Intermittent Computing: Challenges and Opportunities*

Brandon Lucia[1], Vignesh Balaji[2], Alexei Colin[3], Kiwan Maeng[4], and Emily Ruppel[5]

1 **Carnegie Mellon University, Department of ECE, Pittsburgh, PA, USA**
2 **Carnegie Mellon University, Department of ECE, Pittsburgh, PA, USA**
3 **Carnegie Mellon University, Department of ECE, Pittsburgh, PA, USA**
4 **Carnegie Mellon University, Department of ECE, Pittsburgh, PA, USA**
5 **Carnegie Mellon University, Department of ECE, Pittsburgh, PA, USA**

Abstract

The maturation of energy-harvesting technology and ultra-low-power computer systems has led to the advent of intermittently-powered, batteryless devices that operate entirely using energy extracted from their environment. Intermittently operating devices present a rich vein of programming languages research challenges and the purpose of this paper is to illustrate these challenges to the PL research community. To provide depth, this paper includes a survey of the hardware and software design space of intermittent computing platforms. On the foundation of these research challenges and the state of the art in intermittent hardware and software, this paper describes several future PL research directions, emphasizing a connection between intermittence, distributed computing, energy-aware programming and compilation, and approximate computing. We illustrate these connections with a discussion of our ongoing work on programming for intermittence, and on building and simulating intermittent distributed systems.

1998 ACM Subject Classification C.0 Hardware/Software Interfaces, D.4.5 Reliability

Keywords and phrases Intermittent computing, Energy-harvesting devices

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.8

1 Introduction

Recent years have seen a shift toward increasingly small and low-power computing devices across a variety of application domains, including IoT devices [16], wearable, implantable, and ingestible medical sensors [31, 19], infrastructure monitors [28], and small satellites [46, 2]. Advances in energy-harvesting technology [34, 26, 18, 27] have enabled applications that run entirely using energy harvested from their environment without the restriction of tethered power or maintenance requirements of a battery. These devices harvest and buffer energy as it is available and operate when sufficient energy is banked. Operation in these devices is *intermittent* because energy is not always available to harvest and, even when energy is available, buffering enough energy to do a useful amount of work takes time. The hardware of an intermittently operating device can include general purpose computing components, such as a CPU or microcontroller (MCU), an array of sensors, and one or more radios for communication. Typical devices contain *volatile* memory that loses its state on a power

* This work was funded by National Science Foundation grant CNS-1526342 and a gift from Disney Research Pittsburgh.

© Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel;
licensed under Creative Commons License CC-BY
2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 8; pp. 8:1–8:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

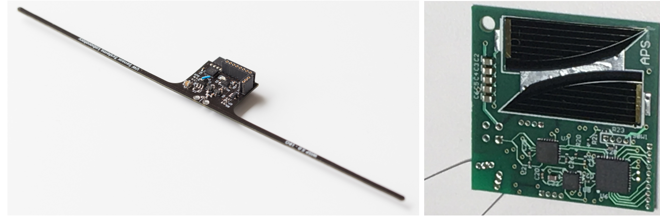

Figure 1 Two energy-harvesting devices. RF-powered WISP Platform [34] (left) and our solar-powered EDBsat single-board satellite (right).

failure, such as SRAM and DRAM, and *non-volatile* memory that retains its state on a power failure, such as Flash and FRAM [42]. Figure 1 shows two energy-harvesting platforms.

Programmers of today's intermittently operating devices use a typical, C-like embedded programming abstraction despite a number of important differences between the intermittent execution model and a typical embedded execution model. In particular, software running on an intermittently operating device executes until energy is depleted and the device browns out. When energy is again available, software resumes execution from some point in the history of its execution, i.e., the beginning of `main()` or a checkpoint [33, 17, 24, 3, 4, 23, 9, 43]. The key distinction between a conventional execution and intermittent execution is that a conventionally executing program is assumed to run to completion but an intermittent execution must *span* power failures. To tolerate power failures that occur hundreds of times per second, multiple layers of the system require an intermittence-aware design, including languages, runtimes, and application logic.

This paper provides a survey of current research challenges in intermittent computing and a vision for future intermittence research in the PL and systems community. To achieve that goal, Section 2 describes several PL and systems challenges brought about by intermittent computing. Section 3 describes the design space of intermittent computing devices, focusing on hardware and software characteristics that are likely to affect future research. A goal of this work is to show how intermittent computing brings together other areas of PL and systems research, including, distributed computing and concurrency, energy-aware programming and compilation, and approximate computing. Section 4 describes several programming languages research directions that address intermittence. This paper is intended to inspire and equip PL researchers to begin using and researching intermittent computing systems.

2 Intermittent Computing Challenges

Intermittent operation is an impediment to programming today's intermittently operating devices. The difficulty stems from the fact that an intermittent execution proceeds in bursts when energy is available and includes periods of inactivity when energy is not available. This succession of active and inactive periods is illustrated in Figure 2. Intermittent execution makes control-flow unpredictable, compromises an application's forward progress, leaves memory inconsistent, leaves a device inconsistent with its environment, and complicates device-to-device communication. We discuss these problems briefly and cite work exploring them in depth.

Control-flow. To an executing program, resuming after a power failure is a discontinuity in control-flow that is not explicitly expressed in source code. Programmers of intermittent devices must deal with implicit control flows to potentially unpredictable points in an

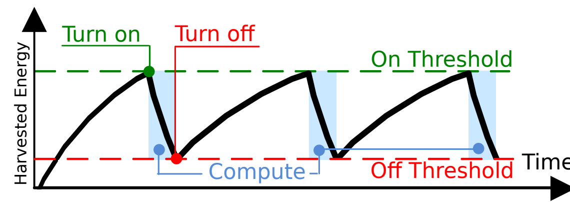

Figure 2 Intermittent execution. An intermittently-powered device executes its program in bursts as energy is available.

execution's history, such as a recent checkpoint [33, 24, 17, 23, 43] or the beginning of a task [6, 9].

Some platforms (like the WISP [34]), always begin executing with the same quantum of energy available and (in effect) do not recharge during execution[1]. If a restarted computation cannot successfully reach a checkpoint or complete a computational task using the start-time quantum of energy, then the system will unsuccessfully attempt to execute the same span of code repeatedly, preventing the program from making meaningful progress. This "Sysiphean" computation problem [33, 6] is particularly problematic in energy-starved environments. Guaranteeing forward progress in intermittent execution models is an important, unsolved challenge, especially for systems with explicit, statically-demarcated checkpoints and tasks.

Data consistency. Recent work [23] demonstrated that a naive combination of checkpointing and direct access to non-volatile memory in an intermittent device [33, 24, 17] can lead to memory inconsistencies. The key problem is that volatile state, such as the device's registers, stack, and global variables, are erased or rolled back to a previous state (e.g., a checkpoint) when the power fails. In contrast, the byte-addressable, non-volatile storage retains its values and those values may be inconsistent with the rolled-back volatile state. Keeping the contents of both types of memory correct requires careful, expert-level programming or system support [23, 9, 43] to ensure that non-volatile values are kept consistent with frequently erased or reverted volatile values. Due to the limited supply of energy, the time [23, 43, 33, 17] and space [9] cost of managing memory is a key factor that determines the resources available to the application.

Environmental consistency. Like other embedded systems, intermittently operating devices receive inputs from the outside world via sensors. Sensed data become stale and unusable if they are buffered across a long time period without harvestable energy. Sensor accesses intended to be atomic with one another may be split by a power failure, causing their resultant data to be inconsistent with the device's real environment. Prior work on system support for intermittent task atomicity [9, 23] avoids this problem by letting the programmer define tasks containing I/O that should re-execute atomically. Other work [11, 15] explicitly tracks time to avoid staleness issues.

Concurrency. Sensors, peripheral devices, and collections of MCUs may all operate concurrently as a single, intermittent device. As is common in embedded systems, concurrency with sensors is largely interrupt-driven. For example, an MCU may request data from a

[1] The recharge rate is non-zero, but negligible compared to the energy discharged during an execution period.

sensor, and a sensor may buffer and reply with data. Similarly, two MCUs may exchange and compute on data in parallel. Concurrency control in such scenarios is complicated by intermittent interruptions. If control-flow in one or more concurrent execution threads is re-directed to an earlier point by an intermittent power failure, how should the system manage the visibility in each thread of values produced by both threads? We are unaware of existing work that specifically addresses concurrency and intermittence together. Most existing intermittence research [9, 23, 43, 3, 4] assumes a single control thread and does not define the behavior of operations that are concurrent with intermittent control threads.

Compounding the state management problem, the timing, precision, and frequency of concurrent components are influenced by the availability of buffered and harvestable energy. Energy-dependent concurrency control becomes especially complex in a device with *federated energy storage* [14]. In a federated system, components charge and discharge their own storage elements independently. As a result, each component becomes an *intermittent resource* available at different times, depending on its energy supply and capacity. The software must synchronize access to the intermittent resources, not only in the relative logical time, but also in real physical time.

Distributed intermittent devices. Distributed collections of intermittently operating devices must interact with one another via radio. Most work has focused on physical-layer mechanisms to enable devices to communicate [22, 5]. We observe several reasons why coordinating distributed intermittent devices is difficult, beyond the issues at the physical layer. The cost of communicating is high: a fixed-length period of communication costs an order of magnitude more energy than a similar period of computation [20, 12]. Deciding when to incur the high cost of communication, and how much data to transmit or receive is a delicate trade-off of energy for precision or functionality. Synchronizing a collection of intermittently operating devices is an unsolved problem and a communication between unsynchronized, intermittent end-points is only successful if both are coincidentally operating for a long enough time, at the same time. A distributed intermittent system must gracefully allow communication to fail very frequently.

3 The Intermittent System Design Space

The challenges in Section 2 are a consequence of the hardware and software design of the energy-harvesting device. Exploring the design space is necessary to understand why programming intermittent devices is challenging and to inform future PL research on intermittent systems. The design space of intermittent devices is rich with inter-dependent hardware and software components that dictate behavior and applicability. Our discussion focuses on three design parameters: (1) energy harvesting and storage; (2) memory and execution models; and (3) software development toolchain.

3.1 Energy Harvesting and Storage.

The behavior of a program running on an energy-harvesting device depends on a number of factors: its energy-harvesting modality, energy storage mechanism, and power-on/power-off behavior.

Energy Harvesting. Energy harvesters vary widely from device to device. Solar panels deliver power proportional to their illuminated area. Solar harvesters with a compact form factor (cm^2) typically generate tens of μW to tens of mW of power. A device powered

by RF energy depends on the availability of radio waves in a specific frequency range. Harvestable RF power varies from nW (ambient sources [22]) to μW (RFID-readers and power transmitters [34, 37, 30]). Mechanical harvesters range from nW-scale buttons [27] and sliders [18] to multi-Watt self-powered knobs [44].

In the simplest design the harvester output is connected directly to the load (i.e. MCU, sensors). This design is only appropriate if the harvester's current output matches the load's current draw (e.g., ~1 mA for a 4 MHz MSP430) and its voltage output is acceptable to the load (e.g., 1.8-3.3 V). In such a design, the duration of an intermittent execution interval equals the duration of the incoming energy burst. This design is rarely applicable, because the harvester rarely matches the current and voltage of the load. Instead, the load is usually *decoupled* from the harvester by an energy buffer, e.g. a capacitor. Hardware or software controls charging and discharging of the storage element. As a result, intermittent execution intervals are regularly periodic even if input energy is erratic.

Energy Storage. The energy buffering mechanism affects system and software behavior. Power systems that decouple the load and harvester operate in repeated charge-discharge cycles. First, the device accumulates energy, while consuming a negligible amount. With sufficient energy stored, the device begins to operate until the energy is depleted. The energy storage *capacity*, fixed at design time, determines the maximum amount of computation that is possible without a power failure.

The energy storage mechanism is a key design parameter because it dictates a device's physical size. Designers may be *volumetrically* constrained by an application (e.g., in-body devices [19]), limiting energy capacity and capability. Capacitors are cheap and small but not energy-dense. Super-capacitors are an order of magnitude more dense, but moderately larger and more costly. An energy harvester can also charge a small battery and, unlike a capacitor that appreciably leaks energy, the battery will leak slowly, permitting operation over long periods without harvestable energy. Batteries, however, have drawbacks. Conventional batteries (e.g., coin-cells, AA) are heavy and fragile. Thin-film batteries are light, but inapplicable in some harsh environments; e.g., suffering permanent failures in low-temperatures space applications [46]. Batteries wear-out, reducing efficiency and requiring replacement, which can be labor intensive or impossible in adversarial environments. Battery chemistry makes assessing a battery's remaining charge difficult. Voltage is a poor indicator of a battery's stored energy because capacity varies with wear, temperature, and workload. In contrast, a capacitor's voltage reflects its energy content, allowing hardware or software to read the voltage and react to energy events, such as a full charge or an impending power failure [33, 3, 8].

Energy Distribution. A device's pattern of intermittent execution activity depends on when energy accumulates and when it is consumed. Charge/discharge behavior can be implicit in the hardware, or controlled explicitly by hardware or software logic. Absent energy-distribution logic, a device will operate whenever its energy buffer's voltage is within operating range. However, relying on implicit on/off behavior is impractical because it leads to *thrashing*: the storage element never has time to accumulate a significant amount of energy before being drained. Instead, explicit on/off logic accumulates charge without consuming energy up to a threshold energy level. With a capacitor as the storage medium, the energy threshold level translates to a threshold capacitor voltage.

Two quantitative design parameters that lead to qualitative differences in system behavior are the turn-on and turn-off voltage thresholds. In some devices (e.g., WISP5 [45],

Powercast [30]) the turn-on threshold is fixed in hardware to the maximum operating voltage. Setting the turn-on threshold to the maximum voltage makes the device turn on with maximum energy stored. Other systems (e.g., WISP4 [34]) set the turn-on threshold to the minimum operating voltage. Setting the turn-on threshold to the minimum voltage allows software to control when the device starts operating. The software may put the processor to sleep and periodically check the accumulated energy until the desired level is reached. With this design, the system can spend only as much time charging as necessary for a particular task. Symmetrically, the turn-off threshold may be fixed in hardware or managed by software. By default, the turn-off threshold is the minimum operating voltage of the device, but a deliberate design may turn off the device at a higher voltage. None of the above designs is unconditionally superior to all others. Threshold settings qualitatively change the turn on/turn off behavior and determine the intermittent execution intervals experienced by the software.

Systems whose load consists of multiple components with separate power rails (e.g., discrete sensor or radio ICs, multiple processors), open a design choice of *federating* [14] the energy storage into multiple isolated banks. In contrast to a shared energy buffer, a federation of per-component buffers de-couples unrelated hardware components letting each fail independently. Federated energy buffers do not necessarily charge in synchrony: one component may accumulate sufficient energy to turn on at a time that is different from and unpredictable to other components. Software on a federated platform faces the inter-component concurrency challenge described in Section 2.

3.2 Memory system and execution model

The effect of a power failure on a system and the system's resumption behavior follows from the memory system and the mechanism for preserving progress in the execution model.

Memory system. The most general model of a device's hardware includes both volatile memory (e.g., SRAM and DRAM) and non-volatile memory (e.g., Flash, FRAM). On some architectures [43] all main memory is non-volatile, leaving MCU-internal state (e.g., registers) volatile. At the extreme of the design space are architectures where all memory and internal processor state (including registers and microarchitectural structures) is non-volatile [21]. Converting volatile structures to non-volatile may eliminate some of the memory inconsistency issues described in Section 2. However, fully non-volatile architectures and main memories have two drawbacks. First, efficiency suffers, because the relatively low-latency, low-energy volatile memory accesses become relatively high-latency, high-energy non-volatile memory accesses. We measured and compared the energy cost of a volatile SRAM access to a non-volatile FRAM access on a TI MSP430FR5969 MCU and found that the FRAM access consumed 2-3x more energy on average. SRAM, with an access latency around 10ns is faster than today's FRAM, which has latency around 50-80ns [39]; however, with the often low clock frequencies of low-power MCUs (around 8MHz), SRAM and FRAM accesses take a single cycle [42]. Furthermore, a fully non-volatile architecture is at best a partial solution to the problem of preserving progress across power failures, because some state is fundamentally not "latchable" and must be re-initialized by executing code. For example, a MEMS sensor must perform an initialization routine before it can be sampled.

Looking forward, it is likely that intermittently operating device designs will include deeper, more complex memory hierarchies with a mixture of cache layers and non-volatility. We anticipate that it will be important to adapt techniques for managing non-volatility [47, 7, 29] to work in the energy, time, and memory constrained intermittent environment.

In particular, in the intermittent execution model, the recovery path is not exceptional but common and must be efficient, in contrast to traditional applications of non-volatile memory on servers or workstations.

Execution Model. The precise execution model of an intermittent device depends on how software and hardware preserve progress and program state. Most intermittent systems run “bare metal” programs, bypassing any operating system support to avoid unnecessary time or energy cost. In typical “bare-metal“ embedded systems, without system support for intermittent operation, a power failure erases volatile values and retains non-volatile ones. Checkpoint-based models [43, 33, 17, 24] preserve register, stack, and global variable values, including the program counter, and restore them after a power failure. As Section 2 discusses, checkpoints alone leave memory inconsistent, necessitating multi-versioning models [23, 9, 43] that also preserve and restore parts of non-volatile memory.

Without system support, after a power failure control flows to the program’s entry point (i.e., `main()`). In checkpointing models [33, 17, 24, 43] execution resumes from a compiler-inserted or dynamically-decided checkpoint. In a task-based model [23, 9], the programmer explicitly deconstructs the program into tasks that execute atomically (and idempotently). After a power failure, execution resumes from the beginning of the most recently executed, statically-demarcated task boundary. Alternatively, some systems propose to stop the execution when power failure is deemed to be imminent and save a checkpoint then [3, 41, 4]. Without a progress latching mechanism, the application is limited to short, uninterruptible “one-shot” tasks [6].

Models with statically defined tasks require some extra programmer effort, compared to dynamic checkpoints. The advantage of a static task system is that the programmer has more control over which regions of the code are atomic and idempotent. Control over atomicity and idempotence is often important in code with application level requirements on I/O operations (e.g., a temperature and pressure sensor must be read atomically, without an intervening delay due to a power failure).

A system’s state and progress preservation strategy, as well as the way the programmer expresses atomicity and idempotence constraints originate the control flow, data consistency, and environmental consistency challenges described in Section 2.

3.3 Development Environment

The effect of the power system on the behavior of software on intermittent devices complicates its development, testing, and debugging. Tools designed for continuously-powered systems do not help find bugs that manifest only under particular power failure timings or test across energy environments. Consequently, recent work proposed targeted tools for monitoring, debugging, and profiling [8], energy tracing [13], and transferring code onto intermittent devices [40, 1].

Our work on EDB [8], the Energy-interference-free Debugger, provided the first support for passively monitoring and interactively debugging intermittently-operating devices with assertions, breakpoints, and watchpoints. Debuggers available before EDB require the device to be powered continuously, making it difficult to observe, diagnose, and fix system behavior that only manifests when running on harvested energy. Working from this motivation, EDB uses a combination of hardware support and a package of co-designed software libraries to provide support for important debugging tasks during intermittent executions on energy-harvesting devices. EDB’s key source of novelty is to avoid “energy-interference”, which is any exchange of energy between the debugger and the target that could perturb the intermittent

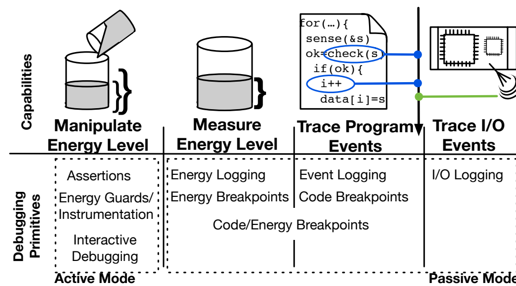

Figure 3 EDB's capabilities and features [8].

execution, changing its behavior. EDB supports passive monitoring tasks, such as tracing the device's energy level, tracing manually inserted code markers, and tracing I/O operations (such as RFID Rx/Tx). EDB also supports "active" tasks, including interactive, breakpoint debugging, high-energy-cost instrumentation, and invasive data invariant checking. The key to supporting energy-hungry "active" tasks is to *compensate* for energy consumed. Before an active task, EDB checks the device's energy level. After completing an energy-hungry active task, EDB restores the device's energy level to its level before the debugging task. With its support for passive and active debugging and tracing, EDB is the first debugger to bring necessary, basic debugging functionality to the intermittent computing domain. EDB is available for release at `http://intermittent.systems` and Figure 3 (reproduced from [8]) shows an overview of EDB's main capabilities.

Prior to EDB, Sympathy [32] addressed the challenges of debugging networks of sensor nodes, although Sympathy did not address intermittent operation. Ekho [13] addressed the lack of tools for measuring and reproducing energy conditions that vary over time. Energy availability at a given time can be represented by a current-voltage (I-V) curve. Ekho records a time-series of I-V curves in the field and replays them on-demand in the lab for reproducing issues and examining the behavior across energy environments. To simplify deployment, recent work [40, 1] developed a mechanism to reliably and efficiently transfer code (or other data) to a device using the RFID protocol, while the device is intermittently-powered.

3.4 Programming Support

Few real programming language design efforts have targeted intermittent and energy-harvesting devices. Eon [36] was the first language for an energy-harvesting system. Eon did not explicitly target intermittence, but instead tried to gracefully degrade application behavior with scarce energy. Eon gives a task a priority and tries to execute high priority tasks more often, subject to energy constraints.

Our work on Chain [9] is the first language designed explicitly to deal with intermittence, through a task-based control-flow abstraction and a channel-based abstraction for non-volatile memory that maintains consistency via static multi-versioning. The key idea in Chain is to decompose the program into a collection of tasks, which are annotated functions, and to explicitly describe the flow of execution from one task to the next. Chain guarantees that, even in the presence of power failures, tasks execute atomically. Tasks can exchange data consistently using *channels*, which are Chain's abstraction of non-volatile memory. A task may only ever read from or write to a normal channel, but not both.

The "channel access exclusion" property of Chain's channels ensure that regardless of the presence or timing of power failures, a task's inputs are always available (in its input channels)

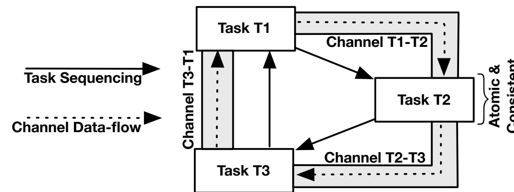

Figure 4 A schematic of a Chain program [9]. The program has three tasks that execute in sequence and pass data to one another via channels.

and its outputs always have a place (in its output channels). Statically multi-versioning data in channels allows a Chain implementation to arbitrarily re-start a task from its entry point with a consistent memory state. Idempotently re-executing a task until an execution attempt eventually completes makes the effects of a Chain task atomic, when a Chain task finally completes. Moreover, Chain eliminates the need to save and restore any volatile state because all volatile variables are required to be task-local, and initialized inside a task. Chain's unique memory abstraction, task-based control-flow, and freedom from costly checkpointing mechanisms leads to a substantial performance improvement, compared to typical volatile data checkpointing systems [33], and even non-volatile data versioning mechanisms [23]. Figure 4 (reproduced from [9]) shows a schematic view of a Chain program. A Chain reference implementation is available for researchers at `http://intermittent.systems`, including support libraries and example code to help get started building Chain applications for the WISP [34] or other intermittent devices.

The development of languages, debuggers, program analyses, and testing tools, for intermittent systems is an area of PL research open for contributions from the community. The impact of this research is widespread use of battery-free, devices across a variety of application domains.

4 Future Research Opportunities in Intermittent Computing

Intermittent computing is a promising, emerging PL research area. Next we outline our work at the intersection of energy-awareness, distributed computing, and approximation in intermittent systems.

4.1 Programming Intermittent Systems

Despite building momentum, existing approaches to programming intermittent devices have several key drawbacks: (1) increased programmer effort [23, 9] to define tasks; (2) no programmer guidance or optimization for sizing tasks [23, 9, 43]; (3) run time [43, 23] and memory [9] overheads; (4) unsound inference of application-level properties (e.g., I/O atomicity) [43]; (5) assumptions about memory volatility [43]. These limitations of prior work motivate further study. Our ongoing work aims to address the above challenges with new programming abstractions that minimize overheads, reduce programmer burden, while retaining programmer control over atomicity.

We are developing a new task-based programming model, based on Chain [9], that fundamentally departs from Chain's static multi-versioning approach. Chain creates a copy of each variable for each pair of tasks that communicate through that variable, which introduces time and space overhead as well as programmer burden. Our new efforts avoid multi-versioning using novel compiler analyses, dynamic multi-versioning, and a simple,

efficient commit mechanism to keep data consistent. The key insight in our new work is that it is possible to *privatize* a copy of data to a task, allowing safe access to copies that can be stored in non-volatile memory, or in energy-efficient volatile buffers. Our initial experiments with applications from Chain [9] including compressive sensor logging and data filtering suggest that eliminating Chain's static versioning and channel management overheads yields up to 4x decrease in memory consumption and a 1.5x-7x performance improvement.

Energy-aware programming and compilation. With our new, task-based programming model efforts, we are building energy-aware compiler support [10] to help the programmer express tasks that are optimized to the underlying hardware. Assuming the common, "execute with maximum charge" hardware model [34] described in Section 3, our compiler statistically assesses whether a task's energy cost exceeds the maximum charge level of the device. Such a task would never complete and the compiler can automatically sub-divide the task, or guide the programmer in sub-dividing the task. Our work represents only a point in the energy-aware programming and compilation design space; intermittent systems warrant further exploration in this area.

Approximate execution models. Approximate execution models offer an alternative approach to handling power failure. In task-based systems (e.g., Chain[9]), after a power failure, previously executed instructions are re-executed. Re-execution burns time and energy in order to complete the task and produce a result. In an approximate execution model, accuracy can be traded off instead of spending time and energy on re-execution, by *abandoning* the interrupted task. Then, the challenge is to decompose the application into tasks and prioritize the tasks such that the completion of any subset of tasks produces a meaningful (approximate) result. For example, in an approximate motion detector, decomposed into tasks spatially, only some regions of the image would be searched under poor energy conditions. An alternative approximate execution model might reduce the cost of multi-versioning state by accepting inconsistency in some of the data values.

4.2 Distributed Intermittent Systems

Building distributed systems of intermittent devices enables new battery-less applications, e.g., sensing and actuation systems, computer vision [25], and swarms of tiny satellites [46, 2]. Realizing this vision demands that the PL community develop programming and system foundations for distributed, intermittent systems. The difficulty of specifying a correct, efficient distributed, intermittent system is compounded by the absence of development tools, specification languages, memory abstractions, and execution models. Our ongoing work focuses on intermittent distributed shared memory abstractions and simulator-based developer support.

Distributed, intermittent shared memory. We are building the first energy- and intermittence-aware, distributed shared-memory system. The key challenge, noted in Section 2, is that a pair of intermittent devices can only interact when both are active. Our intermittent distributed shared memory (iDSM) has a flat address space, with data spanned and replicated across the nodes in a system (similar to continuously-powered DSMs) [38]. Our iDSM's main contribution is an energy- and intermittence-aware memory consistency mechanism.

Maintaining iDSM consistency is difficult because both nodes involved in a request for data are rarely simultaneously powered. We address the problem by tracking request success and failure and adapting nodes' memory request behavior based on the likelihood of a

request's success. If a node's request for another node's copy of a shared page frequently fails, we throttle the rate of requests between those two nodes. Instead, when either node makes a request, it prioritizes a different node with a higher historical success rate. This communication policy is energy-aware and affects memory consistency. The energy-awareness stems from the energy environment's influence over nodes' communication success rate. The policy determines memory consistency because preferentially non-communicating nodes will share updates less often, leaving data inconsistent for longer. Space- and time-dependent energy-availability requires the system to distribute data replicas to avoid "stranding" data on inaccessible nodes. iDSM research will benefit from PL contributions on new data consistency and replication policies, latency-tolerant synchronization mechanisms, and domain-specific language support for constraining how intermittent nodes interact.

Approximate, distributed, intermittent systems. Intermittent, distributed systems can leverage approximate memory consistency to improve performance and ensure progress. Assuming an iDSM with mutex locks, approximate locks with timeout-based release behavior may help prevent deadlocks when a node holding a lock fails. The cost of deadlock-freedom is the need to handle the effects of broken atomicity and potential inconsistency, which can lead to errors or a crash. Such a synchronization mechanism might integrate with type support [35] to ensure that critical program values are never corrupted, even at a cost in performance or progress.

Simulating distributed, intermittent systems. We built a flexible simulation framework for distributed, intermittent systems to help study the performance impact of energy-awareness and approximation on our iDSM without the high engineering cost of a real hardware setup. Our simulator consumes logged power traces (similar to Ekho [13]) to accurately model intermittent power cycling in a simulated collection of distributed nodes. Our inter-node communication model is flexible and currently models ambient backscatter broadcasts within a small network [22]. The simulator models the iDSM memory space and private, per-node scratchpad memory spaces, both of which are accessible through a simulator-defined interface. A simulated node queues local and iDSM operations and attempts to dequeue and execute operations on each reboot. iDSM operations traverse the network to the owner of requested data, succeeding only when the requester and data owner are powered simultaneously. Our simulator provides key insights into the communication and consistency characteristics of our iDSM.

5 Conclusion

Intermittent, energy-harvesting computing devices promise important, future applications, and a variety of future PL and computer systems research challenges. This paper provided a survey of the challenges and the design space of intermittent devices, framing a vision for future PL research into intermittent computing.

Acknowledgments. We thank the anonymous reviewers for their feedback and suggestions to improve our manuscript.

References

1 Henko Aantjes, Amjad Y. Majid, Przemysław Pawełczak, Jethro Tan, Aaron Parks, and Joshua R. Smith. Fast Downstream to Many (Computational) RFIDs. In *IEEE INFOCOM*

2017 – The 36th Annual IEEE International Conference on Computer Communications, May 2017.

2 Justin A. Atchison and Mason Peck. A millimeter-scale lorentz propelled spacecraft. In *AIAA Guidance, Navigation and Control Conference*, August 2007. doi:10.2514/6.2007-6847.

3 Domenico Balsama, Alex Weddell, Geoff Merrettt, Bashir Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded System Letters*, 7(1):15–18, March 2015. doi:10.1109/LES.2014.2371494.

4 D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. 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):1968–1980, 2016. doi:10.1109/TCAD.2016.2547919.

5 Dinesh Bharadia, Kiran Raj Joshi, Manikanta Kotaru, and Sachin Katti. BackFi: High throughput wifi backscatter. In *SIGCOMM'15*, pages 283–296, October 2015. doi:10.1145/2785956.2787490.

6 Michael Buettner, Ben Greenstein, and David Wetherall. Dewdrop: An energy-aware task scheduler for computational RFID. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2011.

7 Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *16th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, March 2015.

8 Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. In *21st ACM Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 577–589, April 2016.

9 Alexei Colin and Brandon Lucia. Chain: tasks and channels for reliable intermittent programs. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 514–530, October 2016.

10 Alexei Colin, Preeti Murthy, and Brandon Lucia. Cleancut: Static task boundary placement for intermittent programs. In *Workshop on Hilariously Low-Power Computing*, April 2016.

11 Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. TARDiS: A branch-and-merge approach to weak consistency. In *International Conference on Management of Data*, June 2016. doi:10.1145/2882903.2882951.

12 G. de Meulenaer, F. Gosset, F. X. Standaert, and O. Pereira. On the energy cost of communication and cryptography in wireless sensor networks. In *2008 IEEE International Conference on Wireless and Mobile Computing, Networking and Communications*, pages 580–585, Oct 2008. doi:10.1109/WiMob.2008.16.

13 Josiah Hester, Timothy Scott, and Jacob Sorber. Ekho: realistic and repeatable experimentation for tiny energy-harvesting sensors. In *12th ACM Conference on Embedded Networked Sensor Systems (SenSys'14)*, pages 330–331, November 2014. doi:10.1145/2668332.2668382.

14 Josiah Hester, Lanny Sitanayah, and Jacob Sorber. Demo: A hardware platform for separating energy concerns in tiny, intermittently-powered sensors. In *13th ACM Conference on Embedded Networked Sensor Systems (SenSys'15)*, pages 447–448, November 2015. doi:10.1145/2809695.2817847.

15 Josiah Hester, Kevin Storer, Jacob Sorber, and Lanny Sitanayah. Towards a language and runtime for intermittently powered devices. In *Workshop on Hilariously Low-Power Computing*, April 2016.

16 International Telecommunication Union. Overview of the internet of things. `http://handle.itu.int/11.1002/1000/11559`, June 2012.

17 H. Jayakumar, A. Raha, and V. Raghunathan. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *Int'l Conf. on VLSI Design and Int'l Conf. on Embedded Systems*, January 2014. URL: `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6733152`.

18 Mustafa Karagozler, Ivan Poupyrev, Gary Fedder, and Yuri Suzuki. Paper Generators: Harvesting energy from touching rubbing and sliding. In *ACM Symposium on User Interface Software and Technology (UIST)*, October 2013. `doi:10.1145/2501988.2502054`.

19 Yoonmyung Lee, Gyouho Kim, Suyoung Bang, Yejoong Kim, Inhee Lee, Prabal Dutta, Dennis Sylvester, and David Blaauw. A modular 1mm3 die-stacked sensing platform with optical communications and multi-modal energy harvesting. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 402–403, February 2012.

20 Ting Liu, Christopher Sadler, Pei Zhang, and Margaret Martonosi. ZebraNet. In *2nd Intl. Conference on Mobile Systems, Applications and Services (MobiSys'04)*, pages 256–269, June 2004. `doi:10.1145/990064.990095`.

21 Ting Liu, Christopher Sadler, Pei Zhang, and Margaret Martonosi. An energy-efficient nonvolatile microprocessor considering software-hardware interaction for energy harvesting applications. In *Intl. Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, April 2016. `doi:10.1109/VLSI-DAT.2016.7482577`.

22 Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua Smith. Ambient backscatter: wireless communication out of thin air. In *SIGCOMM'13*, pages 39–50, October 2013. `doi:10.1145/2534169.2486015`.

23 Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 575–585, June 2015.

24 A. Mirhoseini, E. M. Songhori, and F. Koushanfar. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs. In *IEEE Pervasive Computing and Communication Conference (PerCom)*, March 2013. URL: `http://aceslab.org/sites/default/files/Idetic.pdf`.

25 Saman Naderiparizi, Zerina Kapetanovic, and Joshua R. Smith. Wispcam: An rf-powered smart camera for machine vision applications. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ENSsys'16, pages 19–22, 2016. `doi:10.1145/2996884.2996888`.

26 Joseph Paradiso. Systems for human-powered mobile computing. In *DAC*, July 2006. `doi:10.1145/1146909.1147074`.

27 Joseph Paradiso and Mark Feldmeier. A compact, wireless, self-powered pushbutton controller. In *Proceedings of the 3rd International Conference on Ubiquitous Computing (UbiComp'01)*, pages 299–304, September 2001.

28 Gyuhae Park, Tajana Rosing, Michael Todd, Charles Farrar, and William Hodgkiss. Energy harvesting for structural health monitoring sensor networks. *ASCE Journal of Infrastructure Systems*, 14(1):64–79, March 2008. `doi:10.1061/(ASCE)1076-0342(2008)14:1(64)#sthash.ULLx9D2h.dpuf`.

29 Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *ISCA*, June 2014.

30 Powercast Co. Development Kits – Wireless Power Solutions. `http://www.powercastco.com/products/development-kits/`. Visited July 30, 2014.

31 Proteus Digital Health. Proteus Discover. `http://proteus.com`, 2016.

32 Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. Sympathy for the sensor network debugger. In *Proceedings of the 3rd International*

Conference on Embedded Networked Sensor Systems, SenSys'05, pages 255–267, New York, NY, USA, 2005. ACM. doi:10.1145/1098918.1098946.

33 Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System support for long-running computation on RFID-scale devices. In *ASPLOS*, March 2011. URL: https://spqr.eecs.umich.edu/papers/ransford-mementos-asplos11.pdf.

34 Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. Design of an RFID-based battery-free programmable sensing platform. *IEEE Trans. on Instrumentation and Measurement*, 57(11):2608–2615, November 2008.

35 Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'11, 2011. doi:10.1145/1993498.1993518.

36 Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: A language and runtime system for perpetual systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys'07, pages 161–174, 2007. doi:10.1145/1322263.1322279.

37 Tolga Soyata, lucian Copeland, and Wendi Heinzelman. Rf energy harvesting for embedded systems: A survey of tradeoffs and methodology. *IEEE Circuits and Systems Magazine*, 16(1):22–57, February 2015. doi:http://dx.doi.org/10.1109/MCAS.2015.2510198.

38 Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM'01, pages 149–160, 2001. doi:10.1145/383059.383071.

39 D. Takashima, S. Shuto, I. Kunishima, H. Takenaka, Y. Oowaki, and S. Tanaka. A sub-40 ns random-access chain fram architecture with a 768 cell-plate-line drive. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 102–103, February 1999. doi:http://dx.doi./org/10.1109/ISSCC.1999.759147.

40 J. Tan, P. Pawełczak, A. Parks, and J. R. Smith. Wisent: Robust downstream communication and storage for computational rfids. In *IEEE INFOCOM 2016 – 35th Annual IEEE Int'l Conf. on Computer Communications*, pages 1–9, April 2016.

41 Texas Instruments Inc. Intelligent system state restoration after power failure with compute through power loss utility. http://www.ti.com/lit/ug/tidu885/tidu885.pdf, April 2015.

42 TI Inc. Overview for MSP430FRxx FRAM. http://ti.com/wolverine, 2014. Visited July 28, 2014.

43 Joel Van Der Woude and Mathew Hicks. Intermittent computation without hardware support or programmer intervention. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 17–32, November 2016.

44 Nicolas Villar and Steve Hodges. The Peppermill: A human-powered user interface device. In *Conference on Tangible, Embedded, and Embodied Interaction (TEI)*, January 2010. doi:10.1145/1709886.1709893.

45 WISP. http://wisp5.wikispaces.com/, 2016.

46 Zac Manchester. KickSat: a tiny open-sourced spacecraft. http://kicksat.github.io, 2016.

47 Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *MICRO*, December 2013. URL: http://www.cse.psu.edu/~juz138/files/150-zhao.pdf.