InK: Reactive Kernel for Tiny Batteryless Sensors

Kasım Sinan Yıldırım*
Ege University
Izmir, Turkey
sinan.yildirim@ege.edu.tr

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

Dimitris Patoukas†
Delft University of Technology
Delft, The Netherlands
d.patoukas@student.tudelft.nl

Koen Schaper‡
Delft University of Technology
Delft, The Netherlands
k.p.schaper@student.tudelft.nl

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

Josiah Hester
Northwestern University
Evanston, IL, USA
josiah@northwestern.edu

ABSTRACT

Tiny energy harvesting battery-free devices promise maintenance free operation for decades, providing swarm scale intelligence in applications from healthcare to building monitoring. These devices operate intermittently because of unpredictable, dynamic energy harvesting environments, failing when energy is scarce. Despite this dynamic operation, current programming models are static; they ignore the event-driven and time-sensitive nature of sensing applications, focusing only on preserving forward progress while maintaining performance. This paper proposes InK; the first reactive kernel that provides a novel way to program these tiny energy harvesting devices that focuses on their main application of event-driven sensing. InK brings an event-driven paradigm shift for batteryless applications, introducing building blocks and abstractions that enable reacting to changes in available energy and variations in sensing data, alongside task scheduling, while maintaining a consistent memory and sense of time. We implemented several event-driven applications for InK, conducted a user study, and benchmarked InK against the state-of-the-art; InK provides up to 14 times more responsiveness and was easier to use. We show that InK enables never before seen batteryless applications, and facilitates more sophisticated batteryless programs.

CCS CONCEPTS

• **Computer systems organization → Embedded software**; • **Hardware → Analysis and design of emerging devices and systems**; • **Software and its engineering → Embedded software**;

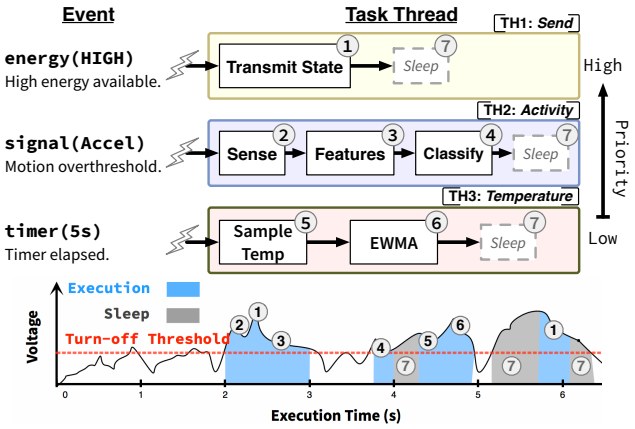

Figure 1: An InK sensing application that measures and sends data depending on available energy, motion triggers, and the output of a power failure-resistant timekeeper. The simulated task execution trace is shown. InK is the first event-driven runtime for batteryless, energy harvesting sensor networks. InK fairly schedules concurrent task threads that span power failures and respond to events—like high energy availability, hardware interrupts, and elapsed time.

KEYWORDS

Kernel, Reactive, Batteryless, Intermittent, Energy Harvesting

ACM Reference Format:
Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemysław Pawełczak, and Josiah Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *The 16th ACM Conference on Embedded Networked Sensor Systems (SenSys '18), November 4–7, 2018, Shenzhen, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3274783.3274837

1 INTRODUCTION

Advances in computational power, decreases in device size, and progress in communication and energy harvesting circuits are enabling stand-alone and sustainable applications for the Internet of Things (IoT) [38]. Soon trillions of tiny sensors collecting, processing, and communicating data will fundamentally change how we view healthcare [10], water infrastructure [48], energy-efficient buildings [4, 15], and our interactions with the natural world [37]. To make this vision feasible energy harvesting must be leveraged

*K. S. Yıldırım is also affiliated with Delft University of Technology, The Netherlands.
†D. Patoukas can be also contacted via patoukas@gmail.com.
‡K. Schaper can be also contacted via kpschaper@gmail.com.

SenSys '18, November 4–7, 2018, Shenzhen, China

ACM ISBN 978-1-4503-5952-8/18/11…$15.00
https://doi.org/10.1145/3274783.3274837

and batteries must be left behind, allowing near permanent sensing at low cost and size with reduced ecological impact.

Batteryless sensing devices like Flicker [20] or WISP [45] harvest energy opportunistically from the environment (via solar cells, RF scavenging, thermal or kinetic conversion) and store the energy in small capacitors. Computation, sensing, and communication tasks proceed when enough energy is available to turn on the processor, at which point execution continues until energy is exhausted and the device abruptly fails: leading to an *intermittent execution*. These often frequent (up-to 10 times per second [39]) power failures reset the processor's *volatile state*, zeroing memory and register contents, clock cycle counts, and rendering timestamps and timers useless. Therefore, programs must store volatile state to *non-volatile memory* (FRAM [50]) before power failures.

While the most successful wireless sensor network operating systems, e.g. TinyOS [31] and Contiki [17] are *event-driven* and *dynamic*, recent work in intermittent computing is neither. Current approaches preserve forward progress using checkpoints automatically inserted at compile time [5, 7, 28, 39], or require program refactoring into task based models [12, 22, 34]. These programming models are static; they cannot respond or adapt to a changing environment or hardware interrupts, and polling-based; meaning they waste energy actively looking for changes, instead of passively waiting for an interrupt. Sensor networks are inherently event-driven: responding to changes in the environment, timer and hardware interrupts, and communications to determine the next task to complete, or the data to collect. Besides, modifying existing event-driven kernels, e.g. TinyOS and Contiki, or programming models, by inserting checkpoints to handle intermittent execution is by no means useful. Power failures, in particular during interrupt handling, can leave non-volatile state partially updated, leading volatile state to be inconsistent with non-volatile state. These inconsistencies cause intermittent execution to deviate from continuously-powered behavior, potentially leading to unrecoverable application failures [32].

In this paper we introduce the *Intermittently-Powered Kernel (InK)*, the first *reactive* task-based run-time system for batteryless, energy harvesting sensors. InK eschews the static task execution model, and instead enables energy-adaptive, event-driven, and time sensitive applications for batteryless sensing devices. Compared to existing kernels for embedded systems, InK exhibits new properties dedicated to batteryless systems. In particular, InK (i) ensures forward progress of computation by executing restartable atomic tasks encapsulated by *task threads* each with unique priority; (ii) ensures time constraints of task threads by employing preemptive and power failure-immune scheduling and building a timer subsystem composed of *persistent timers*; (iii) ensures memory consistency during event handling, as interrupt handlers are not inherently atomic and power failures during their execution might lead to memory inconsistencies. An example application with multiple threads is shown in Figure 1. This figure shows how InK is the first intermittent computing system that allows developers to create programs composed of multiple distinct *prioritized* threads comprising sensing, computing, and communication; to **schedule** periodic sensing tasks, **respond** to events in the environment, and **adapt** to changes in energy harvesting availability—all while managing intermittent power failures, memory consistency, and timekeeping duties behind the scenes with low overhead. InK is generic, enabling reactive sensing applications despite intermittent failures, on a multitude of hardware.

Contributions: We make the following contributions in this paper supporting the InK reactive runtime:

(1) **Event-Driven Programming:** we introduce a new programming model and several new abstractions for intermittent computing; comprised of *task threads* with different priorities, inter- and intra-thread control flow declarations, inter-thread communication interfaces, event notification and handling mechanisms, and time management.
(2) **Reactive InK Runtime:** we design and implement a reactive runtime featuring a*preemptive* scheduling policy; enabling several (in)dependent task threads and applications with different priorities to run in an interleaved manner, responding to energy, time, and sensing events, and ensuring power failure resilience, memory consistency, and correct control flow.
(3) **Performance Comparison and Reactive Applications:** we evaluate InK against state-of-the art systems and find our approach is up to 14 times more responsive to events in realistic intermittent power condition. We develop, for the first time, event-driven and reactive applications for batteryless sensors and tested them on different platforms like an intermittently-powered small batteryless robot. We also conduct a user study that demonstrates usefulness of InK.
(4) **Open Source Release:** we release InK [1][1] as a open-source resource to the community; with example applications to increase the impact of this work and batteryless sensor networks.

2 BACKGROUND AND RELATED WORK

The vision of sustainable, and ubiquitous computing in the IoT [43] will likely require sensor networks to leave their batteries behind and scavenge energy from the ambient [18, 44, 46]. Long lifetimes, low maintenance, and lower ecological impact make *batteryless* devices a promising alternative to the battery-powered devices used today. However, unpredictable energy availability leads to intermittent computation from frequent power failures. Despite new hardware platforms and energy management strategies [20], and an array of approaches for application development (see Table 1), creating batteryless sensing applications is difficult, as evidenced by the lack of widespread deployment, and our own studies of developer adoption (Section 4.5).

2.1 State of the Art

We detail related work below and in Table 1.

Computational RFID Runtimes. Dewdrop [9] was the first runtime to use unstable harvested energy in order to run tasks. A task is considered a short computation which should complete without any interruption. Unfortunately, Dewdrop is not power failure-immune since the program and the computation state is lost after a

[1]We invite the reader to explore the source, documentation, and resources for InK: https://github.com/TUDSSL/InK.

Model	Control flow	Mem. type	Journaled Data	No Dedicated HW	ISR interaction	Concurrent Apps.	C1	C2	C3	C4
TinyOS [31], Contiki [17]	Task-based	DRAM + Flash	None	✓	✓	✓	✓	✓	✓	✓
Dewdrop [9]	Task-based + Scheduler	SRAM + FRAM	None	✓	✗	✗	✓	✗	✓	✗
Mementos [39]	Instruction-based	SRAM + FRAM	Reg. + SRAM	✓	✗	✗	✓	✗	✓	✗
DINO [32]	Instruction-based	SRAM + FRAM	Reg. + SRAM + NV vars.	✓	✗	✗	✓	✗	✓	✗
Hibernius++ [5]	Instruction-based	SRAM + FRAM	Reg. + SRAM	✗	✗	✗	✓	✗	✓	✗
QuickRecall [28]	Instruction-based	FRAM	Reg.	✗	✗	✗	✓	✗	✓	✗
Ratchet [52]	Instruction-based	FRAM	Reg.	✓	✓	✗	✓	✗	✓	✗
Clank [24]	Instruction-based	FRAM	Reg.	✗	✗	✗	✓	✗	✓	✗
HarvOS [7]	Instruction-based	SRAM + FRAM	Reg. + SRAM	✓	✓	✗	✓	✗	✓	✗
Chain [12]	Task-based	SRAM + FRAM	PC + Channel data	✓	✗	✗	✗	✗	✗	✗
Alpaca [34]	Task-based	SRAM + FRAM	PC + NV vars.	✓	✗	✗	✗	✗	✗	✗
Mayfly [22]	Task-based + Scheduler	SRAM + FRAM	PC + Edge data	✓	✗	✗	✗	✓	✗	✗
InK (this work)	Task-based + Scheduler	SRAM + FRAM	PC + NV vars.	✓	✓	✓	✓	✓	✓	✓

Table 1: A comparison of relevant models to program embedded devices. Among them, InK is the only one that overcomes challenges C1–C4 (Section 2.2); NV vars.: *variables in non-volatile memory* , ISR: *interrupt service routine*, Mem.: *memory*, PC: *program counter*, Reg.: *registers*.

power failure. On the contrary, InK preserves the progress of the computation despite power failures.

Non-Volatile Processors. Integration of non-volatile memory to the processor architecture ensures immunity to power loss [33], which removes the burden of explicit checkpointing and recovery with software. Such processors, e.g. [49], are emerging especially for energy-harvesting scenarios in which the available power supply is unstable. However, these processors are still in the experimental stage [5]. Therefore, InK targets conventional off-the-shelf processors that have both volatile and non-volatile memory.

Existing Programming Models: There is an ample body of recent work aimed at programming batteryless devices by ensuring power failure resilience and memory consistency. *Checkpointing*-based systems Mementos [39], Hibernus [6], Hibernus++ [5], Quick-Recall [28], DINO [32], Ratchet [52], Clank [24] and HarvOS [7] journal the processor's *volatile state* in persistent memory. As an example, HarvOS operates at compile time using the control-flow graph of the program to place trigger calls that will measure the voltage level in order to decide checkpoint placement. On the other hand, compiler-based approaches like Ratchet analyze the program code in order to extract idempotent code sections. The Ratchet compiler places checkpoints at the beginning of these sections. However, all of the aforementioned checkpointing-based systems: (i) are not scalable [7, 32, 39], time and energy costs of checkpointing grows with the size of the volatile memory, increasing the possibility of exceeding the device's energy budget; or (ii) can only be applied to batteryless devices where all memory is non-volatile [24, 28, 52]. *Task-based* systems Chain [12], Alpaca [34] and Mayfly [22] introduce considerably less overhead by using a *static task model*: (i) the programmer decomposes a program into a collection of tasks at compile time and implements a *task-based control flow*; (ii) the runtime keeps track of the active task, restarts it upon recovery from intermittent power failures, guarantees its atomic completion then switches to the next task in the control flow. As an example, Chain proposes a programming model that requires programmers to structure their software into idempotent tasks and provides access to the non-volatile memory through input-output channel abstractions among the tasks.

Existing Operating Systems for IoT. The most relevant operating systems to InK in the IoT are TinyOS [31] and Contiki [17], both designed for *event-driven* wireless sensor network applications. Both lack a power failure-immune resource management functionality. For instance, TinyOS is designed with the assumption of no battery depletion in the short term. Therefore, its CPU scheduling and interrupt management services are not useful in the transiently-powered domain. Taking their code as a foundation and inserting checkpoints is not sufficient to enable batteryless applications due to several inconsistencies that occur during interrupt handling. For instance, when a timer event is fired and checkpointed just before a power failure, recovery by re-execution of the interrupt handler is not valid any more since the contents of timer registers are lost. Thus, it is not easy to make these operating systems power failure immune [11].

Need for Event-Driven Paradigm Shift: Traditional wireless sensors use batteries that provide reliable power. In TinyOS [31] and Contiki [17] nodes sleep until woken by hardware interrupts or timers to service events. In contrast, intermittently powered devices do not know if energy will be available in the future, so they greedily consume available energy at the cost of missing events or data of interest to the application (Figure 2). These missed events are missed opportunities for higher quality sensing outcomes; developers want control of their application despite the intermittency, and bringing the event-driven paradigm to batteryless sensing will enable this control. State-of-the-art *batteryless sensor* programming models fail to support event-driven applications since they mask first-class features of wireless sensor operation like external event handling, timekeeping, and energy management. Without these features, developers are unable to schedule tasks or perform periodic sensing. Moreover, these models are rigid, and do not allow for in-situ adaptation based on changing energy availability or data. Event-driven sensing is the next step for batteryless operation, but significant challenges exist.

2.2 Event-driven Sensing Challenges

We now discuss the challenges associated with using state-of-the-art programming models: Chain [12], Alpaca [34] or Mayfly [22], to implement event-driven application with three threads of execution

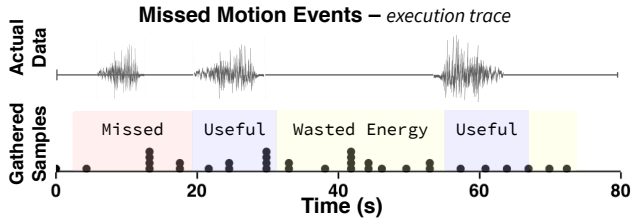

Figure 2: State of the art Intermittent programs, based on e.g. Chain [12], Alpaca [34], Mayfly [22], opportunistically gather data, missing important events or wasting scarce energy.

listed in Figure 1: TH1–TH3. Unfortunately, this application is not feasibly implemented without missing events, wasting energy, and reducing application quality.

C1–Responding to Events: With Chain, Alpaca and Mayfly *these three task threads cannot operate concurrently.* To enable event response, another task that constantly polls the energy level (TH1), the motion (TH2), and the elapsed time (TH3) has to be inserted that could trigger the threads. However, as shown in Figure 2 where an application samples an accelerometer, events can be missed and energy wasted. Task based and checkpointing-based programming models for intermittent computing are rigid in their specification and inherently non-reactive. To approximate event-driven sensing, tasks *check* shared global variables in order to change the control flow upon events. Such *polling*-based decision making puts extra effort on the programmer, wastes considerable amounts of energy, might intercept timely responses and also breaks the memory model—see C3. In order to respond to events in a timely manner, batteryless systems require a dynamic scheduling mechanism that can switch between different threads at runtime. However, this is not an easy task since the scheduler itself should work correctly despite power failures, ensure forward progress of computation, and maintain memory consistency of the running threads—a crucial difference as compared to the existing schedulers.

C2–Scheduling Tasks: Aforementioned programming models *cannot schedule events in the future, or perform periodic sensing tasks* as in TH3. Scheduling tasks using one-shot or periodic timers is a common action in battery powered sensors with a reliable notion of time and persistent power. Again, shared global variables need to be polled continuously by tasks in order to detect periodic events. Scheduling events, like sampling an accelerometer, are a way for a developer to gather information or perform a task at the exact moment it is necessary. Without this ability, intermittent programs are doomed to oversample at the wrong time, miss important events or actions in the environment, and waste precious energy and compute resources. Keeping track of time is challenging as compared to general purpose embedded systems. To schedule tasks, batteryless systems need a power failure resilient timer subsystem that will not lose track of time despite intermittent power. Scheduling mechanisms can leverage this subsystem to make scheduling decisions.

C3–Handling Interrupts: The memory model of Chain, Alpaca and Mayfly allow tasks to access internal non-volatile memory via input/output abstractions (e.g. *channels* in Chain). Global memory

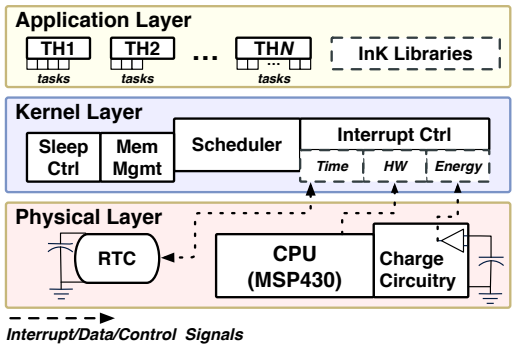

Figure 3: InK system overview; developers define task threads comprised of multiple tasks that are compiled with user and InK libraries in the application layer. The kernel layer interfaces with hardware to gather events, schedule task threads, manage time, and connect events to application level event handlers.

is not accessible to tasks: each task can only read the outputs of predecessor tasks and write to the inputs of the successor tasks. By this means, memory consistency issues are avoided. As a consequence, sharing global variables among tasks and interrupt service routines (and polling) *breaks existing memory models*—making event-driven applications infeasible and continuous sensing to be the only approach. To support event-driven applications, interrupt service routines (ISRs) should be able to activate tasks. A reactive scheduler is required that eliminates polling and abstractions are needed to let tasks receive data from ISRs without data races and breaking memory consistency. All these issues, in particular memory inconsistencies arising from the fact that ISRs are not atomic and in turn re-executable, are unique challenges belonging to intermittent systems.

C4–Adaptation: With Alpaca, Chain, and Mayfly *tasks run a high risk of starvation* as control flow cannot be interrupted or changed based on the changing external environment, or programmer insights. Tasks in intermittent computing applications always run the risk of starvation; sometimes there is not enough energy in the environment to power any computation, but the rigid task models of the state-of-the-art make this more likely. If a single link in the chain of tasks is never able to complete, either because of low energy or a programmer error, then all subsequent tasks starve. Consider that sometimes applications have multiple actions that can be done at any given time: with current programming models, these actions must be put in a sequence with rigid control flow. Developers have few avenues to respond to this; they cannot bake in runtime logic to handle changing and unpredictable energy situations or new application requirements.

3 INK: THE REACTIVE KERNEL

We built the InK kernel and programming model to enable development of reactive, timely, and event-driven applications for intermittently-powered batteryless sensing devices. This section describes the design and implementation. InK is designed to simplify or eliminate the event management and intermittent programming challenges described in Section 2 in three key ways:

(1) **Event classification:** We classify three categories of events that are encountered in energy harvesting sensors that could lead to longer periods of failure, task starvation, and inability to precisely schedule or time execution of a task.
(2) **Task threads Abstraction:** To respond to events, we introduce a novel concept, task threads, that have unique priority and encapsulate multiple restartable atomic tasks dedicated to a particular job.
(3) **InK Kernel:** We design a scheduler and kernel that efficiently manages multiple task threads, handles forward progress of computation, ensures memory consistency, and keeps track of time. InK kernel provides a new programming model and language structures to develop event-driven battery-less programs including services for inter-task thread communication, event notification and handling, and timekeeping.

The **Intermittent Kernel (InK)** (shown in Figure 3) enables developers to write adaptive programs, reduces task starvation, and allows periodic sensing and timely response to externally generated events[2].

3.1 Intermittent Computing Events

Development of InK is motivated by the lack of event handling in intermittent computing literature. Certain types of event can cause problems if not handled differently in the context of intermittent computing versus a traditional continuously powered (battery laden) device. Handling each of these events requires a new programming model and a more dynamic runtime operation beyond the static task based models and the rigid and inflexible time focused models in the state-of-the-art. We integrate handling of each of the following event types in InK.

Energy Thresholds: Energy harvesting batteryless devices only store small amounts of energy and expend it quickly. The amount of energy available for any period is not constant; it changes based on the time of day (for example in outdoor solar environments), the weather, and the location (if mobile). Static task models are not robust to this energy irregularity; if a high energy radio broadcast is set to execute in a low energy situation, that task will never complete. Moreover, if a low energy reading on an accelerometer executes in a high energy situation, excess energy is wasted. Recent hardware designs like UFoP [21], Flicker [20], and Capybara [13] can capture this energy thresholding phenomenon, however current programming models do not associate tasks with their energy requirement, potentially exposing them to starvation.

Timers: Scheduling events in the future is difficult with intermittently powered devices because maintaining time through power failures is not trivial. When the MCU loses power, an external device powered by a small capacitor can support timekeeping until the MCU turns back on. A notion of time beyond timestamps and data expiration like in Mayfly [22] can provide useful scheduling mechanisms for periodic or single shot tasks.

Hardware Interrupts: Nearly all sensing devices generate interrupts of some kind. Sensors like accelerometers, gyros, and magnetometers can gather data without any involvement from the MCU, storing this in a buffer and then alerting the MCU via an interrupt pin when the buffer is full. Analog sensors have thresholding circuitry that will wake a MCU when a point is reached. These hardware interrupts are not captured by current programming models, but are incredibly valuable to battery powered sensors for extending battery life and will be valuable to batteryless sensors by increasing MCU responsiveness (by allowing the MCU to sleep).

3.2 InK Design Space

Before proceeding with the design and implementation details of InK, we outline trade-offs in the design space for intermittent programming models and clarify some of our high level design decisions.

Task-based versus Checkpointing. Two programming models dominate intermittent computing; task-based and checkpoint based (see Table 1). In our view, a task-based system provides language scaffolding that enables reactivity without high runtime cost or extensive static analysis. Tasks have traditionally been seen as a useful abstraction to implement scheduling and to enable concurrent applications. Moreover, task-based systems handle forward progress and memory consistency with less overhead. However, the task abstraction puts a burden on the programmer to decompose the program code into tasks (or task threads composed of several tasks, in the case of InK) and define the control flow. This also requires explicit data handing to ensure memory consistency which is seen in other task-based systems [12, 34]. An alternative method using automatic checkpoints can be imagined to implement a reactive system. This requires the programmer, compiler, or runtime to place checkpoints correctly while respecting memory consistency during event handling. Novel checkpoint placement policies to ensure a correct and consistent system execution would be required. Tasks, instead of checkpointing, lend themselves more naturally to scheduling unique threads of execution and provide scaffolding for dynamic execution to overcome starvation and ensure timely execution.

Dynamic versus Static Scheduling. In batteryless systems tasks should only start execution when sufficient energy is available. On the other hand, tasks that do not require much energy that are executed frequently will starve high energy tasks. For programming models with static tasks this starvation possibility depends on how the programmer defines the task graph and the size of tasks. Once deployed this static schedule cannot react to changing energy conditions. A dynamic scheduling method can solve this at the expense of higher computational overhead. We take a dynamic approach with energy level-driven scheduling enabled by priorities that identify energy requirements as well as criticality of the task. This introduces a higher programmer burden as priorities are decided by the programmer. The tension then becomes managing programmer burden, complexity of dynamic scheduling, and starvation. Our choice of scheduling algorithm matches the requirements of most applications with an implementation which introduces reasonable overhead. We provide details in following sections.

[2]Find source code, documentation, and tutorials at our website [1].

Listing 1: Task thread code for TH2 and Energy ISR.

```
// task-shared persistent variables.
__shared(int data[10]; int i;);

// the entry task of the thread
ENTRY(Sense){
  // sample sensor
  int read = __sample_acc();
  // data[i] = read
  __SET(data[__GET(i)],read);
  ...
  NEXT(Features); // next task is sample
}
...
TASK(Classify){
  ...
  //write pipe
  __WRITE_PIPE(TH2,TH1,value);
  ...
  NEXT(null); //thread finishes
}
...
_interrupt(HIGH_energy)
{
  ...
  event.data = dataptr; // data pointer
  event.size = datasize; // data size
  event.timestamp = __getTime();
  // post to TH1's event queue
  __SIGNAL_EVENT(TH1,&_event);
  ...
  /* turn on CPU */
  __bic_SR_register_on_exit(LPM3_bits);
}
```

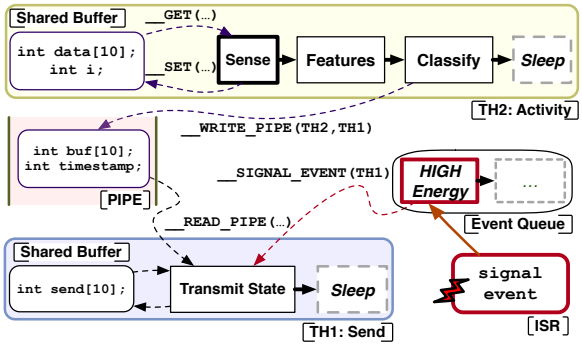

Figure 4: Overview of InK execution/memory model and task threads/ISRs interaction. Arrows indicate API calls of InK services.

Preemption. Dynamic scheduling requires some concept of preemption to provide flexibility in the face of changing energy availability. The design trade-off comes from the coarseness of the preemption strategy. InK scheduler preempts task-threads on the boundary of individual tasks. This task level coarseness of preemption ensures reactivity with less switching overhead (if for example the level of preemption was at the instruction level) while maintaining task atomicity and avoiding concurrency errors. This comes with the price of less reactivity since control flow is changed only after the execution of the active task is finished. Alternatively, the scheduler could preempt tasks at any point during their execution. However this requires checkpointing that introduces extra memory and compute overhead as well as the possibility of memory inconsistencies.

3.3 InK Execution Model and Task threads

Taking into account these event types and design trade-offs we discuss the implementation of InK. InK handles the previously mentioned events by introducing *power failure proof task threads*. These task threads are the main building blocks of an InK program. A *task thread* responds to events and ISRs that triggers corresponding event-handling. An example implementation of `[TH2: Activity]` as described in Figure 1 is presented in Listing 1 and the corresponding execution and memory model is presented in Figure 4. A summary of InK language constructs are given in Table 2.

Task Threads: A *task thread* is a lightweight and stack-less thread-like structure with a single *entry* point that encapsulates zero or more successive *tasks*. These tasks can do computation, sensing, or other actions, are idempotent, atomic, and have access to shared memory. Each *task thread* has a unique *priority* and accomplishes a single objective, e.g. periodic sensing of accelerometer. In order to preserve the progress and timeliness of computation despite power failures, InK kernel keeps track of each task thread by maintaining a *task thread control block* (TTCB) in non-volatile memory[3]. TTCB holds the *state* and the *priority* of the task thread, pointers to its entry task, to the next task in the control flow and to *buffers* in non-volatile memory that holds *task-shared variables*.

Task Thread Scheduling: The InK kernel implements *preemptive* and *static priority-based* scheduling of task threads: the InK scheduler always executes the next task in the control flow of the highest-priority task thread. Upon successful completion of this task, the pointer in the corresponding TTCB is updated so that it points to the next task in the control flow. In InK, tasks *run to completion* and can be preempted only by interrupts. Therefore, task thread preemption may only happen at tasks boundaries. When an ISR preempts the current task, it might activate other task threads of high-priority that are waiting for the corresponding event. Then, InK does not switch control to the higher priority task thread immediately; it waits for the atomic completion of the current task.

3.4 InK Memory Model

Tasks inside a task thread communicate with each other by manipulating *task-shared variables*. InK adheres to the data *encapsulation* principle by limiting the scope of these variables to the tasks of the corresponding task thread. Therefore task-shared variables are bound to the tasks that manipulate them and they are kept safe from misuse and interference by other task threads. InK allocates these variables in the non-volatile memory and they are double-buffered [32] to preserve data consistency across power outages—namely an *original* buffer holding the original copies and a *privatization* buffer holding the task-local copies [34].

Data Privatization: The TTCB of each task thread holds *pointers* to these buffers. Before running any task, InK initializes the privatization buffer by copying the contents from the original buffer. Tasks can read/modify only the content in the privatization buffer (via `__GET` and `__SET` interfaces). On a successful task completion, the buffer pointers are swapped so that the outputs of the current task are committed *atomically*.

Inter-Thread Communication: InK facilitates inter-thread communication through *persistent pipes*. A pipe is a unidirectional buffer in non-volatile memory with a timestamp. Any task inside the producer task thread can write to a dedicated pipe so that any task in the consumer task thread can read and perform computation by considering the timeliness of the data. Since tasks cannot preempt

[3]Commercial off-the-shelf microcontrollers like the MSP430FRxxxx (a common microcontroller for intermittent computing) have volatile SRAM and non-volatile FRAM memory segments. The MSP430FR5969 has only 2KB of SRAM and 64KB of FRAM.

InK Language Construct	Explanation
`__shared(...)`	Declares task-shared protected variables
`TASK(`**name**`)`	Declares an atomic task with given **name**
`ENTRY_TASK(`**name**`)`	Declares a task that will be the entry point of a task thread
`NEXT(`**name**`)`	Delivers control flow to the task with a given **name**
`__EVENT_DATA`	Holds the pointer to the event data in the event queue of a task thread that should be accessed by the entry task
`__EVENT_TIME`	Holds the timestamp of the current event in the event queue of a task thread
`__GET(`**x**`)`	Returns the value of the task-shared variable **X**
`__SET(`**x**`,`**val**`)`	Sets the value of the task-shared variable **X** to **val**
`_CREATE(`**priority**`,`**entry**`)`	Declares a task thread with a given entry task **entry** and **priority**
`_SIGNAL(`**priority**`)`	Activates the task thread with given **priority** within the context of a task thread
`_STOP(`**priority**`)`	Stops the task thread with given **priority** within the context of a task thread
`_interrupt(`**signame**`)`	Defines an interrupt handler with given service point **signame**
`__SIGNAL_EVENT(`**priority**`,`**event**`)`	Pushes event data **event** to the event queue of the task thread with given **priority** and activates it from ISR
`__CREATE_PIPE(`**src**`,`**dst**`,`**size**`)`	Creates a pipe structure of given **size** between task threads **src** and **dst** in order to share data between them
`__GET_PIPE_DATAPTR(`**src**`,`**dst**`)`	Returns the pointer to the data stored in the pipe between task threads **src** and **dst**
`__SET_PIPE_TIMESTAMP(`**src**`,`**dst**`,`**x**`)`	Sets the timestamp of the pipe between task threads **src** and **dst** to the given value **x**
`__GET_PIPE_TIMESTAMP(`**src**`,`**dst**`)`	Returns the timestamp of the pipe between task threads **src** and **dst**

Table 2: Summary of InK Language Constructs. The system API includes necessary calls for task and task thread declaration, memory consistency and control flow handling, event and interrupt management, as well as inter-task thread communication. ,

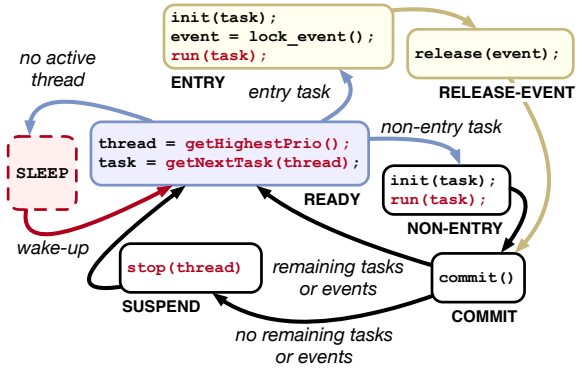

Figure 5: The InK scheduler state machine that selects the next task in the control flow of the thread of highest priority, ensures forward progress and puts the CPU in sleep mode when possible.

each other and also pipes are unidirectional, pipe access do not lead to data races even upon power failures.

For the sake of efficiency and simplicity, we did not provide extra protection over the persistent pipes that enable data sharing among task thread—the consistency of these shared memory regions should be explicitly handled by the programmer. Alternatively, this protection could be handled by InK, however, this increases the implementation complexity and overhead of our system.

3.5 Reactive Execution

In order to ensure reactivity and adaptability, InK implements the state machine depicted in Figure 5 and maintains a *scheduler-state* variable in non-volatile memory in order to ensure forward progress despite power failures.

The Scheduler Loop: At each loop iteration, the scheduler selects the task thread of highest priority and executes the next task in the control flow of the selected thread. During task execution, the scheduler (i) initializes the task privatization buffer via `init`; (ii) for the entry tasks, it locks the event data that triggered thread execution via `lock_event` (to eliminate data races between ISRs and tasks—see following sections), (iii) it executes the task via `run`, (iv) for the entry tasks it releases the event via `release`, (v) it commits the tasks modifications by swapping buffer pointers, (vi) it suspends the thread if there are no dedicated events or remaining tasks. If there is no thread in ready state, the scheduler puts the micro-controller into low-power mode, saving energy and waiting for an interrupt for activation. The state machine enables progress of computation since it continues from the state it is interrupted.

Reducing Starvation: Tasks inside task threads and ISRs can activate and deactivate other task threads and change control flow *dynamically*. In existing run-times, e.g. Mayfly and Alpaca, control flow is static, in the sense that all subsequent tasks in the chain should wait for the completion of predecessor tasks. This leads to the problem of *priority inversion* since high-priority tasks can be blocked due to the lower-priority tasks holding the CPU. On the contrary, since the InK scheduler alternates between the aforementioned states, it can switch execution to the high-priority thread: first, the kernel awaits the completion of the interrupted current task inside the lower priority thread; then it starts executing the entry task of the high-priority thread.

Responding to Events: Each task thread in InK has a dedicated non-volatile *event queue* that holds the events generated by ISRs. When any event is generated, the corresponding task thread is activated so that the thread execution will start from its *entry task*. In InK execution model, the event data is only accessible by the *entry task* of the task thread: the *entry task* locks the event data (see `lock_event` in Figure 5) to eliminate data races between ISRs and tasks. The entry task reads the event data and modifies necessary task-shared variables and then the event lock is released so that the event data will be removed from the event queue.

Event Handling: Circular buffers hold the data to be shared between an ISR and a task thread to prevent data races. The buffer handling introduces additional implementation and execution overhead but eliminates the need for the programmer to be involved in this process. As an alternative, unbounded buffers could be implemented, however management of a dynamically growing buffer

introduces extra overhead at run-time for already memory constrained devices[3].

Interrupt Management: The *pre-processing* of an interrupt is performed by the corresponding ISR. Then, the rest of the computation is done by a task thread. When an interrupt is generated, the corresponding ISR delivers the received or generated data to the upper layers of the system and notifies task thread. Event queues are *ring buffers* dedicated for each task thread. They form an intermediate layer that prevents race conditions and preserves the event data consistency by eliminating ISRs from modifying task-shared data directly. When the event queue is full InK removes the event that has the oldest timestamp from the event-queue to increase the probability of having fresh data. Once an interrupt is generated, the task threads is notified by creating an *event* holding a pointer to the *ISR data* and its *size*, and a timestamp indicating the time at which interrupt is fired. The corresponding task thread is notified (via `__SIGNAL_EVENT`) by passing the pointer event structure so that the event will be placed in the event queue of the given task thread *atomically*.

3.6 Scheduling Events and Timers

InK builds a timer sub-system using an external *persistent timekeeper* [23] that keeps track of time across power failures: (i) when the MCU is running, its internal timers are used to measure elapsed time; (ii) upon a power failure, the external timekeeper keeps running and provides elapsed time until recovery. The timer system implements a *timer wheel algorithm* to provide two types of timers for the task threads: *expiration timers* and *one-shot/periodic timers*.

Expiration Timers: Task threads set expiration timers in order to enable timely execution of task threads and stop unnecessary and outdated computation if necessary; analogous to Mayfly [22] concepts of `expiration`. As an example, data read from a sensor should be processed within a time constraint and if computation exceeds the required deadline the outputs of the computation are not useful any more. When an expiration timer fires, the corresponding task thread is evicted so that it does not consume systems resources, e.g. CPU, anymore.

One-Shot/Periodic Timers: One-shot and periodic timers are used in order to schedule events in the future and generate periodic events, e.g. activating task thread at a given frequency. Since most of the sensing applications are periodic, these timers are the foundations of task threads that perform periodic sensing, these timers build on the *persistent timekeeper* to keep time across power failures.

4 EVALUATION OF INK

We proceed with the experimental evaluation of InK. We compare InK against its counterparts by implementing sensing applications that require timely response to various events. In our evaluation we measure several metrics to observe the reactiveness as well as overhead in terms of time, energy and system resources. Considering these metrics, we show that InK improves the reactivity of batteryless sensing applications **up to 14 times** as compared to its counterparts, by introducing a reasonable system overhead. Moreover, our user studies demonstrate that InK is the preferred language for new generation event-driven applications for batteryless systems. Finally, our case studies show that InK enables new, never before seen sensing applications. We aim to help developers in learning and contributing to InK by providing resources to the community with a dedicated website [1].

4.1 Experimental Setup

We describe the experimental setup used in assessing the performance of InK against existing state-of-the-art runtimes and as a stand-alone system. Our setup considers replicability of results and varying types of energy supply.

Target Embedded Platform: The experiments were conducted using TI MSP-EXPFR5969 evaluation boards [26]. This platforms uses 16 MHz MSP430FR5969 MCU with 64 kB and 2 kB of of non-volatile (FRAM) and volatile memory (SRAM), respectively. We set the micro-controller frequency to 1 MHz during our experiments. Whenever necessary, InK sensing system interacted with low-power accelerometer [47], low-power microphone [3] and infrared transmitter (Vishay Semiconductor TSOP38238)/receiver (generic 950 nm infrared LED) pair.

Runtimes for Intermittently-Powered Devices: InK was compared against two state-of-the-art runtimes: MayFly [22] and Alpaca [34]. For each runtime we have prepared the same application introduced in the subsequent sections and composed of the same set of tasks and control flow.

Measurement Equipment: We used the Saleae logic analyzer [41] to measure the performance metrics of all applications that were implemented during experiments. Data was parsed with dedicated, on-line accessible [1], Python scripts.

Intermittent Power Supply: We used two setups to provide repeatable experimentation: a real wireless power supply (used in InK case studies) and emulated power (for repeatability and replicability of comparative measurements). *Real wireless power supply:* To power MSP430 evaluation boards, we used Powercast [14] TX91501-3W transmitter emitting RF signal at 915 MHz center frequency to P2110-EVB receiver [14] (one per each MSP430 board) co-supplied 6.1 dBi patch antenna. *Controlled-power supply:* we considered two approaches per different experiments. Approach (1) used the Ekho platform [19] that replays a realistic and repeatable (recorded from real harvesters) I-V surface to the MSP430 evaluation board. Approach (2) is based on a dedicated MSP430 evaluation board that interrupts the power supply of the other board by controlling the RST pin [51, Sec. 5.12.2], with a uniformly distributed interrupt period in the interval of [0, 0.5] seconds.

4.2 Reactive Application Performance

We start with demonstrating the main strength of InK: fastest reactivity of programs for transiently-powered devices.

Implementation: We implement a *batteryless condition monitoring* application in InK. Two threads are considered: one (first priority) that detects whether a new event happens (like arrival of a new item to a CNC router), and a (second priority) event which detects the condition of a device (like specific vibration of a machine). The first thread is triggered by a sound overthreshold and implements an

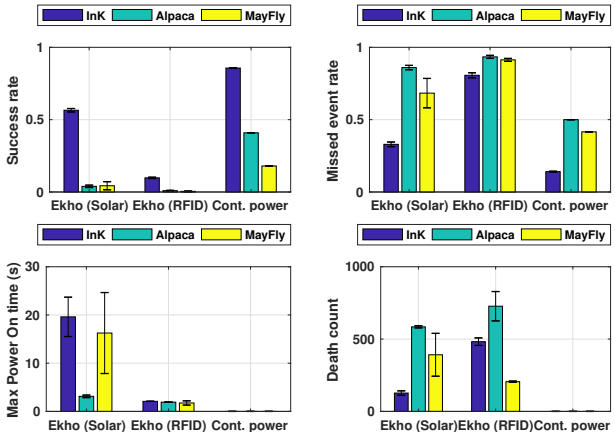

Figure 6: Performance metrics of reactive sensing application, implemented using InK, MayFly and Alpaca. Performance of all runtimes is compared to the continuous power case. InK improves the reactive application performance by orders of magnitude for *all* metrics.

FFT (analyzing data from a microphone), while the second samples and records data from the accelerometer periodically. This application was also implemented using MayFly [22] and Alpaca [34] runtimes for comparison. Since Mayfly and Alpaca are not event-driven and they do not allow interaction with interrupts, the only way to implement this application was to use a control flow that implements a *polling* loop: the microphone is checked to catch the sound overthreshold and perform FFT if required; and then reading the accelerometer and performing sensor data related computations continuously. Since Alpaca has no notion of time (contrary to InK and MayFly) we could not consider timeouts. Also, Alpaca makes it impossible to use external libraries, e.g. accelerated FFT library for TI MSP430 MCU. Therefore, in the comparison we mimic the FFT operation by a constant delay loop.

For the replicability of power failures from energy harvesting an Ekho [19] emulator powered the target embedded platform. Ekho repeats, in an infinite loop, a pre-recorded one minute (i.e. a maximum length Ekho can support) I-V curve recorded from one of two energy harvesting sources: (i) 22×7 mm IXYS Solarbit solar panel [27] which was relocated from indoor to cloudy outdoor sunlight—representing the trace with long periods of energy availability; and (ii) a WISP harvester powered by Impinj Speedway 420 RFID reader [25] with readers' antenna at initial 25 cm distance from WISP was relocated to 10 cm and again to 25 cm—representing trace with very high power intermittency rate. Data from the condition monitoring application was collected for five minutes of continuous operation, for five runs with each runtime.

Metrics: We have measured the following reactivity metrics for each runtime: *Success Rate*—the rate of successfully executed highest priority events; *Missed event rate*—how many high priority events are missed due to either power failures or on-going low-priority computation; *Maximum 'Power On' Time*—the longest duration that the device was alive (at intermittent power); and *Death Rate*—the number of power failures (at intermittent power) during the experiment.

Power	Power Failures	Motion over Threshold	Catch Rate	IR Trans.
Continous	0	8	1	6
RF	36	6	0.66	1

Table 3: The response to the activity event of the batteryless voice-controlled activity recognition application.

Results: The result is presented in Figure 6. We observe that InK is the most reactive runtime of all, **improving over the success rate of Alpaca and Mayfly by 14 and 13 times**, respectively (for the solar power case). Naturally, InK cannot obtain a perfect success rate, simply because of death during the interrupt arrival. Also, InK misses the least number of priority threads (Figure 6 (top, right)) compared to other runtimes. Mayfly is less reactive compared to Alpaca, as it must check the timing constraints of every single task in between actual task execution; for sophisticated programs with more than ten tasks this reduces the percentage of time Mayfly can poll for events. We note that Mayfly died fewer times in our experiments, due to implementation differences in startup where Mayfly stores energy to enable timekeeping, however, we note that this reduced death count did not increase reactivity as Mayfly was unavailable for compute during this startup period. InK had the highest effective 'power on' time and the lowest death count among all runtimes: Figure 6 (bottom left and right).

4.3 Real-World Event-Driven Applications

We demonstrate several never before seen batteryless applications enabled with the event-driven programming supported by the InK kernel. The development of these applications is not feasible with existing runtimes like Alpaca or Mayfly due to the challenges C1–C4 listed in Section 2.

4.3.1 Batteryless Event-Driven Sensing. We start with the first case study: voice-controlled batteryless activity recognition.

Application Challenges: We used a low-power accelerometer [47] for the classification of activity and a microphone [3] for voice recognition connected to TI MSP430 board. Initially the system is sleeping and a voice recognition task thread is waiting to respond to the accelerometer interrupt to detect motion over a threshold (*requirement for C1 and C3*). When this event is detected, the voice recognition task thread starts responding to the interrupts generated by the microphone and recognizes the 'start' command (*requirement for C1 and C3*). After the start command is detected, the activity recognition task thread is activated to periodically sense and classify ongoing activities (*requirement for C2*). When the available energy is over 2.5 V the comparator COMP_E of MSP430fr5969 is programmed to generate an interrupt. When this energy threshold interrupt is generated (*requirement for C4*), the control is switched to task thread that sends the classification results using an infrared transmitter via simple OOK modulation.

Results: We collected the success rate of activity recognition during five minutes. The system is powered continuously (to use as a reference for comparison) and by using Powercast transmitter. Table 3 shows our measurement results. With continuous power, 8 motion threshold interrupts were detected and all of them were processed on time. Among them, energy levels allowed to perform 6

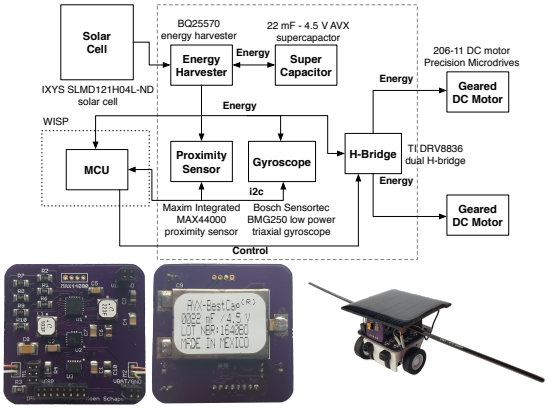

Figure 7: Batteryless small autonomous robot: top figure—block diagram; bottom left—robot PCB design (front side): harvester (U1), gyroscope (U2) and H-bridge (U3); bottom center–robot PCB design (back side): super-capacitor), bottom right—complete robot.

Design	Motion	Speed (cm/s)	Size (mm)	Weight (g)	Storage (mAh)	Time to recharge	Recharge method
Roverables [16]	wheel	N/A	40×26	36	100	45 min	inductive
Zooids [30]	wheel	50	26×26	12	100	1 h	manual
mROBerTO [29]	shaft	15	16×16	10	120	1.5 h	manual
GRITSBot [36]	wheel	25	31×30	60	150	1 h	contact
Kilobot [40]	vibration	1	33×33	17.6	160	3 h	manual
HAMR-VP [8]	legged	44	44×44	2.3	8	3 min	manual
This robot	wheel	25	35×40	22	0.006	<5 s	solar

Table 4: Comparison of our batteryless harvesting robot against state-of-the-art small robotic platforms.

IR transmissions after activity classification. With RF power, power failures were observed 36 times, 6 motions above thresholds were detected and almost 4 of them processed on time. Energy levels allowed to perform only 1 IR transmission in this case.

4.3.2 Batteryless Intermittent Actuation. We continue with the second case study: a tiny batteryless robot designed to perform autonomous reconnaissance sensing tasks. Using this robot, we demonstrate that InK enables reactive control of batteryless energy-harvesting actuators.

Batteryless Robot Motivation: Referring to Table 4, state-of-the-art robotic platforms are battery-dependent and require physical/proximity contact to recharge. Our idea is to remove these obstacles by providing power *directly* from the harvesting source (solar panel) to the environmentally-friendly storage (super-capacitor). The consequence is the intermittent movement of a robot. Movement stops after short move duration (in the order of seconds).

Robot Design: Figure 7 provides our robot design overview. The robot is designed around a WISP 5 [35] which allows the observer to send control messages from the RFID reader. WISP's TI MSP430FR5969 MCU serves as the main robot control system. DC motors were used as actuators. Two motors are mounted diagonally opposite from each other in a 3D-printed frame. Small plastic wheels with rubber tires are mounted directly on each of the motor shafts. Behind the motors a free running caster wheel is mounted to the frame. PWM controls the robot's speed and is used to reduce the average current

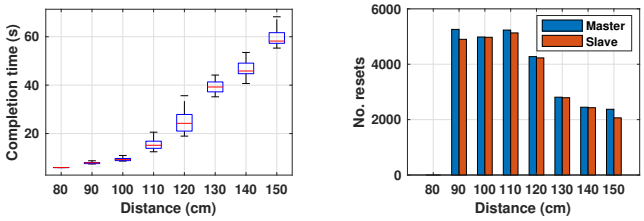

Figure 8: Intermittent communication experiment result: two MSP430 boards are connected together using UART RX/TX ports exchanging messages in an infinite loop, also connected to Powercast receiver boards and powered by the RF. Left figure: completion time, right figure: the number of resets at different distances to the transmitter.

consumed by the motor. A large bulk capacitor supplies short high current demand from the motors.[4]

Robot Control: We implemented a simple PID controller feedback loop to drive the robot. Our transiently-powered robot can make one movement, which requires multiple power cycles to complete. To finish the required move upon power failures and to capture the progress towards the movement target, several InK services are used. The PID control loop is implemented as a task thread that is scheduled to execute every time a robot is powered. At each periodic activation, the thread samples the yaw-rate using the gyroscope and executes the tasks of the control algorithm to update the motor parameters.

4.3.3 Batteryless Intermittent Communication. We conclude with the final case study. We demonstrate that InK enables a basic distributed processing system via communication over the serial port.

Batteryless Communication Challenges: Two master and slave TI MSP430 boards are powered using independent intermittent energy harvesting sources (one Powercast receiver per MSP430 board and placed at several distances to the RF Powercast transmitter—one per receiver). The master board implemented two task threads (*requirement for C1*): the main thread generates a random signal composed of 16 floating point values, transmits the signal to the slave board and computes the DFT of this signal; meanwhile another thread waits UART RX events (*requirement for C1 and C3*), receives the DFT result of the slave node composed of 16 floating point values and pushes the result to the pipe of the main thread and activates it (*requirement for C1*); then the main thread compares its results with the one in the pipe and toggles an output port if they are equal. The slave board implemented one task thread that waits for UART RX interrupt to receive the random signal from the master node (*requirement for C1 and C3*), computes the DFT of the received signal, sends the result to the master node. In order to keep track of the delivered packets, the sender side sets a one-shot timer (*requirement for C2 and C3*) and awaits acknowledgment (ACK) from the receiver side. If ACK is not received, the packet is re-transmitted.

Results: We monitored the output ports of the boards for 15 minutes per each Powercast transmitter/receiver distance. Figure 8

[4] In-depth information about the robot hardware and software and InK implementation of robot control algorithm is provided in [2, 42] and InK repository [1], respectively.

	InK			Alpaca			MayFly		
	Time (ms)	Memory (B)		Time (ms)	Memory (B)		Time (ms)	Memory (B)	
		.text	*.data*		*.text*	*.data*		*.text*	*.data*
AR	4151	3442	4459	8361	7970	724	4464	8700	2496
BC	546	2922	4433	912	6290	818	1019	8066	846
CF	495	2648	4693	199	8494	2352	—	—	—

Table 5: Execution time and memory consumption for three benchmark applications written in InK, Alpaca and MayFly. Since it was not feasible with Mayfly to develop *CF* application, its corresponding values are shown with —. Overall results show that InK's overhead is comparable with its counterparts.

present our measurement results. During the execution of the application, not only the computation but also the communication is interrupted frequently, especially at distances 80–120 cm—since time to charge is longer at further distances, this led to the longer duty-cycles, less power failures and number of completions. We observed that the applications on both master and slave nodes are always completed successfully despite frequent power losses.

4.4 InK System Overhead

We continue with assessing the overhead of InK by implementing common computation-based benchmarking applications and comparing their execution time, code size and memory requirements.

Implementation: The complete suite of benchmarking is composed of: (a) *Activity Recognition (AR)*: machine-learning enabled physical activity classification using locally generated accelerometer data, (b) *Bitcount (BC)*: bit counting in a random string based on sever different methods, cross-verifying correctness, and (c) *Cuckoo Filtering (CF)*: runs a cuckoo filter over a set of pseudo-random numbers and performs the sequence recovery using the same filter. We implemented these applications in InK, Alpaca and Mayfly using the same task partitioning and control flow. Unfortunately, loops are not allowed in a Mayfly task graph as the data and control flow are the same (leading to infinite data growth). Therefore, non-sensing applications like *CF* cannot be implemented in Mayfly because of the multi-level loops and control flow disassociation from data flow.

Results: For the fairness of the comparison, we run the aforementioned benchmarking applications on continuous power. To measure the execution time, we sampled the output port of the MCU using the logic analyzer that is toggled after the program completed its execution successfully and the results are correct.

Table 5 presents the summary of evaluation. We conclude that InK is always faster than MayFly and only slower than Alpaca for *CF*. Additionally, we measured the memory overhead and code size for all runtimes. The increased memory and execution overhead in selected applications is due to the fact that InK maintains queues and data structures in order to manage/schedule task threads and makes scheduling decisions at runtime for the sake of reactivity. Our results show that InK enables event-driven paradigm shift for batteryless devices while introducing a reasonable overhead as compared to Alpaca and Mayfly.

Point to Point Overheads: Table 6 presents detailed overhead of the InK runtime operations. When InK runs for the first time, all of the internal non-volatile state variables are initialized (denoted as the *Initial Boot* overhead). After the first boot, each *Reboot* requires only the initialization procedures of the MCU, peripherals and the recovery operations of the InK scheduler. The *Scheduling* overhead is introduced to select the thread of highest priority and execute the next task. *Task Init* and *Task Commit* overheads are introduced to prepare the privatization buffer and commit the modifications on this buffer to the original buffer atomically, respectively. The time spent for *Activating Thread* is required to change the state of the corresponding task thread to ready so that the scheduler will consider to run it later. *Event Register* is the overhead of committing an event in the event queue.

Operations	**≈Overhead (in μsec)**
Initial Boot	7900
Reboot	70
Scheduling and Selecting Next Thread	89
Task Init (10 B/1 KB shared data, resp.)	121/315
Task Commit	42
Activating Thread	75
Event Register	778

Table 6: Approximate overhead of the initialization and scheduler overhead.

4.5 User Study

We have performed an on-line user study to assess the usability of InK in programming intermittently-powered devices. The study is approved by the Human Research Ethics Committee of Delft University of Technology. The study suggests that (i) InK is *intuitive* and applicable to a varying set of sensing applications, and (ii) InK provides the *necessary constructs to write periodic sensing applications*.

Methodology: Participants were provided a link to the on-line survey (questionnaire and detailed answers in [1]) via a personal invitation. Survey was accompanied by a short document introducing the concept of intermittently-powered devices and the issues associated with programming such devices. Then, participants assessed three programs implementing non-ISR `[sense]` → `[compute]` → `[transmit]` loop written in InK, Aplaca and MayFly languages, after which a set of questions were asked. Additionally, participants had to assess the same program written only in InK, but implemented using interrupt service routines. Finally the participants were asked to write a simple InK program themselves (submitted to us for inspection), and again assess InK usability. An answer to each question was one from a five-level Likert-type scale (From *Strongly disagree* to *Strongly agree*). There was no time limit on the assignment and the survey could have been performed at the most convenient location and time for each participant.

User Pool: We have collected 22 responses in total from a pool of graduate (MSc/PhD level) students studying embedded systems, computer science and experienced scientific developers. 63% of the participants had more than five years of formal computing education. 82% of the participants had more than five years of programming experience. C was considered the language of choice for majority of participants, while 68% of them considered their knowledge of C as average and above average. Also, 42% self-assessed

themselves with above-average knowledge of embedded systems (compared to others with similar background, age and education). 73% of them considered their knowledge of intermittently powered embedded devices as low and very low. Participants were located in at least four different countries in Europe and North America.

Results and Discussion: Participants assessed the ease and intuitiveness of using all three languages in writing generic applications for intermittent devices: Alpaca being the easiest (18 strongly agreed or agreed), followed by InK (13 respective answers) and MayFly (only 8 respective answers). The same order applied to questions on the programming model flexibility. All thee codes were assessed equally in terms of programming mode completeness. All of them were assessed as the language that could be used for a variety of sensing applications(InK being the most selected one for this task).

Considering the task of assessing the reactive programming difficulty with InK, all responders provided their example InK code. 59% of the participants agreed that it was easy to understand how InK handled interrupts for intermittently-powered systems, while 57% agreed that it would be harder to understand if the code was written in plain C. 76% of respondents strongly agreed or agreed that InK provides the necessary constructs to write periodic sensing applications, with only 38% and 30% respondents for Alpaca and Mayfly, respectively. Only 23% agreed that it was hard to write InK program.

We acknowledge that our study is limited due to small sample size and difficulty in preparing the comparable program in three programming languages. Nonetheless, survey results indicate that InK is the right tool for reactive programming of intermittently-powered devices.

5 DISCUSSION AND FUTURE WORK

InK Community Building: InK is available online via [1]. Through this website we aim to help developers learn InK fast. Code of InK is open-source, with datasets accompanying this paper available for inspection as well. In the course of time we will be providing a new set of example programs and tutorials easing the learning process of reactive-based programming for intermittent devices.

InK application developer effort: An InK programmer does not need to reason about power failures or memory inconsistency—which confuse and frustrate even experienced developers—but needs to follow a new programming model that is different from existing ones targeted for continuously-powered systems. In particular, the programmer needs to (i) identify task-shared variables; (ii) provide a task-division and annotate task boundaries/inter-task dependencies; and (iii) define an explicit control flow. Future work can address removing this burden from the programmer: for example with a guided compilation tool that translates programs into InK, or with additions of features like module reuse.

Limits of reactivity: Although InK's goal is to enable reactivity for systems powered intermittently, it will never be able to provide the same reactivity as battery-powered or tethered devices. Simply, with no control over available power there is no feasible way to make a sensing system reacting immediately to stimuli. The question remains how big is the set of applications that can accept reduction of system responsiveness, while not compromising the quality of service. Using our transiently-powered robot as an example: what is the acceptable number of stops (and their duration) in-between consecutive moves that would make robot still considered *reactive*? Future work could investigate networked approaches to this problem, with higher density of cheap batteryless sensing devices, overall coverage increases. Multiple challenges in networking and synchronization must be resolved to make this a reality.

Dealing with peripheral I/O: Sensing systems must manage peripherals (sensors, memorys, radio) that have volatile state. On power failure, this state is lost and peripherals must be reinitialized. InK, nor any other runtime maintains peripheral state consistency, leaving this as a programmer burden. A common solution is to split input operations into two tasks: one task reads the sensor value and another task consumes it accordingly. This guarantees consuming the value once since tasks run in order and cannot preempt each other: the consumer task can be re-executed safely since its output is produced by the former task. However, to re-execute output operations safely at intermittent power, e.g. blink an LED exactly once, hardware assistance is required. Future work could leverage emerging non-volatile sensors, or build software models for handling of failure in peripherals.

Starvation, fairness, multi-tenancy: task threads shows the potential for multi-tenancy on batteryless, energy harvesting devices. However, multiple issues surround the practical use case where third party applications coexist peacefully on a single intermittently powered device.

6 CONCLUSIONS

We have shown that state-of-the-art programming models and runtimes for intermittently-powered systems are inadequate for developing real-world sensing applications: they do not respond to events in a timely manner, they do not react or adapt to changes in available energy, they do not schedule events to perform periodic sensing, and they do not handle interrupts while preserving memory consistency. To address these limitations, we introduced InK: the first reactive task thread scheduling kernel that facilitates event-driven applications for transiently-powered systems. We evaluated InK based on software benchmarks and compared its performance against InK counterparts using real hardware and real energy harvesting traces. Our results showed that InK significantly improves the reactivity of batteryless sensing applications by up to 14 times, introducing a reasonable overhead. We also demonstrated that InK enables never before seen sensing applications such as the first transiently powered robot.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers and our shepherd for their constructive criticism. We express our gratitude to Carlo Delle Donne for technical support during the project and to Brandon Lucia's Abstract Research Group at Carnegie Mellon University for numerous discussions. This research is supported by the Netherlands Organisation for Scientific Research, partly funded by the Dutch Ministry of Economic Affairs, under TTW Perspectief program ZERO (P15-06) within Project P4.

REFERENCES

[1] 2018. InK Website. https://github.com/tudssl/ink. Last accessed: Sep. 20, 2018.

[2] 2018. Intermittently-Powered Robot Website. https://github.com/tudssl/iprobot. Last accessed: Sep. 20, 2018.

[3] Adafruit. 2016. Silicon SPW2430HR5H-B MEMS Microphone Breakout Board (SPW2430). https://www.adafruit.com/product/2716. Last accessed: Apr. 1, 2018.

[4] Omid Ardakanian, Arka Bhattacharya, and David Culler. 2016. Non-Intrusive Techniques for Establishing Occupancy Related Energy Savings in Commercial Buildings. In *Proc. BuildSys.* ACM, Palo Alto, CA, USA.

[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 Trans. Comput.-Aided Design Integr. Circuits Syst.* 35, 12 (Dec. 2016).

[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 Syst. Lett.* 7, 1 (March 2015).

[7] Naveed Bhatti and Luca Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Devices. In *Proc. IPSN.* ACM/IEEE, Pittsburgh, PA, USA.

[8] Remo Brühwiler, Benjamin Goldberg, Neel Doshi, Onur Ozcan, Noah Jafferis, Michael Karpelson, and Robert J. Wood. 2015. Feedback Control of a Legged Microrobot with On-board Sensing. In *Proc. IROS.* IEEE, Hamburg, Germany.

[9] Michael Buettner, Ben Greenstein, and David Wetherall. 2011. Dewdrop: an Energy-aware Runtime for Computational RFID. In *Proc. NSDI.* USENIX, Boston, MA, USA.

[10] Gregory Chen, Hassan Ghaed, Razi M. Haque, Michael Wieckowski, Yejoong Kim, Gyouho Kim, David Fick, Daeyeon Kim, Mingoo Seok, Kensall Wise, David Blaauw, and Dennis Sylvester. 2011. A Cubic-Millimeter Energy-Autonomous Wireless Intraocular Pressure Monitor. In *Proc. ISSCC.* IEEE, San Francisco, CA, USA.

[11] Yang Chen, Omprakash Gnawali, Maria Kazandjieva, Philip Levis, and John Regehr. 2009. Surviving Sensor Network Software Faults. In *Proc. SOSP.* ACM, Big Sky, MT, USA.

[12] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proc. OOPSLA.* ACM, Amsterdam, Netherlands.

[13] 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 (ASPLOS '18).* ACM, New York, NY, USA, 767–781. https://doi.org/10.1145/3173162.3173210

[14] Powercast Corp. 2014. Powercast Hardware. http://www.powercastco.com. Last accessed: Mar. 30, 2018.

[15] Samuel DeBruin, Bradford Campbell, and Prabal Dutta. 2013. Monjolo: An Energy-harvesting Energy Meter Architecture. In *Proc. SenSys.* ACM, Rome, Italy.

[16] Artem Dementyev, Hsin-Liu Cindy Kao, Inrak Choi, Deborah Ajilo, Maggie Xu, Joseph A Paradiso, Chris Schmandt, and Sean Follmer. 2016. Rovables: Miniature On-Body Robots as Mobile Wearables. In *Proc. UIST.* ACM, Tokyo, Japan.

[17] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. 2004. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proc. LCN.* IEEE, Tampa, FL, USA.

[18] Shyamnath Gollakota, Matthew Reynolds, Joshua Smith, and David Wetherall. 2014. The Emergence of RF-Powered Computing. *Computer* 47, 1 (Jan. 2014).

[19] Josiah Hester, Timothy Scott, and Jacob Sorber. 2014. Ekho: Realistic and Repeatable Experimentation for Tiny Energy-Harvesting Sensors. In *Proc. SenSys.* ACM, Memphis, TN, USA.

[20] Josiah Hester and Jacob Sorber. 2017. Flicker: Rapid Prototyping for the Batteryless Internet-of-Things. In *Proc. SenSys.* ACM, Delft, The Netherlands.

[21] Josiah Hester and Lanny Sitanayah Jacob Sorber. 2015. Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors. In *Proc. SenSys.* ACM, Seoul, South Korea.

[22] Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. In *Proc. SenSys.* ACM, Delft, The Netherlands.

[23] 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 Trans. Emb. Comput. Syst.* 15, 4 (Aug. 2016).

[24] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proc. ISCA.* ACM, Toronto, ON, Canada.

[25] Impinj Inc. 2018. Impinj Speedway R420 RFID Reader Product Information. https://www.impinj.com/platform/connectivity/speedway-r420/. Last accessed: Apr. 8, 2018.

[26] Texas Instruments. 2015. MSP430FR5969 LaunchPad Development Kit. http://www.ti.com/tool/msp-exp430fr5969. Last accessed: Apr. 30, 2018.

[27] IXYS. 2011. IXOLAR High Efficiency SolarBIT Solar Panel. http://www.ti.com/lit/ug/tidu383/tidu383.pdf. Last accessed: Apr. 2, 2018.

[28] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. 2015. Quickrecall: A HW/SW Approach for Computing Across Power Cycles in Transiently Powered Computers. *ACM J. Emerg. Technol. Comput. Syst.* 12, 1 (July 2015).

[29] Justin Y. Kim, Tyler Colaco, Zendai Kashino, Goldie Nejat, and Beno Benhabib. 2016. mROBerTO: A Modular Millirobot for Swarm-behavior studies. In *Proc. IROS.* IEEE, Daejeon, Korea.

[30] Mathieu Le Goc, Lawrence H. Kim, Ali Parsaei, Jean-Daniel Fekete, Pierre Dragicevic, and Sean Follmer. 2016. Zooids: Building Blocks for Swarm User Interfaces. In *Proc. UIST.* ACM, Tokyo, Japan.

[31] Philip Levis, Sam Madden, Joseph Polastre, Rober Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. 2005. TinyOS: An Operating System for Sensor Networks. In *Ambient intelligence*, Werner Weber, Jan M. Rabaey, and Emile Aarts (Eds.). Springer, Berlin, Germany.

[32] Brandon Lucia and Benjamin Ransford. 2015. A simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proc. PLDI.* ACM, Portland, OR, USA.

[33] Kaisheng Ma, Xueqing Li, Karthik Swaminathan, Yang Zheng, Shuangchen Li, Yongpan Liu, Yuan Xie, John Jack Sampson, and Vijaykrishnan Narayanan. 2016. Nonvolatile Processor Architectures: Efficient, Reliable Progress with Unstable Power. *IEEE Micro* 36, 3 (May–Jun. 2016).

[34] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution without Checkpoints. In *Proc. OOPSLA.* ACM, Vancouver, BC, Canada.

[35] University of Washington. 2014. WISP 5.0 Wiki. http://wisp5.wikispaces.com. Last accessed: Mar. 30, 2018.

[36] Daniel Pickem, Myron Lee, and Magnus Egerstedt. 2015. The GRITSBot in its Natural Habitat - A Multi-robot Testbed. In *Proc. ICRA.* IEEE, Seattle, WA, USA.

[37] Joseph Polastre, Robert Szewczyk, Alan Mainwaring, David Culler, and John Anderson. 2004. Analysis of Wireless Sensor Networks for Habitat Monitoring. In *Wireless Sensor Networks*, C. S. Raghavendra, Krishna M. Sivalingam, and Taieb Znati (Eds.). Springer, Boston, MA, USA.

[38] R. Venkatesha Prasad, Shruti Devasenapathy, Vijay S. Rao, and Javad Vazifehdan. 2014. Reincarnation in the Ambiance: Devices and Networks with Energy Harvesting. *IEEE Commun. Surveys Tuts.* 11, 1 (First Quarter 2014).

[39] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. In *Proc. ASPLOS.* ACM, Newport Beach, CA, USA.

[40] Michael Rubenstein, Christian Ahler, and Radhika Nagpal. 2012. Kilobot: A Low Cost Scalable Robot System for Collective Behaviors. In *Proc. ICRA.* IEEE, Saint Paul, MN, USA.

[41] Saleae. 2017. Saleae Logic Pro 16 Analyzer. http://downloads.saleae.com/specs/Logic+Pro+16+Data+Sheet.pdf. Last accessed: Mar. 30, 2018.

[42] Koen Schaper. 2017. *Transiently-powered Battery-free Robot.* Master Thesis. Delft University of Technology, Delft, The Netherlands.

[43] Faisal Karim Shaikh, Sherali Zeadally, and Ernesto Exposito. 2017. Enabling Technologies for Green Internet of Things. *IEEE Syst. J.* 11, 2 (June 2017).

[44] Joshua R. Smith. 2013. *Wirelessly Powered Sensor Networks and Computational RFID.* Springer Verlag, New York, NY, USA.

[45] Joshua R. Smith, Alanson P. Sample, Pauline S. Powledge, Sumit Roy, and Alexander Mamishev. 2006. A Wirelessly-Powered Platform for Sensing and Computation. In *Proc. UbiComp.* ACM, Orange County, CA, USA.

[46] Tolga Soyata, Lucian Copeland, and Wendi Heinzelman. 2016. RF Energy Harvesting for Embedded Systems: A Survey of Tradeoffs and Methodology. *IEEE Circuits Syst. Mag.* 16, 1 (First Quarter 2016).

[47] Sparkfun. 2009. Analog Devices ADXL345 Breakout Board. https://www.sparkfun.com/datasheets/Sensors/Accelerometer/ADXL345.pdf. Last accessed: Apr. 1, 2018.

[48] Ivan Stoianov, Lama Nachman, Sam Madden, and Timur Tokmouline. 2007. PIPENET: A Sireless Sensor Network for Pipeline Monitoring. In *Proc. IPSN.* ACM/IEEE, Cambridge, MA, USA.

[49] Fang Su, Yongpan Liu, Yiqun Wang, and Huazhong Yang. 2017. A Ferroelectric Nonvolatile Processor with 46μs System-Level Wake-up Time and 14μs Sleep Time for Energy Harvesting Applications. *IEEE Trans. Circuits Syst. I* 64, 3 (March 2017).

[50] Texas Instruments, Inc. 2014. FRAM FAQs. http://www.ti.com/lit/ml/slat151/slat151.pdf. Last accessed: Mar. 30, 2018.

[51] Texas Instruments Inc. 2017. MSP430FR59xx Mixed-Signal Microcontrollers (Rev. F). http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf. Last accessed: Aug. 30, 2018.

[52] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proc. OSDI.* ACM, Savannah, GA, USA.