may be missed since no inference is done, but events are all captured, in strategy 2; sampling is reduced to meet the deadlines, but events may be missed. While both strategies are likely better than not adapting, the choice of *how* to adapt greatly depends on the application programmers goals.

challenges than those listed, as they suffer from frequent power failures, have severely constrained resources compared to any other adaptive system, and are usually limited in capability because of optimizing for cost.

2.3 Challenges of Intermittent Adaptation

Thus far, adaptation has been employed *implicitly* in various ways in multiple systems as referenced, to meet a particular goal (like meeting deadlines, or coalescing tasks, or scheduling tasks) considering the dynamic energy availability caused by energy harvesting. However, there is still no *explicit*, systematic, and flexible approach to recognize the dynamic energy availability, resize tasks accordingly, and in turn adjust the throughput by minimizing the end-to-end latency required for processing the application, subject to application developer constraints.

In particular, the challenges we introduce in this section are overlooked. A summary of the state-of-the-art from this perspective is presented in Table 1.

C1- Estimating the Energy Environment. The practicality of deciding *when* to adapt is important and ignored. It is hard to estimate the overall availability of energy in an environment, much less predict future energy, even for repetitive harvesting situations, since intermittent devices are operating on the margins of energy, small changes could mean big differences for static tasks. Understanding energy availability in the deployment environment, as well as its volatility, is, however a key tool for developers to provide the best quality of service under varying energy availability. While every environment displays different "modes" or "trends" in every energy harvesting [4] (i.e. diurnal shifts, activity based kinetic harvesters, RF harvesting changing based on distance), finding a general method to discern the energy environment is challenging because of the diversity of devices. Practically, implementing things like max power point tracking and advanced circuitry and computation to record and analyze trends in the energy harvesting environment requires significant energy, space, and time. These

methods also are not power failure resilient, meaning that holes in the data from power failures could lead to incorrect assumptions about the energy harvesting environment.

C2- Configurable/Flexible Adaptation. Responding to energy availability should be configurable in order to provide flexibility to the programmer. This is useful for introducing different strategies to mitigate task starvation and increase system availability. The when or what to adapt (or both) are generally not configurable, and are rigid in the specification of the application that will be executed by the existing runtimes [58, 59]. For useful and flexible adaptation, the programmer should have freedom to change (or not) any task or task variable, at once or in sequence. For instance, rather than degrading one variable, i.e. number of samples, the programmer can prefer an adaptation scheme where the number of samples can be *increased* but the parameter that relates with the transmission power of the radio can be decreased. Different applications require different adaptation strategies at different energy availability. As an another example, many sensing applications can be seen as being composed of a SENSE -> COMPUTE -> SEND loop. According to the energy level, different adaptation strategies can be more appropriate [67]. In high-energy mode, the application can perform SENSE DATA - SEND ALL DATA type execution—without any inference. During this mode, only the number of samples can be decreased up to a pre-defined value or/and the transmission power level of the radio can be adjusted. When the energy is moderate, the application can switch to SENSE + FILTER/COMPRESS + SEND DATA type execution. During this mode, fine grained adaptation can be supported via decreasing the number of samples, filtering out data, etc. In low-energy mode, the application can switch to SENSE - INFER - SEND INFERENCE type execution. Providing a framework for enabling this broad array of adaptation techniques is challening.

C3- Task Starvation. An intermittent program can be composed of several threads of computation with different energy requirements and priorities [80]. The assignment of the priorities to the thread of executions should reflect the criticality as well as timeliness of the computation—but it is not enough to prevent task starvation. In general, the rate at which the data is received and the rate at which it is being processed should match, in order to provide a reasonable quality of service with respect to each level of energy availability. When the stored energy is high; i.e. the capacitor is full, it can be the best time to execute a thread with high energy demand, such as sensor sampling. If the corresponding thread has lower priority than the others at that time, it will not execute and most likely it will starve. On the other hand, if the environmental energy is low, consuming the stored energy fast will prolong the recharge time and in turn the other low-priority threads that are waiting. Degrading the task and in turn thread sizes in a priority-based scheduling environment might prevent task starvation and create more room to execute lower priority tasks when the incoming environmental energy is low. However, by design, events have higher priority than tasks in many systems, e.g. in [58], to meet event deadlines. Events might consume the whole energy in the energy storage, e.g. audio sampling, that prevents the corresponding tasks from processing the event data, e.g. FFT computation—that leads to task starvation. Overcoming this task starvation challenge requires an intermittent program to adapt greedily with respect to the energy environment in order to provide a quality of service for each possible energy availability scenario, even with the risk of missing event deadlines.

C4- Too Many Design Choices. The flip side of the flexibility coin is that having numerous options gives decision fatigue, and more importantly, obscures what the difference in performance will be from different selections. All adaptive systems "bake in" the signals they use to adapt (i.e., off time for ePerceptive [65] or voltage measurements for most other systems). When these are hardcoded, the decision is easy, but the outcome may not be ideal. On the other hand, when there is more flexibility that allows the use of many signals and adaptation strategies, it might be hard to figure out the best combination for the best application performance (which can be user-defined).

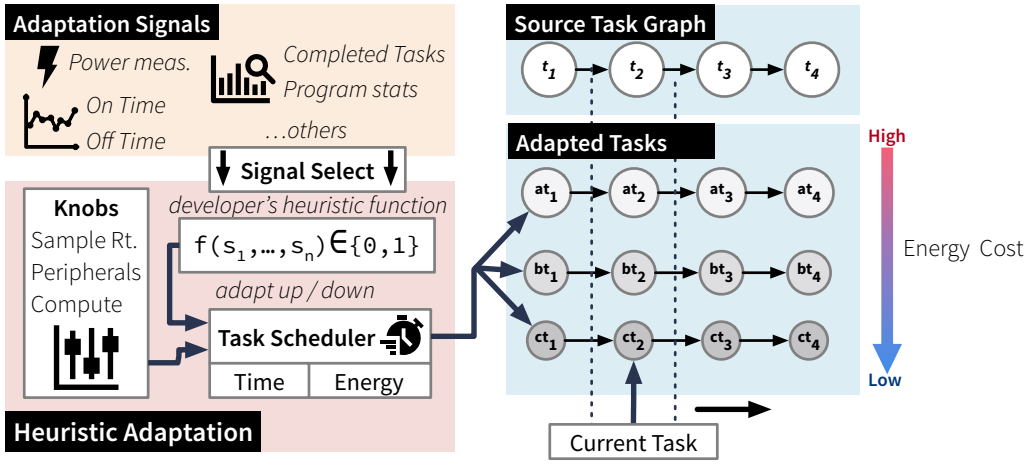


Fig. 3. Overview of our REHASH framework for heuristic adaptation. The hardware measures *signals* of the state of the energy harvesting environment such as on-time, off-time, task count, and recent events. A developer-defined *heuristic function* takes these signals in a logic equation that decides whether to adapt up (increasing energy use) or degrade performance (decrease energy use). The *heuristic function* embodies the developers application goals. The next adapted task replica is chosen, while the task scheduler obliviously executes the next (adapted) task.

3 REHASH: HEURISTIC ADAPTATION FRAMEWORK

We now provide a framework design for adaptation in task-based intermittent systems. The objective of our framework, REHASH, is to enable the development of intermittent applications that can respond to dynamic energy availability and provide the best quality of service. One of the main features of REHASH is employing **heuristic adaptation**, which uses *signals* derived from intermittent execution caused by energy arrivals, and dynamic energy availability in the environment—see Fig. 3. Based on the signals, REHASH uses a preset or developer-defined heuristic function to estimate the current energy availability of the environment with low overhead. The output of the heuristic function gives the adapt-up or adapt-down decision. Based on the adapt-up or adapt-down decision, REHASH task scheduler triggers the adaptation strategy that implements the necessary task upgrade/degrade operation accordingly, taking into account the various knobs that a developer has stated are allowed to be adapted. Then, the task scheduler executes the next (adapted) task—ensuring the forward progress of the computation meanwhile responding to the energy dynamics. Application developers have the domain knowledge to specify an adaptation strategy, which will guide how tasks change based on the up/down signal. Developers choose the heuristic signals that inform adaptation based on the hardware capability and application constraints.

Goals/Design Features. REHASH aims at enabling adaptation for intermittent systems in four key ways:

(1) **Portable energy approximation:** It is difficult to provide a general model to estimate energy consumption and harvesting, considering a variety of different hardware and energy harvesting circuitry. Heuristic adaptation provides a general and portable way to estimate energy availability and make adaptation decisions.

(2) **Many signals:** REHASH allows the selection and integration of many signals for adaptation. These signals can be inputs into a transform to account for history; for example by using functions like min/max/average/EWMA to get the best decision for the adaptation.

(3) **Many knobs:** REHASH allows the programmer to select among many "knobs" to change based on energy availability. These knobs include sampling rates, transmit power, neural network depth, and even configuration of peripherals.
(4) **Flexible and fine-grained adaptation constructs and strategy:** REHASH allows the programmer to control overall adaptation strategy, including the type and level of code degradation, the particular tasks to adapt or not, and the signals to trigger adaptation. When, where, and how to adapt (or not) is completely controlled by the programmer. REHASH is meant to be a super-set of the various adaptive themes explored in intermittent computing, giving the developer freedom to fine-tune adaptation to a specific application.

3.1 Signals: Tracking the Energy Dynamics

REHASH Framework— **signals:** *def.* measurable application based phenomena that exhibit proportional change in response to the energy harvesting environment changing.

Before a system can adapt, it has to have some reliable mechanism for understanding the environment. As shown in Table 1, some systems have used the amount of time a device is off between as a proxy for the energy richness of the environment [65]. Other systems use the number of tasks executed, or the measured voltage on the supply capacitor. These are signals of the environment that give the battery-free device some conception of the state of the outside world, and some ability to predict the future energy available.

REHASH relies on these adaptation *signals* (shown in Fig. 4). Each of which is a statistical measure or count of the *last* execution before a power failure. These signals change in response to the energy harvesting environment and the energy demand of the application. Many adaptation signals can be integrated into our REHASH framework. Obviously, the length of the time interval from the time the capacitor is fully charged and the device starts operating to the time the device fails, i.e. *on-time*, can be exploited as an adaptation signal. On-time captures the relation between the incoming energy rate and the energy demand of the application. If the incoming energy is high or the tasks of the application requires less energy to execute, one can expect longer on time. The inverse is true for *off-time*, which is a measure of how long does it take to charge an empty capacitor—if short, energy is high. A similar adaptation signal can be *task count* with more tasks completed meaning more energy available.

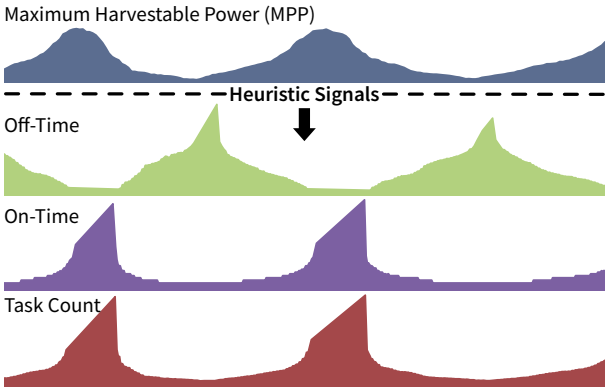


Fig. 4. This figure shows that *signals* in our framework track the maximum power harvestable from an energy environment, providing a useful estimate of energy availability trends. The signal values are drawn from an execution of the MNIST machine learning application on the energy environment.

Another approach for adaptation signals is *direct measurement* of the energy harvesting medium. However, this is a burdensome, and in many cases infeasible method for energy harvesting environment estimation for intermittent systems. Using indirect adaptation signals has multiple advantages compared to implementing a voltage or current measurement circuit on the device. Most importantly, a single point IV measurement (as demonstrated in Ekho [27] and shown in any datasheet for solar panels) does not represent the state of the energy harvesting environment, it is merely a point on an IV curve that is influenced by the solar panel characteristics and the charge on the capacitor. Conducting max power point tracking [21], where a dedicated circuit will dynamically change the point load on the harvester and then

measure the change, to extract the maximum power from a harvester, is a complex task (even modern ICs like Texas Instruments BQ25570 [37] only update MPPT every 16 seconds). Keeping track of statistic and trends of the MPP over time would be more indicative of the energy harvesting environment. However, that has a high burden in terms of power, circuitry, and compute. More confounding, even if the MPP was able to be tracked, *it can not be tracked while the device is off / in power failure.* As shown in Figure 4, the REHASH Framework signals track MPP, are maintained through power failures, and are much more lightweight, requiring low or no extra circuitry and little to no computation to derive.

3.2 Heuristic Function: When to Adapt

REHASH Framework— **heuristic function:** *def.* a developer-defined logic statement that takes an equation composed of measured *signals* and calculates a binary outcome, i.e. *adapt up*, or *adapt down*. The outcome of the *heuristic function* decides when to adapt tasks and knobs.

As depicted in Fig. 3, REHASH heuristic is a function, denoted by $f(s_1, \ldots, s_n)$, with a binary output: either "adapt-up" (0) or "adapt-down" (1). The inputs of the heuristic function are the adaptation signals, denoted by s in this figure. REHASH can accept any heuristic function, that can use many or only one adaptation signal in order to output the adaptation decision, e.g. only on-time or on-time and the number of task completions. REHASH also integrates a measure of past history—that holds the values of the adaptation signal in the previous active time computation, before the most recent power failure. Integrating more history means taking into account the signal values of previous active executions. The heuristic function can be any equation or relation expressed in a mathematical form (with the constraint that excessively complicated heuristics will take longer to compute, and may become less interpretable).

As an example, a heuristic function can be created that compare the latest measured value of a single adaptation signal, e.g. the *task completion count* with the average value of the corresponding adaptation signal stored in the history. If the current value of the signal is less than the average, the application should adapt down (for example decrease the sampling count), since the trend is that less active time and fewer tasks are completed, which likely means the energy environment is trending low. Otherwise, the application should adapt up, since the time between power failures is increasing, so more energy is available to do more tasks and stay on for longer.

3.3 Knobs: What to Adapt

REHASH Framework— **knobs** or **adaptive variables:** *def.* application specific variables controlling sensor sampling, peripheral selection, communication, and compute, that can be tuned to use more/less energy based on the outcome of the *heuristic function.*

REHASH exposes multiple knobs to the heuristic adaptation functions, so that these knobs can be tuned in order to provide the best quality of service. Battery-free energy-harvesting devices support a wide variety of applications, ranging from measuring body vitals: heartbeat, activity recognition, gait monitoring etc.; to managing built environments: sensing temperature, humidity, proximity, lighting conditions, etc.; and many more. These devices employ multiple sensors to measure desired facet(s); use different filtering algorithms to process sensor data; encode/encrypt this data for user's privacy; send it to a server for further processing or actuation, and repeat the whole cycle again. A typical control flow graph of battery-free sensing applications is shown in Fig. 5.

For the sake of simplicity, we broadly classify all peripheral and sensing related operations such as sensor selection, sensing parameters (e.g. data rate), radio configuration etc., into the hardware layer; and all the compute:

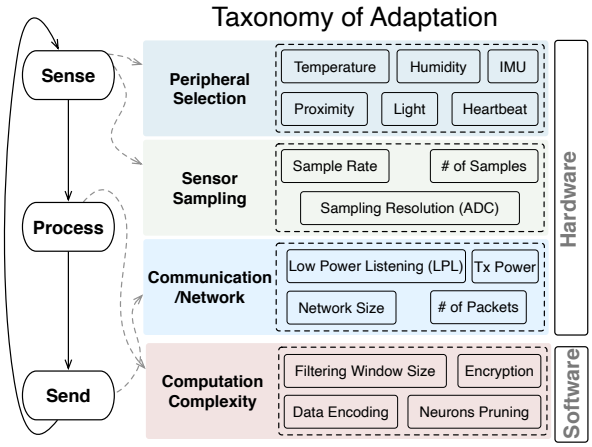


Fig. 5. Typical control flow graph of sensing is shown on the left; at each stage, energy-aware adaptation can be viewed from different *dimensions*: peripheral selection, sensor sampling, etc. For each dimension, there are different *knobs* for adaptation. For instance, while sensing, sampling rate may be adapted based on available energy.

Table 2. Survey of intermittent computing applications with *knobs* for adaptation listed.

Applications	Sensors	Adapt. Knobs
Activity Recognition [12, 55]	Accelerometer	SS, CC, CV
Green House Monitoring [30]	Temp, Humid, L.Wetness	PS, SS, CV
Condition Monitoring [80]	Micro., Accelerometer	SS, CC
Correlated Sensing & Report [14]	Magneto., Proximity	SS
Cuckoo Filter [56, 57]	–	CC
Air Quality Learning [50]	Temp, Light	SS, CC
Visual Sensing [40]	Camera	CC
Bitcount [56, 57]	–	CC
Acoustic Sensing [40]	Microphone	SS, CC
Blowfish Encryption [56, 57]	–	CC
Vibration Learning [50]	Accelerometer	SS, CC
RSA Encryption [43, 69]	–	CC
Visual Sensing-Camaroptera [67]	Camera	CC, CV
Kalman Filter [9]	–	CC
CRC [69, 78]	–	CC

filtration, encryption, encoding, machine learning algorithms, etc., into the software layer. These are different *dimensions* that help visualizing adaptation in a hierarchical way. With these dimensions defined, REHASH can be used to employ several adaptation strategies that can tune different knobs at each stage of the execution.

To ground our taxonomy in the real world, we conducted a literature review and outlined all the intermittent computing applications and their possible adaptation strategies in Table 2.

Peripheral Selection (PS). Typically, sensing is done periodically or by polling. When a multi-sensor device is in high energy mode, all the sensors can be sampled regularly. However, if energy availability is lower, routine sampling of all sensors would take much longer, causing delay in processing and actuation, and thus jeopardizing the whole sensing cycle. We propose adaptation of sensing facets in such a scenario by turning some sensors off, in order to give the best possible outcome with limited energy. For instance, an infant monitoring application uses an IMU which consists of three internal sensors gyroscope, accelerometer, and Hall-effect magnetic sensor. In low energy mode, instead of sampling all three sensors, we can gracefully compromise on gyroscope and Hall-effect magnetic sensors as accelerometer data is more crucial to infant monitoring, reducing energy burden and increasing throughput.

Sensor Sampling (SS). Another knob is the sampling rate and resolution. A sensed value may sometime be erroneous, due to unknown internal faults or extreme weather conditions. To overcome this problem, applications usually gather multiple samples, then exhaustively compute on collected samples or calculate their average before making decisions. As an example, a Correlated Sensing and Reporting application [14] collects N (32) magnetometer samples to keep track of magnet movement, where N is a fixed number and is not changed at the run-time. Such a static execution often leads to *non-termination*. By dynamically adjusting sampling rate and choosing different resolution with varying incoming energy, applications can be made more adaptive with their limited energy budgets.

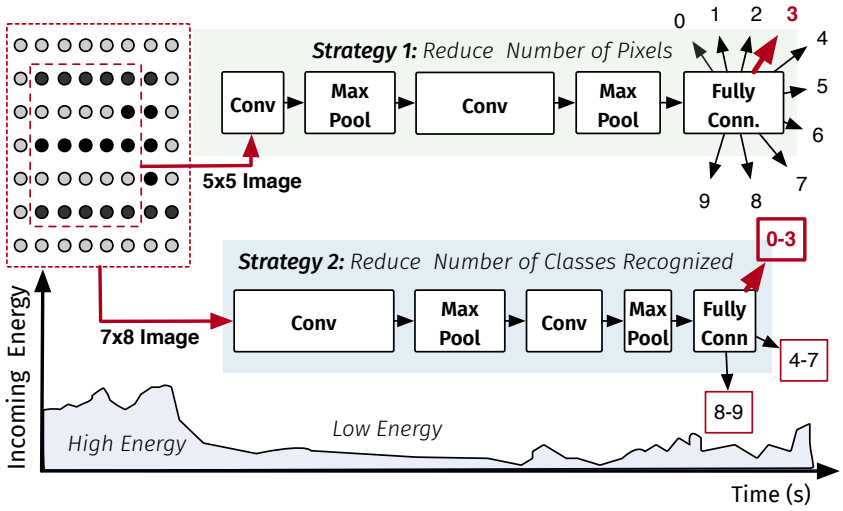


Fig. 6. For a machine vision application, two different adaptation strategies are shown: changing number of pixels used for classification (by cropping the image), and changing number of classes recognized. In the former, fewer pixels are used when energy decreases, which reduces accuracy across the board, in the latter, some classes are thrown out completely, which maintains accuracy by using all pixels, but loses distinguishability between objects. While both strategies are likely better than not adapting, the choice of *how* to adapt greatly depends on the application programmers goals.

Compute Complexity (CC). Computational load can sometimes be decreased at a loss of accuracy. As machine learning goes beyond the edge [25, 40, 50], computational load increases. Adapting compute complexity, computing over fewer samples, perforating loops, pruning layer or neurons in a network, early exiting the network [40, 65, 79], may compromise on the accuracy, but may make better use of available energy with at least some application quality.

Communication Variables (CV). Sending a packet has a high energy cost. Adaptation via network size, Low Power Listening (LPL) and other parameters provides options. Instead of communicating with the whole network, and sending/ receiving multiple packets, a sensor could decide to communicate with a small portion of the network (reduce transmit power) or send one inference packet instead of sending all data [67], reducing network traffic and thus overall energy consumption.

3.4 Bringing it all Together, Adaptation Strategy and "How to Adapt"

Finally, the application developers can take the chosen signals, the heuristic function that takes transformed adaptation signals to make an adapt up or down decision, the knobs and variables that they wants to change, and then designate tasks that will be adapted within the program. At this point, the programmer is merely making connections ("glue" code) between the heuristic adaptation approach and the tasks themselves. In addition to two potential AR adaptation strategies mentioned earlier (shown in Fig. 2), here, two strategies are shown in Fig. 6 for a machine learning application, one which reduces pixels analyzed, the other reduces classes recognized. Only the developer can make that choice.

4 USER EXPLORATION OF ADAPTATION: FORMALIZATION AND TOOLING

Having a mechanism and framework to adapt is not enough to enable building of robust intermittent computing applications. In this section we describe a new tool that complements the REHASH runtime system and heuristic adaptation framework. The tool allows a user to explore the impact of design decisions they make on program operation and quality of application– including changing adaptation strategies, tasks, or energy environment. The simulation tool called REHASH-`explorer`, is built on a formalization of the mechanisms of heuristic adaptation, allowing for a hardware free exploration of adaptation. The previous sections provided a informal description and motivation for the heuristics of our framework, and how all the pieces fit together. Here, we formalize our model of adaptation bottom up, from energy models, signals, and task graph based programs, use these as components to describe heuristics, and then integrate this modeling into a simulation environment that allows for users to explore and compare different adaptation schemes quickly and easily.

4.1 Energy Harvesting and Storage Model.

Adaptation adjusts the throughput of the system by considering the available energy, which is changing according to environmental dynamics. The objective of adaptation is to maximize system availability, by minimizing the end-to-end latency required for processing the tasks of the application. Therefore, how fast the system is charged by considering the rate of incoming energy and how fast the energy is consumed by considering the rate of energy consumption are key points.

An energy harvester's power output is not constant, but depends on the load on the harvester. This relationship between the harvester current I and the harvester voltage V, is captured by an IV curve which represents the state of possible power outputs at a point in time. Solar panel IV curves are presented in datasheets and have a distinct shape with a "knee" where the most power can be harvested. We denote the input power from the environment at time t as $P_{in}(t)$. Based on the received power, the IV function λ of the harvester can be defined as $I_{EH}(t) = \lambda(V_{EH}(t), P_{in}(t))$ where V_{EH} denotes the voltage provided by the harvester and I_{EH} is the corresponding current that is dependent on the IV characteristics of the harvester.

Let RC be the time constant of the energy storage capacitor. The voltage across the energy storage capacitor, being charged by the environmental energy, at a time instant $t_0 + \Delta t$ can be denoted as:

$$V_{cap}(t_0 + \Delta t) = V_{cap}(t_0) + V_{EH}(1 - e^{-\Delta t/RC}) \tag{1}$$

where we assumed that $V_{EH}(t) = V_{EH}$ is constant within the corresponding interval. Similarly, the harvested energy stored in the capacitor in the time interval Δt is denoted by:

$$E_{harv}(\Delta t) = 1/2C(V_{cap}(t_0 + \Delta t)^2 - V_{cap}(t_0)^2).$$

4.2 Application Model.

An intermittent application can be represented by a directed graph $\mathcal{G} = (V, E)$, where the vertex set $V = \{v_i | i = 1, \ldots, N\}$ represents the set of N tasks forming the application and the edge set $E \subset VxV$ represents the control flow among these tasks. Each task v_i has e_i, that represents the energy requirements to execute the task. We assume that the energy of a task is not fixed and it can shrink or grow depending on the adaptation strategy employed. For example, by reducing the number of samples captured in a task. For each task $v_i \in V$, we assume that:

$$\forall t \in V : e_i \in [e_i^{min}, e_i^{max}],$$

where e_i^{min} denotes the minimum energy and e_i^{max} denotes the maximum energy to execute task v_i at different adaptation levels. At the minimum programmer defined adaptation level, for example, a task might only sample a temperature sensor one time, the energy cost of the task is therefore e_i^{min}. At the highest programmer defined level, a temperature sensor might be sampled ten times and averaged, to have a more accurate reading. the energy

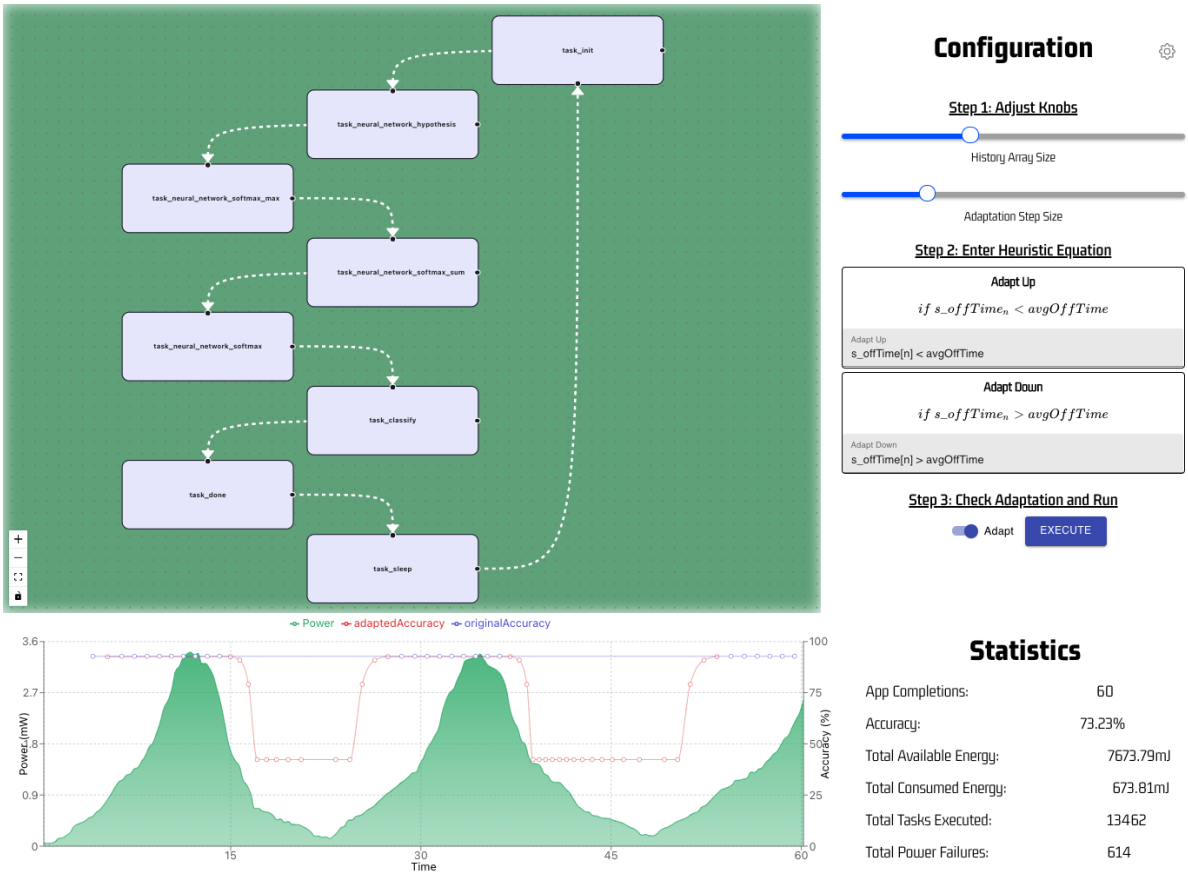


Fig. 7. REHASH-explorer screenshot. A selected application is rendered on the screen where the task graph of the program is shown. The sliders along the right-hand side of the window allow the level of adaptation by changing programmer defined adaptive variables (in this case skipping pixels of an image). The bottom left area of the screen displays the energy harvesting environment selected. Also shown is the accuracy of the image classified which is varying with changing harvested energy; markers (red or blue) show the time when classification is completed. The bottom right, summary statistics on the outcome of the simulation with the given task graph, adaptive variables, and energy environment. To view a full-size interactive demo of REHASH-explorer, please visit https://adaptationprofiler.github.io/adaptation-profiler/.

cost of the task is then e_i^{max}. This is further represented by a step-wise function for mapping the energy of a task to the current adaptation level q.

$$e_i^q = q * e_i^{min}$$

This mechanism is shown in Figure 3, where lower energy cost task replicas are executed when the adaptation level (q) changes. Moreover, t_i denotes the time it takes to execute task v_i having e_i energy demand. Assume that task v_i is the current task to be executed at time t_0. The energy in the capacitor at time $t_0 + t_i$ is denoted by:

$$E_{cap}(t_0 + t_i) = E_{cap}(t_0) + E_{harv}(t_i) - e_i$$

The voltage on the capacitor at time t_i is then denoted as a function of the energy and the size of the capacitor C:

$$V_{cap}(t_0 + t_i) = \sqrt{2\frac{1}{C}E_{cap}(t_0 + t_i)}$$

If $V_{cap}(t_0 + t_i)$ is lower than the operating voltage of the microcontroller running the application, the device dies before it can complete the task. In this case, the task must be retried when the device turns back on. Otherwise, the next task in the control flow is executed.

4.3 Signals Model.

Signals are physical or application based phenomena that change in response to the energy harvesting environment. These are often directly measured in hardware, or are application based statistics. The set of signals, denoted by the signal set $\mathcal{S}$, capture the dynamics of the energy harvesting environment. As an example, the time spent for charging the capacitor, denoted by s_{off}, is highly related with the input voltage as indicated by equation (1). Besides, on time, represented by s_{on}, indicates the energy requirements of the application. More specifically, if $V^* \subset V$ denotes the set of tasks that were executed during a power cycle, one can calculate on time as $s_{on} = \sum_{v_i \in V^*} t_i$ where t_i denotes the execution time of the corresponding task v_i. More signals can be produced by capturing other physical or application based phenomena that change in response to the energy harvesting environment, such as $s_{task} = |V^*|$ which denotes the number of tasks that were executed during the corresponding power cycle. These signals form the components of the heuristic, that is used to decide when to adapt.

4.4 Heuristics Model

Heuristics are logic statements composed of a mathematical combination of *signals*, that compute an outcome of either "adapt up" or "adapt down". The heuristic equation is an approximation of the energy harvesting environment. As shown visually in Figure 4, some signals will work better than others for predicting future energy for different types of applications, energy harvesting environments, or hardware platforms. Heuristics allow for combining signals to enable better predictive quality for heuristic adaptation. The outcome of the heuristic changes the value of the adaptive variables (i.e. number of samples to collect) which changes the energy of the task e_i.

A heuristic function f takes signals $s_0, ..., s_N \in \mathcal{S}$ as a parameter and outputs a binary value denoting either "adapt-up" (0) or "adapt-down" (1). Heuristic functions perform statistical/mathematical operations on the signals to infer the future energy availability of the environment based on past observations. An adaptation algorithm $\mathcal{A}$ considers the heuristics to look at the future and adjusts the task sizes to fit the energy requirements of the application into the harvested energy. In other words, this algorithm tries to provide a best effort in order to match the rate of incoming energy to the rate of energy consumption. As a heuristic predicting a complex situation, $\mathcal{A}$ can have a non-perfect prediction about the energy that will be harvested in the upcoming time interval using the heuristic functions. For an example, shown below is the *off time* heuristic function. This function takes as input the last n values of the *s_off* signal. The heuristic takes the average of the last $n - 1$ power failures time length, if the last measured off time is less than the history, then power failure times are decreasing, and the energy must be increasing. The inverse happens for adapting down. We include a hysteresis value, ϵ to add hysteresis to the adaptation activations. Essentially, if the power failure time is decreasing then we adapt up.

$$f(s\_off) = \begin{cases} adapt\_up, & if\ (s\_off_n - \epsilon) < \frac{1}{n}\sum_{i=1}^{n-1} s\_off_i \\ adapt\_down, & if\ (s\_off_n + \epsilon) > \frac{1}{n}\sum_{i=1}^{n-1} s\_off_i \\ do\_nothing, & \text{otherwise} \end{cases} \quad (2)$$

This is just one example of a heuristic function that a user can create. The *adapt_up* and *adapt_down* equations can be significantly different, for example if upgrading performance should be more conservative than downgrading. Multiple signals and varying lengths of history can also be used to make the heuristic function more dynamic.

4.5 REHASH-explorer: User Exploration of Adaptation via Simulation

From this formalization of the energy harvesting environment, application, signals, and heuristics, we can inform a user facing simulation platform, which we call REHASH-explorer. The knobs and potential inputs to an adaptive intermittent computing program are vast, making it hard to see broad trends or draw out conclusions from point based testing on real hardware. For this reason a simulation platform that can give broad strokes of how adaptation strategies work can be very helpful. The REHASH-explorer tool lets a user understand and explore the tradeoffs in different adaptation strategies and illuminates the interplay of energy harvesting environments, adaptation, and application tasks. We have developed the tool as a browser application, to facilitate straightforward, multi-platform, and frustration-free installation and deployment. A screenshot of the REHASH-explorer is shown in Figure 7.

The tool is a step based simulation, loosely modeled after MSPSIM [20] and the Mementos [69] and SIREN [22] additions to it which allowed for instruction level simulation, but aware of IV surfaces. Instead of granular instruction level simulation, which is architecture dependent, we focus on coarser grained task level simulation, which allows for any architecture and fits within our formalization. Based on hardware profiling, the minimum and maximum energy requirements of each of the tasks can be obtained. We assume that each task v_i has a set of knobs denoted by $\mathcal{K}_i$. As an example, these knobs can be the number of samples taken and iterations performed within a task. Based on the heuristics, $\mathcal{A}$ adjusts the knobs to adapt the task sizes using a coarse-grained approximation of the amount of energy harvested in the future.

REHASH-explorer tracks statistics throughout the execution of a particular configuration of the application, that will provide data and context for the user to then determine if the configuration under test was successful, or better than other configurations. These statistics build off the previous formalizing combining all the factors of the program and environment. They are:

(1) Total energy available.
(2) Total energy consumed.
(3) Number of tasks executed.
(4) Number of app completions.
(5) Number of power failures.
(6) Average accuracy of inferences performed.

With these results, the user can make a judgement call on the performance of the application and adaptation strategy, continue to tweak the heuristic function and use of signals, and compare and evaluate alternative energy harvesting environments or slightly changed tasks. This method enables a broad exploration of the space of adaptation, before any hardware has to be deployed, allowing for users to take advantage of the flexibility of the REHASH framework and have some level of understanding of all the tradeoffs.

4.6 Caveats

This formalization of heuristic adaptation, as well as the simulation platform, has some caveats to keep in mind when making decisions based on the tool outcomes.

Simulation Accuracy. Because the simulation works at a coarse, task level, and uses a simplified model of the actual physics happening in the real hardware, accuracy will not be comparable to a real world device. Even minor changes in capacitance would have an impact on the statistics mentioned. However, this is not the point; as the tool is meant to compare the range of configurations to each other, and understand the *proportion of change* as configuration variables are changed. A perfect accuracy (for example with energy consumed) is not required or really relevant, as what is more relevant is what is the magnitude of the difference in energy consumed between two configurations.

Minimum Quality of Application. Let $P = v_0, ..., v_N$ be a path in graph $\mathcal{G}$ and let e_P be the sum of the energy requirements of the tasks along this path. Let us denote $\mathcal{P}$ as the set of all possible paths whose starting node is the start task and end node is the end task of the application. Let the *maximum quality*, and in turn, energy consumption of the application be $e_{app}^{max} = \max_{P \in \mathcal{P}}\{e_P\}$ where for each $v_i \in P$ it holds that $e_i = e_i^{max}$. In other words, we select the maximum energy consumption for these tasks. Similarly, the *minimum quality* of the application is denoted by e_{app}^{min} where for each $v_i \in P$ it holds that $e_i = e_i^{min}$. Algorithm $\mathcal{A}$ performs adaptation to execute applications with best available quality. It is worth mentioning that, the application quality will not be degraded below the minimum quality. If energy does not exist to execute the application at minimum quality as defined by the user, then the application will never complete.

5 REHASH: RUNTIME AND TOOL IMPLEMENTATION

We now present the implementation details of REHASH, the first runtime for intermittently powered devices that provides explicit support for energy-aware and adaptive computing. This represents an instantiation of the REHASH framework, and we expect other instantiations (for other classes of energy harvesting devices) to follow. The architecture of REHASH runtime is shown in Fig. 9. REHASH is built on state-of-the-art InK [80] runtime system. With InK, developers specify tasks (which sense, compute, infer, or communicate) and link them together to establish control flow. Multiple tasks can be wrapped into a thread, and threads can be given priority levels. InK schedules these tasks and maintains memory consistency and progress of computation despite power failures. Our work provides the necessary functions to allow for adaptation, including signals, heuristics, and adaptation strategy constructs. By building applications in both InK and REHASH we can have a baseline comparison to a well established static (non-adaptive) runtime and programming model. We expect that any runtime system (even checkpoint based) could make use of the adaptive functions and methods described here.

5.1 REHASH Signals and Heuristic Function

We capture the adaptation signals from hardware and software, to enable heuristic adaptation. We consider three adaptation signals and recognize that these could be combined to form heuristic functions, along with transformations.

(1) On-Time. The on-time or active time is the measured time after recovering from a power failure (when sensing or computing starts) till the start of next power failure. This time can change dramatically based on the energy availability, as more energy means fewer power failures and longer on time.

(2) Off-Time. The off-time is the duration of time from a power failure to the successive recovery. This signal is usually (in practice) much longer than the on-time, as it takes longer to harvest energy than to spend it. This time can be captured using an RTC, or Remanence Based Timekeeper [34, 80].

```
/* adaptive variables. */
__knob(int numSamples = 10);

/* task-shared variables. */
__shared(int samples[10]; int i;);

/* adapts variables considering the heuristics */
__ADAPT_UP{
    /* increase the number of samples */
  __TUNE_UP(numSamples, STEP_SIZE);
}

__ADAPT_DOWN{
    /* decrease the number of samples */
  __TUNE_DOWN(numSamples, STEP_SIZE);
}

/* an adaptive task */
TASK(Sense){
  /* sample temperature sensor */
  int read = __sample_temp();
  /* samples[i] = read */
  __SET(samples[__GET(i)],read);
  /* increment i */
  __SET(i,__GET(i)+1);

  /* check if number of samples taken */
  if(__GET(i) < __READ_KNOB(numSamples)){
      /* sample again */
      NEXT(Sense);
  }
  else{
      /* next task -> compress samples */
      NEXT(Compress);
  }

}
```

Fig. 8. Overview of REHASH adaptation language constructs. The __knob keyword is used to define adaptive variables whose value can be modified only by __TUNE_UP and __TUNE_DOWN methods. __READ_KNOB returns the value of the corresponding adaptive variable.

(3) Task-Count. The task count is the number of tasks completed in the previous power cycle of the device. These tasks, defined by the developer, can be of any length or size. In general, as more energy becomes available in the environment, more tasks are expected to be completed (not accounting for energy differentials in tasks, for example, a task doing computation versus a task sending a packet on the radio).

While these signals we consider intuitively useful, we expect many other signals could be integrated into this heuristic adaptation framework. Importantly the signals track the maximum power harvestable in an energy environment, as shown in Fig. 4. This is important because this means these signals are information rich in estimating the current energy availability, while being very lightweight to gather.

Heuristic Function. Our framework maintains a history of these signals, which is developer tunable. The signal values are recalculated and then added to the history at each power failure. The heuristic function is defined in a single C/C++ file included in the runtime. This function can be rewritten in plain C/C++ and has access to all signals and their histories and all standard math libraries to form a interesting and useful heuristic function. The current implementation of the heuristic function used for the paper is based on one signal value selected by the programmer, see Equation 2 for the s_off (off time) based heuristic function. In this case, the heuristic function takes the average value of the signal using the history and compares the average with the current value of the signal in order to make an adapt up/down decision. The REHASH scheduler, based on the decision of the heuristic function, calls the necessary adaptation up and down routines described in the adaptation logic of the application.

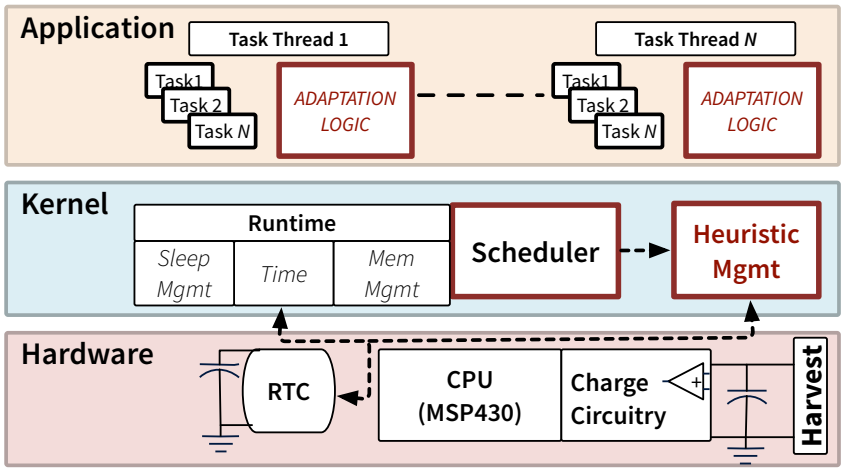


Fig. 9. REHASH high level architecture. The kernel and runtime ensures progress of computation and memory consistency; meanwhile heuristic management block collects necessary information that will be used by the application. Adaptation logic uses the heuristics provided by the kernel and performs adaptation. We note in red the major blocks.

5.2 REHASH Language Constructs

In InK, the programmer annotates task-shared persistent variables by using `__shared` block, provides task implementations by using `TASK(...)` and indicates control flow by `NEXT(...)` keywords; respectively. Since the task-shared variables are double-buffered so that tasks become atomic, the access to these variables happens via a `__SET` and `__GET` interface. In addition to these language constructs, REHASH provides additional keywords to enable adaptation for applications. Fig. 8 presents a sketch application that is developed by using the language constructs provided by REHASH runtime.

Adaptive Variables. In REHASH, the programmer annotates the persistent adaptive variables by defining them using `__knob` interface. Unlike task-shared variables, these variables can only be modified by the adaptation logic and they are **read-only** for the tasks. This simplifies the programming interface and removes the need for the runtime to anticipate or inform the tasks about changes to energy environment.

Adaptation Logic. The `__TUNE_UP` and `__TUNE_DOWN` interfaces are used to modify adaptive variables. Based on the output of the heuristic function, the REHASH scheduler calls either `__ADAPT_UP` or `__ADAPT_DOWN` routines within the adaptation logic.

Adaptive Tasks. The tasks that are adaptive should include decision logic to make inferences by reading the updated adaptive variables—the `__READ_KNOB` interface reads the value of an adaptive variable. In our example, the adaptive `SENSE` task checks the value of the `numSamples` variable in order to sample more or to change the control flow for compressing the sensed data.

5.3 REHASH Kernel

The general architectural overview of REHASH is presented in Fig. 9. We implemented a *Heuristic Mgmt* component to measure the signals at each power cycle (or reboot). The values of these signals are maintained in non-volatile memory by using a FIFO *queue* data structure that has a predefined capacity—a history of measured signals is provided to applications. At each reboot, in the initialization steps of the REHASH kernel, the value of the persistent timekeeper is measured and the value of the off-time is appended to the corresponding queue of

Algorithm 1: REHASH-`explorer` main simulation loop.

Input: Developer configuration, $Tasks$, $Environment$, $Inputs$, $Heuristic$.
Result: Performance outcome from user configuration

```
/* Initialize and load the inputs                                          */
Tasks ← load_application()
Environment ← load_energy_environment()
Inputs ← load_user_inputs()
Heuristic ← load_heuristic()
/* Application quality starts at lowest setting                            */
Quality ← minimum
/* Main step-based simulation loop                                         */
while time_sim < time_end do
    Power_env ← Environment.get(time_sim)
    V_cap ← update_capacitor(Power_env, V_cap)
    Signals ← calculate_signals()
    if V_cap > V_operating then
        /* Determine the next task to execute, and the level of quality to execute at.  */
        r ← Heuristic.execute(Signals)
        if r ≡ adapt_up then
            Quality ← Quality + 1
        end if
        if r ≡ adapt_down then
            Quality ← Quality − 1
        end if
        V_cap ← execute_next_task(Signals, Quality, V_cap)
    else
        die()
        rollback(Tasks)
    end if
    /* Capture performance statistics: power failures, tasks executed, etc.  */
    gather_statistics()
    time_sim ← (time_sim + time_step)
end while
/* Report statistics to the user interface                                 */
report_statistics()
```

the *recovery-time* (or off-time) based heuristic function. For *active-time* (or on-time) based heuristic functions, a periodic timer is set to generate interrupts at regular time intervals. Upon each interrupt, the value of the corresponding persistent variable is incremented. At each reboot, the value of this variable is multiplied with the timer period to obtain the on time and this value is appended to the queue of the active-time heuristic. At each task completion, a persistent variable holding the count of successfully executed tasks is incremented. The calculation of the heuristic function as well as triggering the adaptation logic by the scheduler can be performed either at each reboot, at the completion of the thread or application. This can be configured by the programmer. The adaptive variables are not double-buffered since they can only be modified by the adaptation block. Therefore, write-after-read dependencies [78] and the need for two-phase committing these variables are eliminated.

5.4 REHASH-explorer

The simulator is a browser viewable tool. The simulation main loop is shown in Algorithm 1. The user is expected to supply the task graph, annotated with energy and time values for the base non-adaptive case (i.e. the minimum quality setting). These values can be generated using EnergyTrace++ as used in other works like Zygarde, SONIC/TAILS, and others. The user is also expected to select the adaptive variables and associated tasks they would like to use. The energy harvesting environment must be chosen, this can be gathered from the existing IV surfaces available in the tool, or a new IV surface added. Finally, the heuristic function is either selected from the existing functions (one of which is the history of off time, as show in Equation 2), or the user may use the heuristic designer to create a new function using the signals available.

Main Loop and Task Execution. The main loop evaluates the state of the task execution every time step, keeping track of the progress of the program, the current part of the energy environment, and the adaptation settings. These all go into calculating the capacitor voltage, which determines if a task can be executed successfully or not. Signals are captured at each time step, and are fed into the the heuristic function before any task is executed, to inform the adaptation level. When a task is executed, the energy cost of the task, calibrated to the adaptation level, is removed from the capacitor. If the energy cost causes the voltage on the capacitor to go below the operating threshold, then a power failure is registered and a rollback occurs. As the simulation proceeds, statistics are gathered to inform the user of the success/failure of the configuration under test.

User Input and Interaction. The interface allows for significant flexibility in designing a configuration for an adaptive intermittent computing application. The user may change the heuristic function, even designing one there self using the available signals, providing the most direct way to set an adaptation strategy. The user can also change the adaptive variables and tasks, for example, one configuration might adapt the sensor sampling, while another the amount of communication, these configurations can be compared against the outcomes such as power failures and task count, to help the user decide which is better for their purpose. Finally, the user can try the configuration on multiple energy harvesting environments (IV surfaces), which could represent indoor and outdoor light, kinetic, thermal, RFID, something as fine grained as solar harvesting in a specific location and time, or any other energy harvesting scenario.

Built With. We built the simulator platform using JavaScript and web based tools, to allow for the fastest deployment and use by programmers. Often, it takes significant time and expertise to get intermittent computing software, hardware, and tools running. We skip the install burden by enabling everything in the browser. We used ReactJS, with JavaScript libraries based on KaTeX for rendering and evaluating heuristic equations. These functions can be written in regular mathematical expression format and are converted on the fly in the web app to a rendered picture (as shown) and an object which can be evaluated for a output (adapt up or down) within the simulation script.

5.5 Developing Applications with REHASH and REHASH-explorer

The REHASH approach gives the flexibility to write any kind of adaptive intermittent applications and the tools to understand the impact of configuration and adaptation decisions before deployment. We now describe how a user would go about building an application leveraging the REHASH runtime system and the fine tuning the application using the REHASH-explorer. The workflow is shown in Figure 10. We also provide some rules of thumb on writing adaptive applications. The first duty of the programmer is to provide a task division and annotating the shared variables among tasks (just as in InK, Alpaca, and every other Task based runtime). After this, the programmer will:

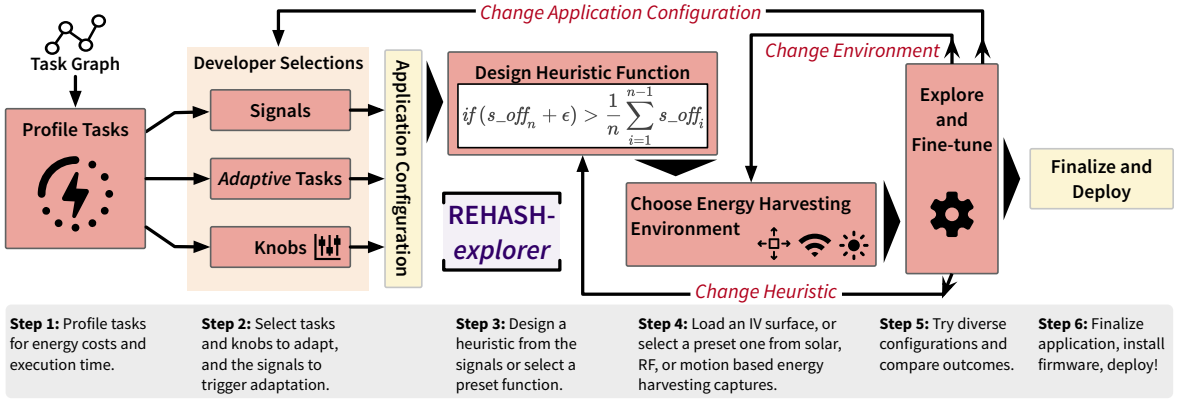


Fig. 10. Typical workflow for a developer writing applications with REHASH and REHASH-explorer.

(1) **Profile Tasks** for the energy costs, and execution time, these values are used by the tool. These profiles are at the base level (or most degraded) quality— REHASH-explorer will use this baseline to calculate upgraded tasks energy cost.
(2) **Identify the signals** that are supported by the specific hardware platform or software. For example, not all hardware platforms will have an remanence based timekeeper which allows for measuring off-time, however, every platform will have a way to measure on-time using built in timers.
(3) **Identify the adaptable parts of the program**, based on the taxonomy given in Fig. 5. This means selecting tasks, and the adaptive variable (knobs) that will be exposed to the tool for simulation.
(4) **Load into REHASH-explorer** the built application task graph, adaptive variables, and task profile.
(5) **Pick the set of energy environments** to test on from the selection available or a custom set.
(6) **Choose the heuristic function**, or build one using the heuristic designer.
(7) **Explore the configuration space** by changing the sliders for the adaptive variables and seeing the change in statistics (i.e. power failure, tasks complete) using the tool until a suitable, robust version of the program is found that satisfies the programmer constraints and definitions of quality.
(8) **Iterate or finalize the development.** The programmer can go back to any previous point and tweak the system inputs and then explore the tradeoff space. Eventually the program is finalized, installed on a real device, and deployed.

Overall, this adaptive approach is not much more demanding than typical task based programs, but can provide a mechanism where applications are much less brittle when energy environments have larger dynamic changes, and will give confidence and more understanding via user exploration of the adaptation schemes.

6 APPLICATIONS

We experiment with three applications to demonstrate and test the heuristic adaptation approach: Activity Recognition (AR), Greenhouse monitoring (GHM), and a machine learning application that recognizes handwritten digits from MNIST [48] (ML) in 8KB of memory. For each application we developed a baseline, non-adaptive version for the unmodified InK runtime [80], then ported that application to REHASH. This provides a solid baseline comparison for understanding the benefits of the adaptive version of the corresponding application over that of the non-adaptive version. Below we provide the adaptation strategies used for each application.

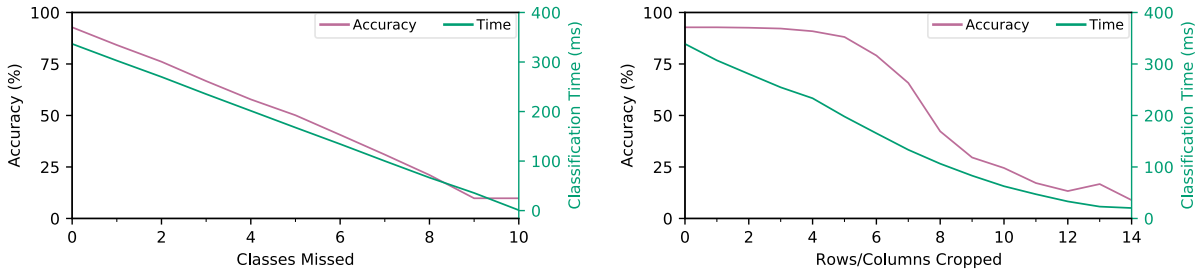


Fig. 11. Two different adaptation strategies for MNIST image classification application, and their effect on execution time is shown. When changing the number of classes recognized (on the left), the behavior of accuracy and time is linear as the dataset has almost equal number of images for each digit. When changing the number of pixels used for classification (by cropping the image from the borders), the rate of change in accuracy is smaller than the rate of change in execution time. This is because most of the information in MNIST dataset lies in the middle pixels, so cropping the image from borders does not affect accuracy much.

6.1 Activity Recognition (AR)

AR is a human physical activity classification application. During high energy execution (*send-all mode*), AR collects 16 accelerometer samples, buffers into a sliding window and sends all the data to a base-station that classifies it to five activities using a complex neural network and logs it to a database. The number of samples are halved each time a degrade signal is received from the REHASH. If the REHASH still sends degrade signals, AR switches to *send-class* mode by taking only 4 samples and sending the classification a simple two-class classifier, i.e. only one packet per execution. If this adaptation is still not sufficient, i.e. REHASH still sends degrade signals, AR takes 4 samples, classifies them as moving or stationary, and sends a packet only if moving is detected (*send-event* mode).

6.2 Greenhouse Monitoring (GHM)

Greenhouse monitoring (GHM) is a sense-and-send application that uses multiple sensors to approximate the health of plants in a greenhouse; from soil moisture readings, sunlight, and temperature. During *send-all* mode, GHM takes 128 samples and sends all of them. If adaptation is needed, the number of packets and samples are halved each time until a lower threshold is reached (4 samples per execution). Degrading further, it takes 128 samples, because sensor coverage is more important in this case, takes the average, and sends only one packet per iteration of the application (*send-avg* mode). After that, a packet is only sent if it is above a limit (*send-event* mode). This notion of different modes of execution, that are modulated with number of samples and packets, gives a nice flavor of adaptation that is trivial to implement with REHASH.

6.3 Digit Recognition ML Example (ML)

We consider the MNIST dataset [47] classification as a computational benchmark for understanding heuristic adaptation. We choose MNIST because large images do not fit in a resource-constrained device's memory. In a real-world perfect scenario, a trained model could be ported to the MCU, that takes an image using a camera and then classifies it in real-time. We build a toy implementation where we load the model and some images on the microcontroller that was powered by harvested energy while evaluating REHASH, similar to [25].

We first explore different adaptation strategies on a MSP430FR5989 microcontroller when run at 16MHz. For the purpose of this exploration only, we run the MCU on continuous power. We try changing the number of classes recognized and observe its behavior on accuracy and time. Both accuracy and time decrease linearly

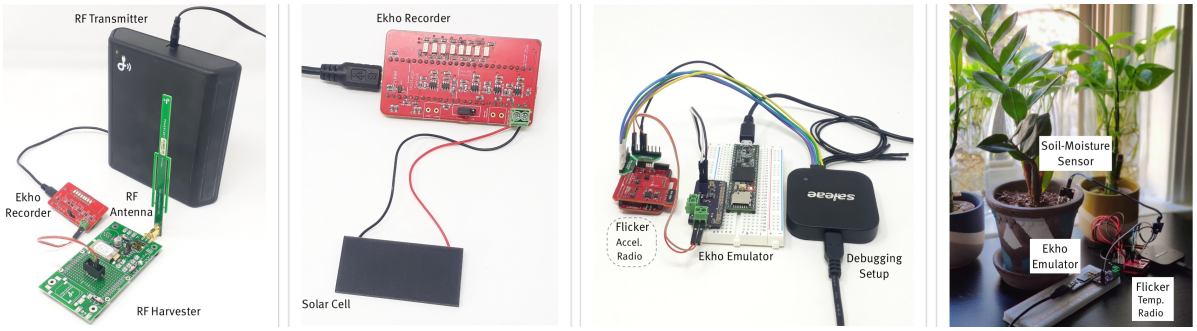


Fig. 12. REHASH experimental setup.

when number of classes recognized are decreased as shown in Fig. 11. The is because of the uniform image distribution in the dataset, meaning that all the digits (0 to 9) have almost equal number of images. This strategy loses distinguishability between objects, but if the user has more confidence in some classes that can be specified to get higher accuracy. Therefore, this adaptation strategy could be useful when some classes have a higher probability than others. The other adaptation approach is changing the number of pixels used for classification by cropping the image before passing it to the neural network. This strategy slowly degrades the accuracy while decreasing the execution time at a higher rate, thus resulting in more classification in a shorter period of time. Since the images in MNIST dataset have most of the information in the middle, this may not be a good strategy for cases where the camera or object is in motion while the image is being captured. While both strategies are likely better than not adapting, the choice of how to adapt greatly depends on the application needs. A visual representation of both these strategies is shown above in Fig. 6.

We used both these strategies for evaluating REHASH. In the non-adaptive case, REHASH computes over the whole 28×28 image and then classifies it as a digit (0-9). If adaptation is needed, depending on the user preference, it may skip one row and one column from the boundary of the image at a time, and keep decreasing until the lower threshold (14×14) of the image size is reached, or keep reducing / coalescing the number of classes (i.e. digits recognized) until a threshold is reached. We refer to former strategy as *skipping pixels* and the latter as *reducing classes* in below sections.

7 EVALUATION

We conducted an evaluation of our instantiation of the REHASH framework. The goal of the evaluation is to understand the benefits and drawbacks of adaptation in terms of performance (defined differently by each application), the amount of flexibility and generality of the framework, and the overhead and user experience.

7.1 Methodology

We investigate aspects of adaptation in intermittent computing by running real world applications (described in Section 6, in diverse energy environments, and with different types of heuristic adaptation leveraging our proposed adaptation signals (*on-time, off-time, and task count*). Key to our approach is that we emulate energy harvesting environments precisely using Ekho [28, 29] and Shepherd [23], so that our experiments are repeatable. Without realistic emulation of energy harvesting environments, experimentation would have no controls, as devices would be executing in slightly different conditions, as an example, without Ekho and Shepherd, different executions of the same program would have different numbers of power failures, even if the energy environment did not seem to change. Using Ekho and Shepherd our comparisons are fair across heuristics and applications. We compare against only the non-adaptive InK runtime as other adaptive task-based systems are rigid, e.g. [58, 59],

and do not support the implementation of our flexible adaptation strategies for the applications mentioned in Section 6.

Devices Used. We use Flicker [32] as our batteryless platform for experimentation, as it has support for multiple peripherals (like an accelerometer, radio, and soil moisture sensor), has energy harvesting management circuitry, and has sufficient compute ability to run many interesting applications. We also use a MSP430FR5994 Launchpad [38], with external power management circuitry, to show the portability of REHASH to other platforms. Our Ekho emulator device acts as a energy harvester on the Flicker inputs and provides repeatable power for all activities (Fig. 12), while Shepherd emulates the power for MSP430FR5994 Launchpad.

Energy harvesting environments. We use five IV surfaces for measuring adaptation, which offer a broad range of different potential situations encountered in the wild. Fig. 12 shows our experimental setup for recording, emulation and application testing. We first record IV surface, using Ekho, produced by a Powercast harvester [16] and a 1.85in × 0.9in IXYS Solar Cell [53]. We name these IV surfaces as, (i) *Fixed Low:* recorded by the Powercast harvester where transmitter was far away from the harvester, (ii) *Fixed High:* recorded by the solar cell when placed near a light source in indoor environment, (iii) *Moving Slow:* recorded by the solar cell when the person carrying it is walking slowly inside a building, (iv) *Moving Fast:* recorded by the solar cell when the person carrying it is walking fast inside a building, (v) *Random:* recorded by the solar cell when the person carrying it is moving randomly inside a building. Each of these IV surfaces may manifest in the wild; for example the moving IV surfaces would be encountered in solar environments where sunlight increase or decreases during the day. We attempt to increase variety and understand how different adaptation heuristics respond to changes in energy availability.

Adaptation Performance Metrics. We compare the executions of our applications on Flicker devices with the following metrics, (i) *Total Executions/App Completions:* the number of times the critical part of the task graph of an app is successfully executed (which can span many power failures), (ii) *Accuracy (MNIST):* the averaged inference accuracy, which decreases as the computation is degraded, (iii) *Sensor Coverage:* the percentage of time that is covered by a sensor reading, (iv) *Performance (AR/GHM):* the percentage of samples (or adaptation) relative to the 100% performance baseline. We extract the metrics using external instrumentation; a Salea logic analyzer, Keysight Scope, and Ekho tools.

7.2 REHASH Performance

We ran each adaptive application multiple (5) times on each IV surface, comparing to the non-adaptive runtimes InK and Alpaca [56]. We compare against former as REHASH is built on top of InK and choose latter since it is used by many other intermittent runtimes like Celebi [41], ePerceptive [65], and Zygarde [40]. We choose InK as our baseline for all experiments. The app completions are counted and compared. The results are shown in Fig. 13, the higher bars mean more throughput, the green numbers in the bars note the percent increase (against InK) in throughput, while the red numbers designate the decrease in accuracy or samples captured. We implemented MNIST and GHM with timekeeping on for all type of executions i.e., adaptive or non-adaptive, and kept it off for AR for all executions except for when *off-time* heuristic is used. This is why adaptive AR has fewer executions than non-adaptive for the fixed IV surface—due to the timekeeping overhead.

REHASH has higher numbers of executions than both InK and Alpaca. This happens by compromising on either accuracy, samples taken, or packets sent per execution. Figures 13a, 13b and 13c show the performance of MNIST, GHM and AR respectively. In case of *fixed low* and *fixed high*, REHASH has no to minimal overhead. This is because of the heuristic-based adaptation which works when energy is changing. Since adaptation is triggered based on the previous data and has no static energy limits, with fixed energy REHASH did not adapt. REHASH always increases app completions with all other IV surfaces. In all cases, the increase in app completions

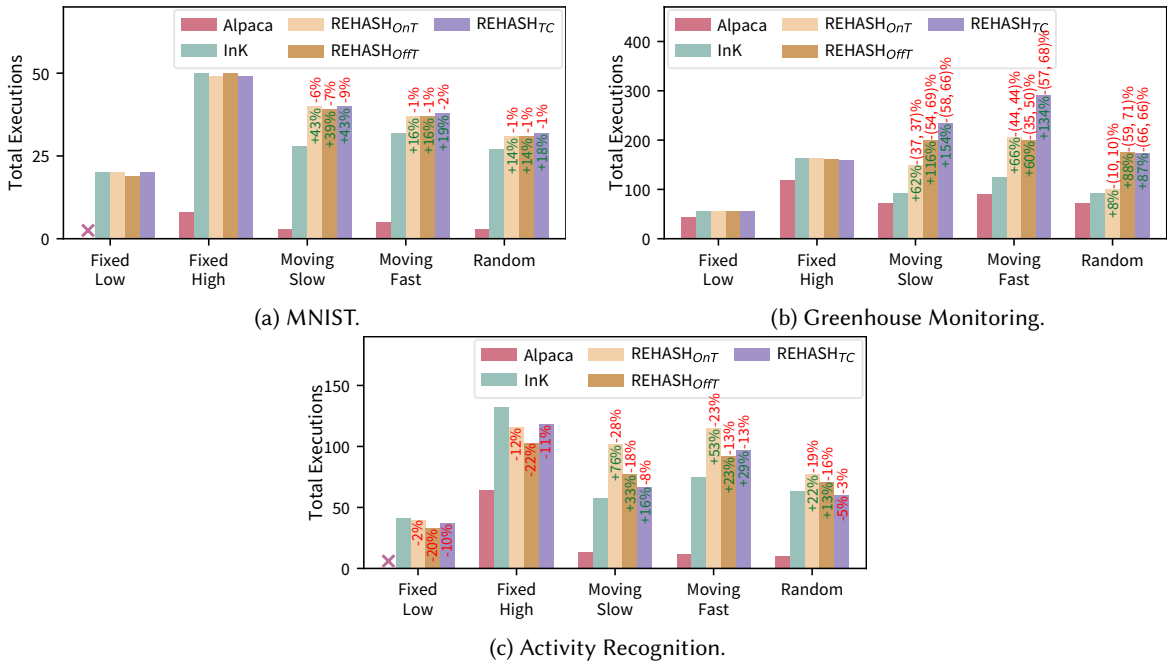


Fig. 13. MNIST, AR, and GHM applications performance compared to non-adaptive executions with InK and Alpaca. The numbers in bars denote the increase in app completions (w.r.t. InK) and the number on top of the bars denote the classification accuracy loss (MNIST), or number of samples and packets reduced (AR,GHM). As an example, from 13a, it can be concluded that *MNIST had 43% more completions with only a 6% drop in accuracy on moving slow IV surface*. As a conclusion, REHASH outperforms non-adaptive InK in dynamic energy environments with only nominal loss of accuracy and sampling.

is significant compared to the nominal decrease in accuracy, samples taken, or packets sent; at least for the applications represented.

Also, notice that the app completions with InK are much higher than Alpaca. This could be partly because of the different compiler used by both runtimes. Alpaca is used out of the box and is compiled using GCC while InK is compiled using TI's MSP430 Compiler. To minimize its effect, we keep the compiler optimizations off for both. But the major reason for the difference between app completions is the shorter task transition times introduced by InK. InK uses DMA (direct memory access) block copy during task commit operation upon finishing task execution. Alpaca traverses a commit list to commit the task-modified variables one-by-one, which introduces more overhead. Nevertheless, REHASH performs better than both runtimes.

Choice of Heuristic: Both *on-time* and *off-time* heuristics perform better than non-adaptive executions of InK and Alpaca for all applications and for all dynamic IV surfaces. However, *task-count* shows different behaviour in each application. Since MNIST is just a computational toy application and does not involve any real sensor or camera, there is no I/O involved which results in fixed task sizes. Therefore it works better in MNIST. Whereas, both GHM and AR are real world sensing applications and use sensor and radio module which involve I/O operations. This results in dynamic task sizes. It is nearly impossible to profile tasks before deployment and set task weights for a weighted task count. One task that takes 1ms at compile time, may take tens of milliseconds when deployed in the wild. So, *task-count* may not be a good option for real-world sensing applications. This insight has been overlooked in Coala [59] as used as the only heuristic for any type of application.

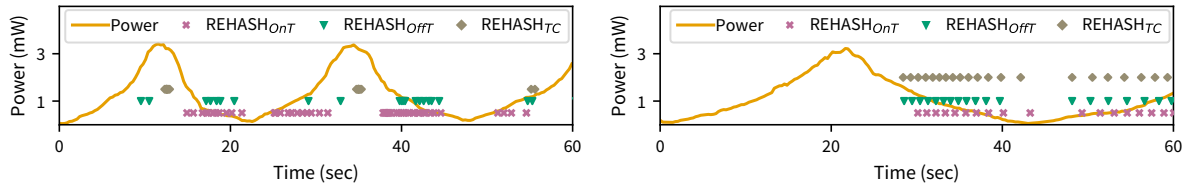


Fig. 14. Heuristics are timely in triggering adaptation signals. Shown are markers representing the time when adaptation is triggered for AR (left) and MNIST (right) applications when run on two difference IV surfaces, following the trend of max harvestable power and adapting when is needed

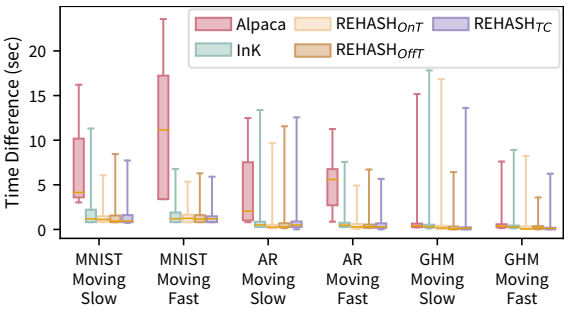


Fig. 15. REHASH reduces the time between two consecutive executions, increasing sensing coverage.

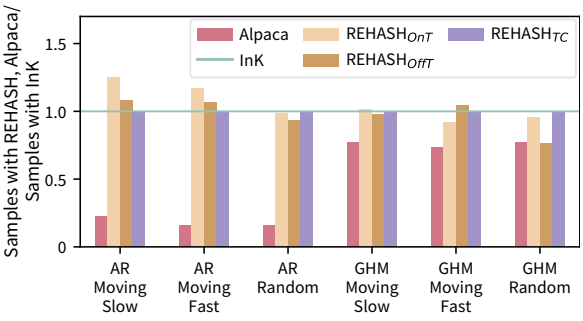


Fig. 16. Ratio of sensor samples taken with REHASH and Alpaca to the sensor samples taken with InK. With InK as a baseline, the number of samples taken for AR and GHM with REHASH is nearly the same as with non-adaptive execution of InK, but much higher than that of Alpaca since computation and communication are degraded in favor of sensing.

Adaptation Triggering Position. As stated earlier, the signals used by REHASH efficiently capture MPP. With the diverse set of applications that we use in our measurement study, where I/O and high energy radio operations are involved, heuristics may trigger adaptation at the wrong time when it is not needed. We therefore wanted to see when adaptation is triggered in REHASH. Fig. 14 shows a test run of AR and MNIST on *moving fast* and *moving slow* IV surfaces, respectively. While *task-count* heuristic does not trigger adaptation at all desired points in AR because of it inability to capture the effect of high energy I/O operations, both *on-time* and *off-time* heuristics accurately trigger down-adaptation when power decreases, and up-adaptation when increases for both applications.

7.3 Sensor Coverage

The non-adaptive greedy approach of InK and Alpaca results in task starvation where samples are taken in bursts when high energy is available. It may not be useful to send these samples as they could be expired when in low energy mode. Throughput (or total samples collected) does not always correlate to application quality. Ten samples all gathered within one second are not very useful, or worse are redundant and a waste to gather. Often one sample is enough for every period of a few seconds, or minutes, or even hours depending on the application. We use sensor coverage [75] as a measure of how spread out samples are. We look at time between two app

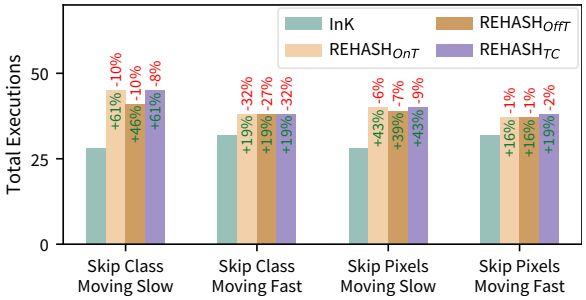


Fig. 17. REHASH is flexible and lets programmer add any adaptation strategy. Two adaptation strategies are tested for MNIST application. Skipping pixels or dropping classes. The number in bars represent the increase in app completions and the number on top of bars represent the drop in average accuracy for all classifications.

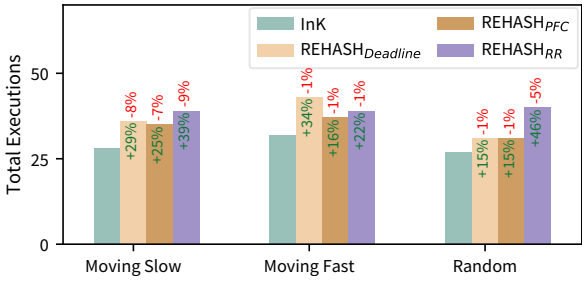


Fig. 18. REHASH allows the addition of new signals and heuristics. More power failures (*power-failure-count* or PFC) and a deadline miss indicate low energy conditions, which means the application should adapt down. REHASH, with these signals and heuristics, makes more classifications for MNIST application showing it is able to capture the change in energy availability.

completions/classifications as way to measure sensor coverage. This sensor coverage metric is important for motion event detection or continuous monitoring applications like those seen in infrastructure monitoring.

We use GHM and AR with dynamic IV surface to see the behavior of sensor coverage with our proposed heuristics. Fig. 15 shows the time difference between two consecutive app completions (keeping in mind that data becomes available when app is completely executed). All heuristic based adaptive executions of REHASH show a smaller time difference. Another way to understand the notion of better sensor coverage is to look at the number of total samples taken within a unit of time. If the number of samples are similar to the non-adaptive case, but with more app completions, than that means better sensor coverage. Taking InK as a baseline for this experiment, Fig. 16 shows the ratio of total samples taken with REHASH and Alpaca to InK for AR and GHM when run on dynamic set of IV surfaces for one minute. REHASH performs better or similar in most of the cases.

7.4 REHASH Flexibility

REHASH offers programmers the flexibility to write any adaptation strategy or create any type of custom heuristic function to trigger adaptation, beyond those built in. We approach flexibility in three different ways and demonstrate how REHASH and its heuristics make the development process of adaptive applications much more easier and flexible.

Adaptation Strategy. Two example adaptation strategies are previously described in Section 6. Fig. 17 shows the performance when these strategies are used for MNIST (skip pixels, or reduce classes). In REHASH's implementation of MNIST, it seems skipping pixels intelligently is a better option if higher overall accuracy is desired. But if the user has more confidence in some classes then that can be specified as well so others can be skipped when adaptation is triggered. This may increase expected accuracy.

New Signals. REHASH uses three signals: *on-time*, *off-time*, and *task-count* for adaptation. But many other signals stem out of the intermittent execution of the applications. For instance, i) the number of power failures that occur within one app execution cycle, ii) the total execution time of the application, or iii) the intermittency rate; all are directly proportional to energy harvesting conditions. These signals are either totally new and can be easily embedded in REHASH, or utilize other basic signals already embedded in REHASH. For instance, the power-failure-count is a new signal that is kept in a non-volatile integer which increments at each power failure. An app-completion-time signal can be constructed by adding both the on-time and off-time signals. A

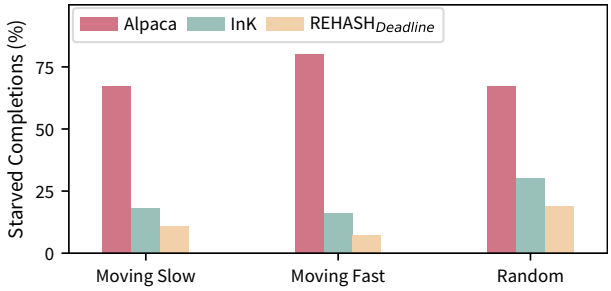


Fig. 19. The percentage of completions that could not meet the deadline for MNIST appliaction. This directly related to task starvation and REHASH reduces this number of by downgrading the workload.

recovery-rate signal can be obtained by dividing off-time and on-time signals. REHASH allows the addition of such signals without too much effort. We use these new signals and make simple heuristics as discussed in previous sections but with slightly different parameters this time. We discuss the behavior/performance of these heuristics below.

Custom Heuristic. REHASH also allows programmers to add any heuristic that can trigger adaptation by either using the signals that are already embedded in REHASH or by introducing new ones. Here we use three new signals described above and make slightly different heuristics than before to show flexibility. For the app-completion-time signal, we make a *deadline* heuristic that triggers down-adaptation if a deadline is missed. Though it is a fixed number but can be changed at runtime. For the MNIST application, the requirement is to classify an image every 2.5 seconds. Also, we don't want to trigger up-adaptation whenever it meets the deadline as it will keep fluctuating between different adaptive states; we rather want to see the trend of harvested energy. Therefore, we compare the current app completion time with the average of previous ones. Similarly, two other heuristics: *power-failure-count*(PFC) and *recovery-rate*(RR) use the signals described above, respectively. However, due to their volatility, we increase the history size to 4 instead of 2.

Fig. 18 shows total executions completed for MNIST when run on dynamic IV surface for one minute. Again, the adaptive execution with REHASH results in the classification of more images than InK, with a small loss in average accuracy. This shows that these new signals and heuristics are able to capture the changes in energy harvesting conditions. One thing on *deadline* heuristic and REHASH, in general, to notice here is that they don't make up a real-time scheduler. There may still be deadline misses, but REHASH can surely detect them and trigger appropriate actions. While more new heuristics can be made using available signals, or these heuristics can be explored further for more applications; the aim here is to show the flexibility and portability of new signals and heuristics in REHASH.

7.5 Task Starvation

Tasks in intermittent computing applications always have the risk of starvation since there could literally not have any energy to make forward progress. But this condition can certainly be detected, and appropriate actions can be triggered to reduce the execution time and thus task starvation. With REHASH, we provide a mechanism to address task starvation though it cannot be eliminated. The deadline heuristic we made in 7.4 shows that we have a flexible system. But this also shows one way to design a heuristic to focus on only task starvation. As a deadline miss could most likely occur due to task starvation, we count the number of deadline misses against total app completions to see how many times tasks have likely starved. Fig. 19 shows that the *deadline* heuristic

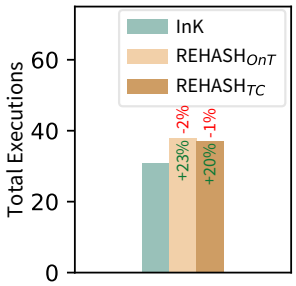


Fig. 20. REHASH's performance on MSP430FR5994 for a run of MNIST application. REHASH heuristics are platform independent and efficiently capture the changes in harvested energy and inform the *adaptUp* and *adaptDown* decisions. Executions with *on-time* and *task-count* heuristics result in more classification with a very minor drop in accuracy.

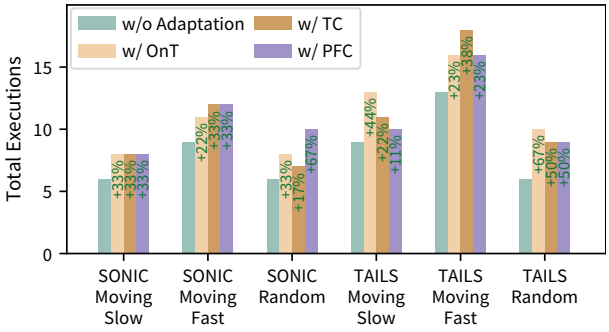


Fig. 21. REHASH heuristics are imported to SONIC and TAILS image classifiers that are built on top of Alpaca. *Skipping pixels* adaptation strategy is used that is able to make more classifications. The number in bars represent the increase in the number of classifications made in one minute compared to the non-adaptive baseline.

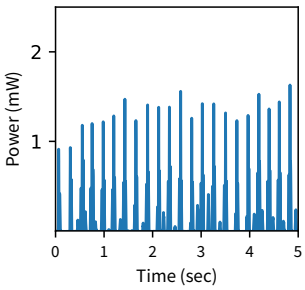
of REHASH reduces task starvation significantly. This simple heuristic shows how REHASH can help detect task starvation. One could be more aggressive with adaptation strategies and degrade significantly, or radically change the task, i.e., reading the accelerometer instead of a radio transmission.

7.6 REHASH Portability

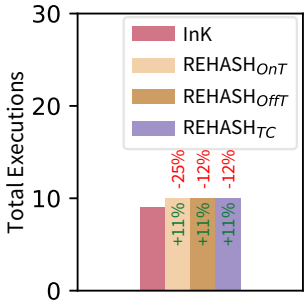
Heuristics used in REHASH are platform independent and work well on other platforms. We show portability in two ways. Firstly, we take REHASH as a whole (as a runtime), compile it for a different hardware (MSP430FR5994) and tests its usefulness with two heuristics. For this purpose, we first record a different power trace, of a 1.85in $\times$ 0.9in IXYS Solar Cell [53] again, with Shepherd this time, and then emulate it so our experiments could be repeatable. We then port the MNIST application, which needed very little modifications, and run the *skipping pixels* adaptive version against vanilla InK implementation.

Fig. 20 shows that the total executions with REHASH are significantly higher than InK with nominal loss in accuracy. For this experiment, we only evaluate *on-time* and *task-count* heuristics because the MSP430FR5994 Launchpad does not have a persistent timekeeper, like other battery-less sensing platforms, that can be used to keep track of *off-time* during power failures. While an external timekeeper or a RTC could be used to time the power outages if *off-time* heuristic is needed, our goal here is to show that the heuristics, and REHASH in general, are portable to other platforms.

Secondly, we use the heuristics library of REHASH and port it to existing intermittent runtimes SONIC and TAILS [25] that use the MNIST application, and are built on top of Alpaca. SONIC uses a loop continuation technique to reduce the cost of loop-heavy DNN applications and TAILS uses MSP430 hardware accelerator to improve energy efficiency. We use three heuristics: *on-time*, *task-count*, and *power-failure-count* as they are hardware agnostic, and test SONIC and TAILS with dynamic IV surfaces that are emulated by Ekho. Fig. 21 shows the total number of classifications made on each IV surface with and without adaptation. *Skipping pixels* adaptation strategy is used whenever needed; though *skipping pixels* could have been used instead. More classifications with adaptive executions show the effectiveness, generality, and portability of REHASH heuristics to any other intermittent runtime.



(a) Power traces of the magnetic coil while running.



(b) The *on-time* heuristic is unable to capture the changes in energy because of the energy impulses. Although adaptive run leads to more executions, the number of samples are reduced by a higher amount. The *off-time* heuristic triggers adaptation efficiently and results in a better response

Fig. 22. REHASH's response to impulse based harvesters.

7.7 REHASH with Novel Energy Harvesting Sources

We explored how our heuristics behave when in exotic energy harvesting environments. We wanted to understand whether our proposed heuristics can capture the change in harvested energy for harvesters beyond the ambient (such as RF and Solar) that tend to give varying, but continuous, levels of power. To investigate a different type of harvester, we used a slightly different approach to harvest energy. We designed a magnetic coil with permanent magnets attached to a 3D-printed case that harvests energy when shook by hand or human body movements. We attached this to the hand and connected Shepherd with it to record the harvested energy while running. Fig. 22a shows the harvested power while running. We then emulate this data in the lab to study the behavior of our heuristics. Since this harvester generates energy with body movements, we consider AR (implemented on Flicker) as the most suitable application to run our tests on.

Fig. 22b shows that the total executions with REHASH are a bit higher than InK but at the loss of a large number of sensor samples. We notice that the *on-time* heuristic is not able to efficiently capture the trend of harvested energy. This is because a coil harvester generates energy in the form of bursts at around 4Hz (frequency varies with running speed). These burst slowly charge the capacitor until it reaches a level where Flicker can start running AR application. The capacitor on Flicker depletes in a few milliseconds, while the energy bursts come once every 240 to 260 milliseconds (Fig. 22a). This burst is too slow to renew the capacitor charge during execution, and therefore will not change the active time of the device. This may keep the on-time constant across many power failures, resulting in no adaptation. On the other hand, the *off-time* heuristic performs better since it works during inactive period and does not have anything to do with energy bursts. It efficiently keeps the record of inactive period and trigger adaptation when needed.

We therefore conclude that *on-time* works best for harvesters that provide continuous power and *off-time* may work best for all kind of harvesters, such as those that are impulse or burst based. The behavior of *task-count* is somewhat random in applications that perform I/O operations, so it can not be easily generalized. While these conclusions need more investigation with other harvesters, our purpose is here is to show that REHASH framework can work with multiple energy harvesters. And in some cases, if proposed heuristics cannot capture the changes in energy, REHASH is flexible enough to let developers integrate their own software or hardware signals and heuristics.

Table 3. Memory consumption (in B) for applications in InK and REHASH.

	InK		REHASH	
	.text	*.data*	*.text*	*.data*
AR	7882	5662	14216	5708
GHM	5658	5262	11624	5308
MNIST	11412	37272	14010	37318

Table 4. Initialization overhead (in ms) for applications in InK and REHASH.

	InK	REHASH		
	non-adaptive	*on-time*	*off-time*	*task-count*
AR	1.61	4.21	4.07	4.19
GHM	1.65	4.24	4.08	4.19
MNIST	1.64	4.24	4.10	4.22

Table 5. Minimum capacitor size (in µF) required for InK and REHASH.

	InK	REHASH		
	non-adaptive	*on-time*	*off-time*	*task-count*
AR	1.07	2.70	2.62	2.69
GHM	1.05	2.72	2.64	2.71
MNIST	1.06	2.72	2.62	2.69

Table 6. Lines of code (LOC) for applications in InK and REHASH.

	InK	REHASH		
			adaptUp()	*adaptDown()*
AR	1664	1727(+3.8%)	15	15
GHM	1273	1343(+6.8%)	19	19
MNIST	677	723(+5.5%)	8	8

7.8 Microbenchmarks and User Experience

Memory Overhead. Shown in Table 3. The memory overhead of making applications adaptive is small, as most of the logic is contained within the kernel.

Initialization Overhead. Shown in Table 4. The initialization overhead of REHASH's adaptive applications is 2.5× of non-adaptive InK. Our empirical observation shows that the initialization overhead of this magnitude is tolerable and does not affect intermittently powered applications badly.

Capacitor Size. Since the initialization overhead of REHASH is higher than InK, it needs a bigger minimum-size capacitor to store the energy required to execute the overhead code. We use the time overhead (t_o) information from Table 4, calculate corresponding energy overhead (E_o), and then use upper and lower threshold operating voltages (V_{on} and V_{off}) of Flicker to find the minimum size of the capacitor using equations below.

$$\frac{1}{2}CV_{on}^2 - \frac{1}{2}CV_{off}^2 = E_o$$

As,

$$E_o = IV_{cc}t_o$$

Equating above two equations

$$C = 2\frac{IV_{cc}}{(V_{on}^2 - CV_{off}^2)}t_o \tag{3}$$

Table 5 shows the minimum capacitor size of REHASH which is almost same for every heuristic. Typically, intermittent computing platforms use a capacitor of value greater than tens or hundreds of microfarads. So a 2.7µF capacitor requirement for REHASH is almost negligible. Other than this, there are no known components of REHASH that can be affected by capacitor size; only the largest task in an application determines the minimum capacitor size.

Energy Overhead. We calculated the energy harvested by non-adaptive and adaptive application variants from the supply voltage and current draw, and found negligible differences, meaning adaptation was not a burden over typical non-adaptive runtimes. Further, the higher app completion rates and sensor coverage suggest adaptation used the energy more wisely.

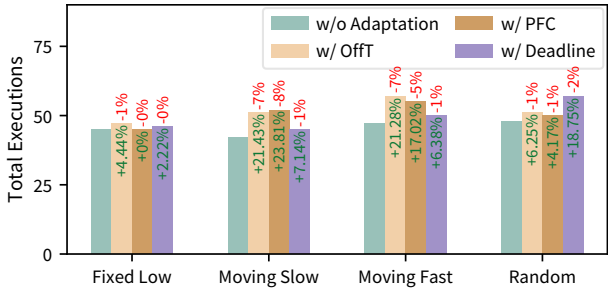


Fig. 23. REHASH-explorer with and without adaptation for a run of MNIST application on different IV surfaces. Adaptive runs result in more app completions. Different configurations of REHASH-explorer show different trends, narrowing the design space, that developer can use to write heuristics for the hardware.

Running REHASH-explorer We look at the trend of predictions made by REHASH-explorer. Absolute accuracy (i.e., the exact number of joules of energy spent or app completions compared to the hardware) is not a useful metric for REHASH-explorer. The reason is that our simulation model does not capture the minute hardware differences between platforms. It also does not account for large amounts of bypass capacitance and diverse energy management circuitry found on some platforms in intermittent computing. We posit that the trend or behavior with different heuristics/configurations is helpful for developers. They want to see, when history size or ϵ value is changed on the interface, or a different step size is selected, is the proportional change reflected or not. To conduct this evaluation we look at multiple configurations of our MNIST application and note the trend in the statistics reported by REHASH-explorer. The results are shown in Fig. 23. As shown, the predicted app completions of REHASH-explorer follow the same trend when compared to the actual tests run on the hardware (see Fig. 13a and Fig. 18)

User Experience. While we did not conduct a formal user study, REHASH only adds a few additional semantics to enable heuristic adaptation. Since the runtime hides the actual heuristic function, and the gathering of adaptation signals, the programmer need only to concern themselves with making tasks able to handle varying sizes of samples. In total, the lines of code added beyond a vanilla InK application to make them adaptive using REHASH, are as shown in Table 6. Almost half of the lines added are for the `__ADAPT_UP` and `__ADAPT_DOWN` routines that control the adaptive variables. The rest of the lines are distributed across the tasks that user wants to adapt. This represents a small amount of effort compared to the development of the actual application. But puts power in the hands of the developers to ensure adaptation is done right, instead of relying on invisible degradation policies as in other runtime systems like CatNap, Celebi, and COALA.

7.9 Summary of Findings

As shown, heuristic adaptation with REHASH provides a structured way to make applications energy-aware. Using our implementation, the three diverse applications we built have higher throughput (more app completions / executions / classifications) and higher sensor coverage, at the cost of nominal drops in accuracy or performance. The heuristics we develop closely track the maximum energy in an environment, providing a good estimation of scarcity/abundance. REHASH is low overhead, and only requires a small amount of code to make non-adaptive tasks adaptive.

8 RELATED WORK

The system proposed represent a general framework for energy-aware adaptation for energy harvesting devices; instantiated for intermittently powered battery-free devices. Much of the related work was discussed in Section 2, of note however, are other adaptive runtime systems and real time schedulers, and other tools like REHASH-`explorer` that give insight into program operation pre-deployment.

8.1 Energy Management

Prior work on energy management for battery-powered energy-harvesting sensing systems like LT-ENO [10], and ENO-MAX [77] offer strategies for energy utilization in order to avoid battery depletion or waste of energy, but they do not react to the changing energy conditions. Eon [75], maximizes the quality of service under available energy constraints by choosing the program flow that is marked by the programmer for the given energy state. PreAct [24] offers better energy management for these systems via adapting the duty cycle of the application considering the variations of the energy in the long-term from weeks to months.

8.2 Scheduling and Adaptation in Intermittent Computing

Adaptation in general computing systems has been explored at length under many different names. The ideas of "imprecise computing" for real time systems [52] discussed trading accuracy for performance. Similarly the idea of "Approximate computing" has been heavily explored in high performance systems for the last two decades [64]. Intermittent computing builds on these ideas, but takes the cost of failure to the extreme, and represents the most constrained computing systems in terms of memory and power. More recent intermittent computing runtimes have grappled with adaptation in application specific ways, or looked at a single mechanism of adaptation.

Dewdrop [11] adapts the task execution to best match task energy requirements to available energy, by scheduling when to start the next task iteration. Coala [59] aims to reduce the overhead of saving tasks' state in non-volatile memory, by grouping tasks adaptively based on the estimated energy availability and postponing their commit operations. Both Dewdrop and Coala do not play with the energy demands (sizes) of the tasks. This makes their response to changing energy dynamics limited since they cannot degrade the quality of computation. Camaroptera [67] demonstrates the execution of different versions of the same application, by selecting the best match based on the charging rate of the device. However, the granularity of the adaptation is limited by the versions provided by the programmer. Similar to Dewdrop and Coala, Camaroptera cannot fine-tune the sizes of the tasks in these versions. Quite similar to the objectives of Dewdrop, Celebi [41] studies the real-time scheduling problem for intermittent systems by taking into account the time, energy demands of the fixed-size tasks and harvested energy in order to increase the number of jobs that meet the deadline. Similar to Celebi, CatNap [58] focuses on meeting the schedules of the time critical events on intermittent systems. On the contrary to Celebi, CatNap degrades the application quality in a rigid manner upon a decrease in the environmental energy in order to meet the deadlines of the time critical events. The main design principle is to separate the energy required to service the events from the energy that will be used for the tasks. Inference specific runtimes, like ePerceptive [65] and Zygarde [40] use off time and hardware voltage measurements, respectively, as a trigger for exiting early from the execution of an inference task. Zygarde attempts to schedule tasks, in a mechanism similar to Dewdrop, so that failures are less likely.

For all scheduling and energy management approaches, the mechanisms of adaptation, the when and how, are either rigid or unexplored. REHASH for the first time formalizes a general purpose and flexible framework for adaptation in intermittent computing. REHASH is a super set of these previous adaptive approaches, pulling together the various implicit signals, heuristics, and knobs used by other work, and it in a principled approach to adaptation for programmers.

8.3 Programmer Tools for Energy Introspection

Power and energy consumption of mobile computing systems has long been an important benchmark. Industry tools for understanding energy consumption like Apple's Instruments and Texas Instrument's EnergyTrace++ enable a coarse look at a specific application. Tools developed by the wireless sensor networks community like like Power TOSSIM [71], and Cooja/MSPSIM [20] provide instruction level insight into power, without the need for actual hardware. More recently, energy harvesting based simulation platforms have arisen, including SIREN [22], GreenCastalia [7] , and Fused [74]. These simulate the operation of the microcontroller under dynamic energy harvesting conditions at a low level, including power failures. These tools could assist in understanding adaptation, but are too low level (often simulating at the instruction level) to provide fast feedback on the performance of a particular adaptation strategy.

Other tools, more closely related to REHASH-`explorer`, are focused at design time or deployment time. EPIC is a compile-time energy analysis tool which takes the assembly instructions for transiently powered computers [2]. EPIC can accurately predict the energy of an arbitrary intermittent program considering variations in power consumption and clock speed. EPIC is complementary to REHASH-`explorer`, and could be integrated for more accurate power projections from selected REHASH-`explorer` configurations. PowerForecaster [63] help users make decisions about energy efficiency at install time to understand battery lifetime. PADA [62] gives an enriched simulation for developers on their power draw for phone based apps. The Amulet ARP-view [26] provides a parametric simulation at compile time that derives battery lifetime impact of programmer code changes. These tools are not focused on intermittent computing systems, and do not have a notion of energy harvesting, or adaptation. REHASH-`explorer` is unique in its targeted application domain, and it's ability to explore different configurations of an adaptive intermittent application.

9 DISCUSSION AND FUTURE WORK

This paper began with the idea that adaptation at the application (task) level could be beneficial to sensing programs embedded for ultra long periods in a changing environment. The devices that best exemplify and motivate this type of ultra long deployments do not have batteries, so we focused on their constraints. Other methods of adaptation, for example purely focused on forward progress by checkpointing less in non-task based languages, do not consider the application level constraints and information that could be applied to bring more performant and data rich applications to the space. This section discusses the limitations of the study and system, as well as potential future work.

9.1 Limitations

Heuristic adaptation is only one approach to adaptation, that focuses on the execution statistics to guess about energy availability in the future. Heuristics are rules of thumb that can be confounded in some instances. For example, extremely random energy harvesting environments may disadvantage approaches that try to adapt, as the history and statistics will be much more noisy. Other situations where the energy environment is just too low to support any operation, or the energy is so high that adaptation is unnecessary, may also disadvantage heuristic adaptation methods. This work attempts to generalize across a range of energy environments and a few applications, but does not cover all confounding situations.

Comprehensiveness: We did not investigate other types of adaptation signals like number of uncompleted tasks, or the relative energy cost / size of a task. Profiling energy costs of tasks would be difficult, but could lead to fruitful and more precise adaptation decisions. We also left unstudied more complex heuristics that integrate more than just the average of past history. EWMA based adaptation or other more complex statistical estimation techniques could be used, potentially at much higher compute cost. Additionally, the output of heuristic functions could be increased from 1-bit, to provide a more continuous output which would allow more fine-grained adaptation. That

said, due to the flexibility of REHASH and REHASH-`explorer`, a developer could create many of these heuristics themselves.

Heuristics vs. Machine Learning: Many of the knobs and signals could be imagined as a machine learning process; where features are constructed from these adaptation signals and proper weights per feature are given using a neural network. In this case, we prefer a heuristic to reduce training per-application, which may be difficult or impossible for intermittent nodes. Moreover, we appreciate the simplicity and explainability of a heuristic; simplicity enables quick decisions with limited resources, while explainability may enable programmers to choose a heuristic more easily for their application, instead of training a neural network.

Simulation Limitations: REHASH-`explorer` is limited by the user inputs it is supplied, and if they are accurate (i.e. energy of tasks). We found that the absolute accuracy of REHASH-`explorer` compared to real hardware was not exact, however, the behavior of change matched well. REHASH-`explorer` is not meant to model perfectly the hardware, but is instead meant for comparison of configurations within an application. This is the role of most simulation platforms that handle a large and diverse set of hardware possibilities. We happily trade off absolute accuracy (not as useful for the developer anyway for design) for speed and lower user burden to explore many configurations of adaptation in a short time.

Continuing Efforts and Community: The REHASH framework provides a scaffold for adaptive computing that is complemented by REHASH-`explorer` so that developers can take advantage of the flexibility of the framework, but understand the cost/benefit of their design decisions. REHASH-`explorer` is a research tool still under development, and is not fully integrated and automated (for example, energy profiling is not yet automatic). Likely other aspects of the tool and usage of the framework could be more seamless, and we expect that by releasing this to the community free and open source, we can partner with others to improve the integration and friendliness of the platform, and enable future research.

9.2 Future Work

A number of follow-on ideas come from the exploration and implementation of heuristic adaptation presented in this paper.

Capacitor Voltage as Signal: Another potential signal for adaptation could be the voltage on the capacitor. Some previous work has used this signal as the time when a checkpoint of the entire volatile RAM should be stored to NV memory [5]. This signal is useful for just-in-time checkpointing, but further study would be required to ascertain usefulness as an adaptation signal, since the capacitor voltage is highly variable and a high voltage does not mean a high energy trace (because of IV curve behavior of energy harvesters). Voltage is not a great signal because the amount of power being harvested depends on the behavior of the load.

Language Constructs: We extended the programming model of the state of the art task based programming language for intermittent computing, InK, however much more intricate constructs could be imagined. For example, our measurement study explored how heuristic adaptation could enable choosing different paths in the task graph, not just based on priority or events, as in InK, but based on the heuristic estimate of the energy availability of the environment. Implementing language constructs to support this and more are interesting directions for future work.

Energy-Neutral Systems and Generality: We focused in this paper on exploring heuristic adaptation for tiny, batteryless, energy harvesting systems—which can lose power many times a second, and have volatile power supplies and energy harvesting. However, the tools and techniques could potentially be applied to other larger classes of sensing systems that deal with less frequent, or longer- term interruptions, such as energy-neutral sensing systems [61]. These sensors have more computation power, larger energy harvesters, and rarely

experience power failures. However, the task completion and on-time statistics could be used to make longer term adaptation decisions to account for seasonal changes in energy availability. Mapping new heuristics to new device classes would be the first step towards a general system; for example a heuristic signal from speed of capacitor depletion / emergency reserve access on Permamote [42], or the metered energy harvested with SignPost [1].

Profiling Tasks. If we had perfect, oracle-like knowledge of the task energy costs at each level of adaptation, and the knowledge of future energy harvesting, we could leverage algorithms like LT-ENO [10], and ENO-MAX [77] to provide an optimal quality of service of the application with respect to the energy availability. As mentioned previously, it is extremely hard to predict future energy due to the diversity of energy harvesting environments. Even if it might be possible to have a coarse-grained prediction of for some deployments, the energy profiling of tasks is also extremely hard. Pre-computing the task energy costs is not feasible due to the start up and tear down costs, peripheral usage, randomization and input dynamics. There might be non-deterministic tasks whose energy consumption might vary from execution to execution. [11]. Moreover, the objective of adaptation is to scale tasks with respect to the energy availability by degrading their quality. This requires profiling each scale of the task [13], where the number of states we have to measure per each task scale can be intractable.

10 CONCLUSIONS

Intermittently powered, energy harvesting computers and sensing devices have great potential for revolutionizing sensor networks and the Internet-of-Things. This paper has explored the challenges of developing robust energy aware adaptation for batteryless sensors. By adapting to the energy environment via down-sampling, task-skipping, pixel/class-skipping (explored in this paper) more consistent and reliable data delivery can be realized. We aim to give developers flexibility to account for the broad range of applications, hardware, and energy harvesting scenarios that could be encountered. However, flexibility is a double edged sword, as developers suffer from analysis paralysis and decision fatigue. So our contribution is two part. First, we conceptualized the REHASH framework for energy-aware, heuristic adaptation, and developed an instantiation for intermittent computing. This general framework gives developers complete freedom to build their own adaptation strategy for task based programs. We made heuristic functions from easily gathered signals and explored heuristic adaptation over three applications in multiple energy harvesting scenarios. Second, we develop a formalization and online simulation tool that helps users understand the tradeoffs between different adaptation strategies and configurations of their application. This developer focused approach reduces the cognitive decision burden. We found that heuristic adaptation can be highly useful for sensing and machine learning applications. Adaptive MNIST performs up to 46% more classifications with only a 5% drop in performance or accuracy for MNIST application. Adaptive GHM gives up to 154% more useful data by keeping the number of samples and packets small in low energy mode, but maintaining the same number of samples in a time window compared to non-adaptive. Adaptive AR had 76% more useful data compared to non-adaptive AR, with reduction of 30% samples per execution on average, but takes overall more samples in a time window. Heuristic adaptation, with toolchain support seems a promising route towards large scale, zero-maintenance, long term deployments of useful sensing devices.

ACKNOWLEDGMENTS

This research is based upon work supported by the National Science Foundation under award numbers CNS-1850496, CNS-2032408, CNS-2038853, CNS-2030251, and CNS-2107400. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation

REFERENCES

[1] Joshua Adkins, Branden Ghena, Neal Jackson, Pat Pannuto, Samuel Rohrer, Bradford Campbell, and Prabal Dutta. 2018. The signpost platform for city-scale sensing. In *2018 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 188–199.

[2] Saad Ahmed, Abu Bakar, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2019. The Betrayal of Constant Power × Time: Finding the Missing Joules of Transiently-Powered Computers. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2019)*. Association for Computing Machinery, New York, NY, USA, 97–109. https://doi.org/10.1145/3316482.3326348

[3] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2019. Efficient Intermittent Computing with Differential Checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2019)*. Association for Computing Machinery, New York, NY, USA, 70–81. https://doi.org/10.1145/3316482.3326357

[4] Abu Bakar and Josiah Hester. 2018. Making sense of intermittent energy harvesting. In *Proceedings of the 6th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems*. ACM, 32–37.

[5] Domenico Balsamo, Alex S Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M Al-Hashimi, Geoff V Merrett, and Luca Benini. 2016. Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 12 (2016), 1968–1980.

[6] 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. *IEEE Embedded Systems Letters* 7, 1 (2015), 15–18. http://ieeexplore.ieee.org/abstract/document/6960060/

[7] David Benedetti, Chiara Petrioli, and Dora Spenza. 2013. GreenCastalia: An energy-harvesting-enabled framework for the Castalia simulator. In *Proceedings of the 1st International Workshop on Energy Neutral Sensing Systems*. 1–6.

[8] Naveed Anwar Bhatti and Luca Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN '17)*. ACM, New York, NY, USA, 209–219. https://doi.org/10.1145/3055031.3055082

[9] Naveed Anwar Bhatti and Luca Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN '17)*. ACM, New York, NY, USA, 209–219. https://doi.org/10.1145/3055031.3055082

[10] Bernhard Buchli, Felix Sutton, Jan Beutel, and Lothar Thiele. 2014. Dynamic power management for long-term energy neutral operation of solar energy harvesting systems. In *Proceedings of the 12th ACM conference on embedded network sensor systems*. 31–45.

[11] M. Buettner, B. Greenstein, and D. Wetherall. 2011. Dewdrop: An Energy-Aware Runtime for Computational RFID. In *Proc. 8th USENIX Conf. Networked Systems Design and Implementation (NSDI'11)*. ACM, Boston, MA, USA, 197–210.

[12] 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 (OOPSLA 2016)*. ACM, New York, NY, USA, 514–530. https://doi.org/10.1145/2983990.2983995

[13] Alexei Colin and Brandon Lucia. 2018. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction*. 116–127.

[14] 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*. ACM, 767–781.

[15] 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

[16] Powercast Corp. [n.d.]. Powercast Hardware. http://www.powercastco.com.. Accessed: 2019-04-11.

[17] Jasper de Winkel, Carlo Delle Donne, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. 2020. Reliable Timekeeping for Intermittent Computing. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 53–67. https://doi.org/10.1145/3373376.3378464

[18] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. 2020. Battery-Free Game Boy. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 3, Article 111 (Sept. 2020), 34 pages. https://doi.org/10.1145/3411839

[19] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *29th annual IEEE international conference on local computer networks*. IEEE, 455–462.

[20] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón. 2009. COOJA/MSPSim: interoperability testing for wireless sensor networks. In *Proceedings of the 2nd International Conference on*

Simulation Tools and Techniques. 1–7.
[21] Nicola Femia, Giovanni Petrone, Giovanni Spagnuolo, and Massimo Vitelli. 2005. Optimization of perturb and observe maximum power point tracking method. *IEEE transactions on power electronics* 20, 4 (2005), 963–973.
[22] Matthew Furlong, Josiah Hester, Kevin Storer, and Jacob Sorber. 2016. Realistic Simulation for Tiny Batteryless Sensors. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys'16)*. ACM, New York, NY, USA, 23–26. https://doi.org/10.1145/2996884.2996889
[23] Kai Geissdoerfer, Mikołaj Chwalisz, and Marco Zimmerling. 2019. Shepherd: a portable testbed for the batteryless IoT. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*. 83–95.
[24] Kai Geissdoerfer, Raja Jurdak, Brano Kusy, and Marco Zimmerling. 2019. Getting more out of energy-harvesting systems: Energy management under time-varying utility with preact. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*. 109–120.
[25] Graham Gobieski, Nathan Beckmann, and Brandon Lucia. 2018. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. *arXiv preprint arXiv:1810.07751* (2018).
[26] Josiah Hester, Travis Peters, Tianlong Yun, Ronald Peterson, Joseph Skinner, Bhargav Golla, Kevin Storer, Steven Hearndon, Kevin Freeman, Sarah Lord, Ryan Halter, David Kotz, and Jacob Sorber. 2016. Amulet: An Energy-Efficient, Multi-Application Wearable Platform. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems (SenSys '16)*. ACM, New York, NY, USA, 216–229. https://doi.org/10.1145/2994551.2994554
[27] Josiah Hester, Timothy Scott, and Jacob Sorber. 2014. Ekho: Realistic and Repeatable Experimentation for Tiny Energy-harvesting Sensors. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems (SenSys '14)*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/2668332.2668336
[28] Josiah Hester, Timothy Scott, and Jacob 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.
[29] Josiah Hester, Lanny Sitanayah, Timothy Scott, and Jacob Sorber. 2017. Realistic and Repeatable Emulation of Energy Harvesting Environments. *ACM Trans. Sen. Netw.* 13, 2, Article 16 (April 2017), 33 pages. https://doi.org/10.1145/3064839
[30] Josiah Hester, Lanny Sitanayah, and Jacob Sorber. 2015. Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys '15)*. ACM, New York, NY, USA, 5–16. https://doi.org/10.1145/2809695.2809707
[31] 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 (SenSys '17)*. ACM, New York, NY, USA. http://josiahhester.com/cv/files/flickersensys.pdf
[32] Josiah Hester and Jacob Sorber. 2017. Flicker: Rapid Prototyping for the Batteryless Internet-of-Things. In *Proceedings of the 15th ACM Conference on Embedded Networked Sensor Systems (SenSys '17)*. To appear.
[33] Josiah Hester and Jacob Sorber. 2017. New Directions: The Future of Sensing is Batteryless, Intermittent, and Awesome. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17)*. ACM, New York, NY, USA. http://josiahhester.com/cv/files/newdirssensys2017.pdf
[34] Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17)*. ACM, New York, NY, USA. http://josiahhester.com/cv/files/mayflysensys2017.pdf
[35] Josiah Hester, Nicole Tobias, Amir Rahmati, Lanny Sitanayah, Daniel Holcomb, Kevin Fu, Wayne P Burleson, and Jacob Sorber. 2016. Persistent Clocks for Batteryless Sensing Devices. *ACM Transactions on Embedded Computing Systems (TECS)* 15, 4 (2016).
[36] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 228–240. https://doi.org/10.1145/3079856.3080238
[37] Texas Instruments. [n.d.]. BQ25570-Ultra Low power Harvester power Management IC with boost charger, and Nanopower Buck Converter. https://www.ti.com/product/BQ25570. Accessed: 2020-11-14.
[38] Texas Instruments. [n.d.]. MSP430FR5994 Launchpad. https://www.ti.com/tool/MSP-EXP430FR5994. Accessed: 2020-12-11.
[39] Texas Instruments. [n.d.]. MSP430FRxx FRAM Microcontrollers. http://www.ti.com/lsds/ti/microcontrollers_16-bit_32-bit/msp/ultra-low_power/msp430frxx_fram/overview.page. Accessed: 2019-03-09.
[40] Bashima Islam, Yubo Luo, and Shahriar Nirjon. 2019. Zygarde: Time-sensitive on-device deep intelligence on intermittently-powered systems. *arXiv preprint arXiv:1905.03854* (2019).
[41] Bashima Islam and Shahriar Nirjon. 2020. Scheduling Computational and Energy Harvesting Tasks in Deadline-Aware Intermittent Systems. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 95–109.
[42] Neal Jackson, Joshua Adkins, and Prabal Dutta. 2019. Capacity over capacitance for reliable energy harvesting sensors. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*. 193–204.
[43] Hrishikesh 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.

[44] J. M. Kahn, R. H. Katz, and K. S. J. Pister. 1999. Next Century Challenges: Mobile Networking for &Ldquo;Smart Dust&Rdquo;. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '99)*. ACM, New York, NY, USA, 271–278. https://doi.org/10.1145/313451.313558

[45] Vito Kortbeek, Abu Bakar, Stefany Cruz, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. 2020. Bfree: Enabling battery-free sensor prototyping with python. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4, 4 (2020), 1–39.

[46] 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 (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 85–99. https://doi.org/10.1145/3373376.3378476

[47] Yann LeCun. [n.d.]. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* ([n. d.]).

[48] Yann LeCun, Corinna Cortes, and CJ Burges. 2010. MNIST handwritten digit database. (2010).

[49] E. A. Lee, B. Hartmann, J. Kubiatowicz, T. Simunic Rosing, J. Wawrzynek, D. Wessel, J. Rabaey, K. Pister, A. Sangiovanni-Vincentelli, S. A. Seshia, D. Blaauw, P. Dutta, K. Fu, C. Guestrin, B. Taskar, R. Jafari, D. Jones, V. Kumar, R. Mangharam, G. J. Pappas, R. M. Murray, and A. Rowe. 2014. The Swarm at the Edge of the Cloud. *IEEE Design Test* 31, 3 (June 2014), 8–20. https://doi.org/10.1109/MDAT.2014.2314600

[50] Seulki Lee, Bashima Islam, Yubo Luo, and Shahriar Nirjon. 2019. Intermittent learning: On-device machine learning on intermittently powered system. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 3, 4 (2019), 1–30.

[51] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. 2005. TinyOS: An operating system for sensor networks. In *Ambient intelligence*. Springer, 115–148.

[52] Jane WS Liu, Wei-Kuan Shih, Kwei-Jay Lin, Riccardo Bettati, and Jen-Yao Chung. 1994. Imprecise computations. *Proc. IEEE* 82, 1 (1994), 83–94.

[53] ANYSOLAR Ltd. [n.d.]. IXOLAR High Efficiency Solar MD. https://tinyurl.com/y3pf3d85. Accessed: 2020-04-11.

[54] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent Computing: Challenges and Opportunities. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 8:1–8:14. https://doi.org/10.4230/LIPIcs.SNAPL.2017.8

[55] 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 '15)*. ACM, New York, NY, USA, 575–585. https://doi.org/10.1145/2737924.2737978

[56] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 96.

[57] Kiwan Maeng and Brandon Lucia. 2018. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 129–144.

[58] Kiwan Maeng and Brandon Lucia. 2020. Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1005–1021.

[59] 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.

[60] Marcelo Martins, Justin Cappos, and Rodrigo Fonseca. 2015. Selectively taming background android apps to improve battery lifetime. In *2015 USENIX Annual Technical Conference*. 563–575.

[61] G. V. Merrett and B. M. Al-Hashimi. 2017. Energy-driven computing: Rethinking the design of energy harvesting systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 960–965. https://doi.org/10.23919/DATE.2017.7927130

[62] Chulhong Min, Seungchul Lee, Changhun Lee, Youngki Lee, Seungwoo Kang, Seungpyo Choi, Wonjung Kim, and Junehwa Song. 2016. PADA: Power-Aware Development Assistant for Mobile Sensing Applications. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '16)*. Association for Computing Machinery, New York, NY, USA, 946–957. https://doi.org/10.1145/2971648.2971676

[63] Chulhong Min, Youngki Lee, Chungkuk Yoo, Seungwoo Kang, Sangwon Choi, Pillsoon Park, Inseok Hwang, Younghyun Ju, Seungpyo Choi, and Junehwa Song. 2015. PowerForecaster: Predicting smartphone power impact of continuous sensing applications at pre-installation time. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. 31–44.

[64] Sparsh Mittal. 2016. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 1–33.

[65] Alessandro Montanari, Manuja Sharma, Dainius Jenkus, Mohammed Alloulah, Lorena Qendro, and Fahim Kawsar. 2020. ePerceptive: energy reactive embedded intelligence for batteryless sensors. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*. 382–394.

[66] Saman Naderiparizi, Pengyu Zhang, Matthai Philipose, Bodhi Priyantha, Jie Liu, and Deepak Ganesan. 2017. Glimpse: A programmable early-discard camera architecture for continuous mobile vision. In *Proceedings of the 15th Annual International Conference on Mobile*

Systems, Applications, and Services. 292–305.
[67] Matteo Nardello, Harsh Desai, Davide Brunelli, and Brandon Lucia. 2019. Camaroptera: A batteryless long-range remote visual sensing system. In *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems*. 8–14.
[68] Alok Prakash, Hussam Amrouch, Muhammad Shafique, Tulika Mitra, and Jörg Henkel. 2016. Improving mobile gaming performance through cooperative CPU-GPU thermal management. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.
[69] Ben Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-Running Computation on RFID-Scale Devices. In *Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. ACM, Newport Beach, CA, USA, 159–170.
[70] 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.
[71] Victor Shnayder, Mark Hempstead, Bor-rong Chen, Geoff Werner Allen, and Matt Welsh. 2004. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM, 188–200.
[72] Sivert T Sliper, Oktay Cetinkaya, Alex S Weddell, Bashir Al-Hashimi, and Geoff V Merrett. 2020. Energy-driven computing. *Philosophical Transactions of the Royal Society A* 378, 2164 (2020), 20190158.
[73] Sivert T. Sliper, Oktay Cetinkaya, Alex S. Weddell, Bashir Al-Hashimi, and Geoff V. Merrett. 2020. Energy-driven computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378, 2164 (2020), 20190158. https://doi.org/10.1098/rsta.2019.0158 arXiv:https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2019.0158
[74] Sivert T Sliper, William Wang, Nikos Nikoleris, Alex S Weddell, and Geoff V Merrett. 2020. Fused: closed-loop performance and energy simulation of embedded systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 263–272.
[75] 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 *Proceedings of the 5th international conference on Embedded networked sensor systems*. 161–174.
[76] J-M Tarascon. 2010. Key challenges in future Li-battery research. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 368, 1923 (2010), 3227–3241.
[77] Christopher M Vigorito, Deepak Ganesan, and Andrew G Barto. 2007. Adaptive control of duty cycling in energy-harvesting wireless sensor networks. In *2007 4th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*. IEEE, 21–30.
[78] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation without Hardware Support or Programmer Intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 17–32. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/vanderwoude
[79] Wu Yawen, Wang Zhepeng, Jia Zhenge, Shi Yiyu, and Hu Jingtong. 2020. Intermittent Inference with Nonuniformly Compressed Multi-Exit Neural Network for Energy Harvesting Powered Devices. *arXiv preprint arXiv:2004.11293* (2020).
[80] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. 2018. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. ACM, 41–53.
[81] Hong Zhang, Jeremy Gummeson, Benjamin Ransford, and Kevin Fu. 2011. Moo: A batteryless computational RFID and sensing platform. *Department of Computer Science, University of Massachusetts Amherst., Tech. Rep* (2011).