Automatically Enforcing Fresh and Consistent Inputs in Intermittent Systems

Milijana Surbatovich
Carnegie Mellon University
Pittsburgh, PA, USA
milijans@andrew.cmu.edu

Limin Jia
Carnegie Mellon University
Pittsburgh, PA, USA
liminjia@andrew.cmu.edu

Brandon Lucia
Carnegie Mellon University
Pittsburgh, PA, USA
blucia@andrew.cmu.edu

Abstract

Intermittently powered energy-harvesting devices enable new applications in inaccessible environments. Program executions must be robust to unpredictable power failures, introducing new challenges in programmability and correctness. One hard problem is that input operations have implicit constraints, embedded in the behavior of continuously powered executions, on *when* input values can be collected and used. This paper aims to develop a formal framework for enforcing these constraints. We identify two key properties—*freshness* (i.e., uses of inputs must satisfy the same time constraints as in continuous executions) and *temporal consistency* (i.e., the collection of a set of inputs must satisfy the same time constraints as in continuous executions). We formalize these properties and show that they can be enforced using *atomic* regions. We develop Ocelot, an LLVM-based analysis and transformation tool targeting Rust, to enforce these properties automatically. Ocelot provides the programmer with annotations to express these constraints and infers atomic region placement in a program to satisfy them. We then formalize Ocelot's design and show that Ocelot generates correct programs with little performance cost or code changes.

CCS Concepts: • **Computer systems organization** → **Embedded software**; • **Software and its engineering** → *Source code generation.*

Keywords: intermittent computing, energy harvesting, timeliness

ACM Reference Format:
Milijana Surbatovich, Limin Jia, and Brandon Lucia. 2021. Automatically Enforcing Fresh and Consistent Inputs in Intermittent Systems. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454081

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454081

1 Introduction

Energy-harvesting computer systems collect their operating energy from the environment, enabling autonomous operation over long periods of time without the need for battery maintenance. The key challenge of energy-harvesting systems is that power fails if there is insufficient energy to harvest. When an energy-harvesting system runs software, a power interruption may impede forward progress [23, 46], leave memory state inconsistent [31, 53], leave I/O state [5, 47] or data [51, 52] inconsistent with execution state, or leave I/O data inconsistent with a device's environment [20, 27].

Intermittent execution [31] of software enables sophisticated computation on energy-harvesting systems, leveraging tightly integrated non-volatile memory to retain state across failures. There are many approaches to address the software reliability challenges of intermittent computing. Most prior efforts focus primarily on problems related to progress and memory consistency. To save state, these techniques rely on in-code checkpointing (or tasks) [11, 31–33, 46, 48, 53, 57], or rely on a dynamic "just-in-time" (JIT) checkpointing mechanism [2, 3, 15, 23, 34, 35, 37, 56] that captures a snapshot of volatile state just before power fails.

Most intermittent computing happens on sensor-enabled devices destined for deeply-embedded deployment, where I/O drives the computation. Fortunately, recent work has begun investigating the unique challenges of I/O in intermittent systems. Some work ensures the basic, correct operation of peripherals and their drivers across power failures [5, 7, 30, 34, 36, 47], avoiding crashes, hangs, and driver state corruption. Other work addresses subtle interactions between I/O and checkpointing that lead to data corruption [51, 52]. These efforts enable correct basic operation of I/O devices in an intermittent execution. Operating in the real world, however, places correctness requirements on an intermittent system that go beyond ensuring that drivers and data remain uncorrupted.

Unlike a continuously-powered execution, an intermittent execution may violate implicit constraints on *when* inputs should be collected and used, due to the unpredictable time spent recharging after a power failure. An intermittent execution may use an input that is too old (i.e., stale) if the system checkpoints after the input is collected, but power fails before it is used. The need to avoid use of stale inputs is a *freshness* requirement. Some programs require *multiple*

input values to be sampled together (e.g., a pressure and a humidity reading) so that they come from a consistent point in time. The need to ensure that multiple inputs are sampled together is *temporal consistency*, which is violated by a checkpoint and power failure between these readings.

Freshness and temporal consistency belong to the broader category of *timeliness* requirements on inputs. Prior work explored timely intermittent execution [15, 20, 27, 45] but lacks formally specified correctness conditions. Existing approaches rely on the addition of hardware to track time during power failures and often require writing extra code to mitigate the misuse of expired inputs. Moreover, existing work focuses on freshness (e.g., using inputs before they expire) and does little to enforce temporal consistency.

In this work, we introduce formal definitions of freshness and temporal consistency and develop Ocelot, which automatically enforces specified timing constraints in intermittent systems without needing timekeeping hardware. Ocelot gives the programmer constructs to specify what timing properties matter for their program and enforces that specification by leveraging atomicity, generating programs that are correct-by-construction. Instead of enforcing programmer-specified expiration times, Ocelot enforces freshness and temporal consistency by ensuring that an intermittent execution does what some continuous execution would do; *the continuous execution is the specification of correct behaviour*. Ocelot asks the programmer to express freshness and temporal consistency requirements only and asks neither for real-time specification on the collection or use of inputs, nor for mitigation actions to handle expired inputs. Ocelot's atomic region inference algorithm then automatically inserts atomic regions that contain input-derived variable definitions and uses. If power fails during an atomic region, the region re-executes (idempotently) from the start. Outside of an atomic region, the system defaults to a baseline intermittent execution model (i.e., in our work, JIT checkpoints [3, 34]).

We formalize this notion of freshness and temporal consistency using a modeling language and investigate how to prove our design correct. We implement Ocelot for Rust using analyses built in LLVM [29]. We evaluate our implementation on a real energy-harvesting hardware platform [12] using a collection of applications taken from prior work, and a new tire safety monitoring application that we developed. Our results show that Ocelot effectively identifies atomic regions that enforce both freshness and temporal consistency. Ocelot imposes less than 10% runtime overhead compared to both JIT checkpoints and to atomic regions implementations from prior work. Ocelot demands less of the programmer, compared to two systems from prior work that address I/O timeliness [27, 34]. Most importantly, Ocelot provides a formally defined correctness criterion for collection and use of intermittent inputs, which no prior system provides.

To summarize, the main contributions are:

- We provide the first formal definitions for *freshness* and *temporal consistency* and show that atomicity is sufficient to enforce these properties.
- We develop Ocelot, an analysis that inserts atomic regions to enforce these properties without asking the programmer to think about real-time constraints, mitigations for timeliness failures, or added hardware.
- We prototype Ocelot for Rust and use it to add atomic regions to a set of programs from prior work and a tire monitoring program we developed.
- We evaluate Ocelot on real energy-harvesting hardware and show that its atomic regions ensure these properties at little runtime or programming overhead.

Due to space, we relegate detailed formalisms to the appendices, located at https://arxiv.org/abs/2104.04616.

2 Background and Motivation

Software executes intermittently on energy-harvesting systems, relying on system support to ensure progress and memory correctness despite power failures. I/O complicates an intermittent system, requiring additional correctness reasoning to ensure both correct device operation *and* the freshness and temporal consistency properties addressed by Ocelot.

2.1 The Basics of Intermittent Computing

Software executing intermittently on an energy-harvesting system makes forward progress only as sufficient energy is available. We show this in the graph in Figure 1, top. A system collects energy using, e.g., a solar panel or radio wave collector, storing small amounts of energy in a tiny battery or capacitor (red segments). After a system-specific amount of energy accumulates, hardware activates the system to begin executing, quickly consuming the energy (green segments). The executing system may collect sensor inputs, run computations (e.g., machine learning to process sensor data [16, 17, 38]) on an ultra-low-power CPU or microcontroller, and log or transmit results via a wireless radio link.

Prior work identified and addressed several progress [23, 37, 46] and correctness [2, 3, 11, 21, 23, 31, 32, 34, 35, 37, 46, 48, 53, 57] challenges to intermittent execution. The main idea in these works is to ensure that non-volatile memory remains consistent as execution proceeds in bursts. There

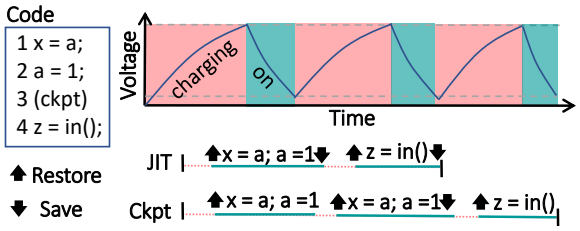

Figure 1. JIT and Checkpoint based intermittent execution

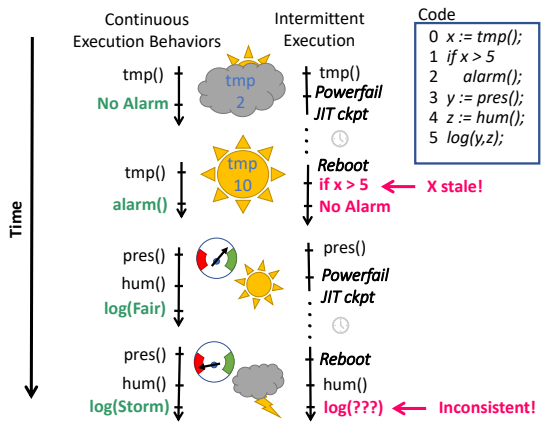

Figure 2. Freshness and temporal consistency problems.

are two broad classes of solutions, "just-in-time" checkpointing systems [3, 23, 34] and checkpoint- (or "task-") based systems [31, 32, 48, 53]. We illustrate the difference using the code snippet in Figure 1. A JIT checkpointer uses hardware to monitor energy. The software runtime backs up volatile state (registers, stack) just before power fails. On reboot, the system restores volatile state and continues. In the example, power fails after executing line 2. On reboot, the execution resumes from line 4. A checkpoint-based system encounters explicit code points where it collects a checkpoint and continues executing, such as line 3. After a power failure, the system resumes from the last saved explicit checkpoint. If power fails after executing line 2, the execution restarts from line 1. The checkpoint saves volatile state, like JIT, but also saves some non-volatile state to ensure that they remain consistent. Prior work showed that a checkpoint must back up non-volatile memory that will be accessed after the checkpoint first by a read, then by a write, i.e., a Write-After-Read (WAR) dependence [31, 53] (a in the example.)

Inputs complicate checkpointing. Unlike checkpoint-based systems, a JIT system never re-executes code after reboot. In some cases, however, correctness *requires* re-executing to re-collect an input; in such cases JIT checkpointing always renders execution incorrect. Prior work showed that checkpointing causes incorrect behavior if a value derived from an input is not correctly backed up [51, 52]. To avoid the incorrect behavior, a system must add to the checkpoint conditionally-written, non-volatile data not already checkpointed due to a WAR dependence (the "exclusive may-write" or EMW set [51, 52]). Even after resolving these memory consistency issues, inputs still complicate intermittent correctness, because of input *timeliness constraints* [12, 20, 27, 45].

2.2 Inputs Violating Freshness and Consistency

Intermittent execution can violate *freshness* and *temporal consistency*, which are implicit correctness constraints illustrated in Figure 2. The example program reads a thermometer, raising an alarm for high temperatures. The program then logs pressure and humidity sensor data that may indicate a storm. Time flows down and at left are possible continuous executions, each corresponding to the weather in the middle. On the right are intermittent executions (using JIT checkpointing). Power fails between instructions 0 and 1 and between 3 and 4, spending arbitrary time while powered off.

Violating Freshness The time delay after a power failure violates data freshness, causing incorrect behavior if the temperature changes during the delay. The continuous execution raises the alarm at high temperature. The intermittent execution, however, senses cold, then checkpoints and powers off. On reboot, it raises no alarm, despite the high temperature. The code implicitly requires x to be *fresh* when used, but the power failure prevents this. For an intermittent execution to match a continuous execution, power must not fail between sensing the temperature and executing the branch on x.

Violating Temporal Consistency The time delay after a power failure may compromise the *temporal consistency* of a collection of sensor data. With initially fair weather that becomes stormy, a continuous execution may sense high pressure and low humidity (i.e., no storm), or sense low pressure and high humidity (i.e., a storm), logging either condition. The intermittent execution, however, reads high pressure before power fails (fair weather), and high humidity after rebooting (storm). The sensed values are inconsistent with the fair or stormy weather seen by continuous executions. For intermittent execution to match continuous execution, power must not fail between the pressure and humidity readings.

2.3 Prior Approach: Timeliness

Freshness and temporal consistency are correctness conditions on *when* data from input operations may be used, similar but distinct from *timeliness* conditions in prior work [20]. Recent work on input timeliness requires an input value to be used within a programmer-specified "expiration" window after collection [15, 27, 45]. These approaches *add hardware* to keep track of time during power failures. On use of an expired value, the program must recollect the value or treat the use as an exceptional error case and run mitigation code.

While prior work has made progress toward the goal of timely intermittent execution, fundamental challenges remain unaddressed. First, the notion of timeliness (which we call "freshness") ignores important cases in which *two* input values must be from the same moment in time, but have no absolute expiration constraint. We call this timing property "temporal consistency", drawing inspiration from data-centric concurrency control [8, 9, 18]. Temporal consistency ensures that multiple values (e.g., the pressure and humidity readings) come from the same point in time.

Second, prior techniques burden the programmer by requiring reasoning about real time and demanding a distinct expiration time for each value. If the programmer incorrectly

assigns expiration times, an execution may misbehave without an expiration time violation. While identifying the data that require an expiration time may be simple, assigning the right expiration time requires choosing the correct real time value for a given program, platform, and deployment, which is not simple. Some systems [20, 27] demand more of the programmer, asking for a recovery action for expired data.

Third, prior timeliness techniques add extra time-keeping hardware: a low-power real-time clock [20, 27] or a time-keeper based on charge remanence [15, 45, 56]. The need for time-keeping hardware precludes the adoption of these techniques on unmodified platforms.

Fourth, and *most critically*, prior approaches do not formally define the timeliness properties they aim to provide, nor do they relate the behavior of an intermittent execution to that of a continuous execution. Lacking formal definitions and correctness relations makes it difficult or impossible to reason if a system is correct. A key contribution of this work is to formally define correctness criteria in relation to continuously-powered executions and to use these definitions to develop a formalism to prove if a system is correct.

3 Ocelot: Correct Inputs via Atomicity

The core of Ocelot is a compiler analysis that inserts atomic regions into code to enforce freshness and temporal consistency in intermittent executions of Rust programs. Ocelot is the first system designed to support the development of software for intermittently operating systems using Rust.

3.1 Continuous Execution as a Correctness Spec

Ocelot's correctness definitions use the idea that a continuous program execution is implicitly a specification of behavior that should be allowed by an intermittent execution, including freshness and temporal consistency properties. The arbitrary time that passes during a power failure can cause an intermittent execution to operate on inputs with timing impossible in any continuous execution, leading to incorrect behavior. Prior work [11, 32, 48, 57] uses *atomicity* of a code region to keep memory consistent. We show that atomicity is also linked to freshness and temporal consistency.

An *atomic region* saves memory state at its start. If power fails during a region, the region restores the saved state and execution continues from the start of the region on reboot. A partially executed region's updates to state are not visible. If a region completes, its effects become visible to later operations. Moreover, the region must have executed without a power failure. If a region executes without a power failure, i.e., atomically, its span of code will match the timings of a continuous execution. If multiple input operations execute atomically, the operations are temporally consistent. If an input operation and user of the input value execute atomically, the value will be fresh when used. Ocelot leverages this observation and uses atomic regions as the mechanism to enforce time constraints in intermittent executions. Code with freshness or temporal consistency requirements executes in an atomic region; atomicity ensures that the execution behavior will match some continuous execution. Code with no such requirements executes with JIT checkpoints, enjoying the low overheads of taking action only when power fails.

Jit + Atomic Execution Model Ocelot combines JIT checkpoints with atomic regions as an execution model, modularly working with any JIT checkpoint and atomic region implementation. The JIT checkpoint mechanism must checkpoint volatile memory and registers when energy is low, restoring from that checkpoint on reboot. The atomic region implementation must disable JIT checkpoints at the region's entry, instead checkpointing volatile system state and sufficient non-volatile state to ensure idempotent re-execution of the region [31, 32, 53]. Ocelot allows nested or overlapping regions, flattening them into a single region with the outermost region bounds. We describe the implementation of the JIT and Atomic runtimes used in the evaluation in Section 6.3 and show the small-step semantics in Appendix H.

3.2 From Annotations to Correct Executables

Ocelot relies on simple programmer-provided annotations to generate programs whose freshness and temporal consistency constraints are automatically enforced. Figure 3 illustrates Ocelot's workflow. The programmer annotates (in blue, upper-left) which variables have freshness or consistency constraints. Section 4 defines the precise meaning of these annotations. Ocelot must ask programmers for annotations as freshness and temporal consistency requirements are highly application- and deployment-dependent. Consider the program in Figure 2. The code logs a pair of values representing sensed pressure and humidity at line 5. If power fails between executing lines 4 and 5, the values are consistent but not fresh. If power fails between executing lines 3 and 4, the values are neither consistent nor fresh. Without annotations, it is not explicit from program syntax which of temporal consistency and freshness matter for the pair of values.

Ocelot uses the annotations to infer atomic regions satisfying the constraints. Ocelot's region inference algorithm searches for operations that must execute atomically to enforce each annotation. These operations include inputs that each operation depends on and each operation's uses of annotated data. The algorithm computes points that dominate all such operations, and adds a region enclosing those points. Section 5 describes the algorithm and its correctness; Section 6 gives the implementation details. Ocelot links the transformed code to its JIT checkpointing and atomic region runtime library, generating a correct executable.

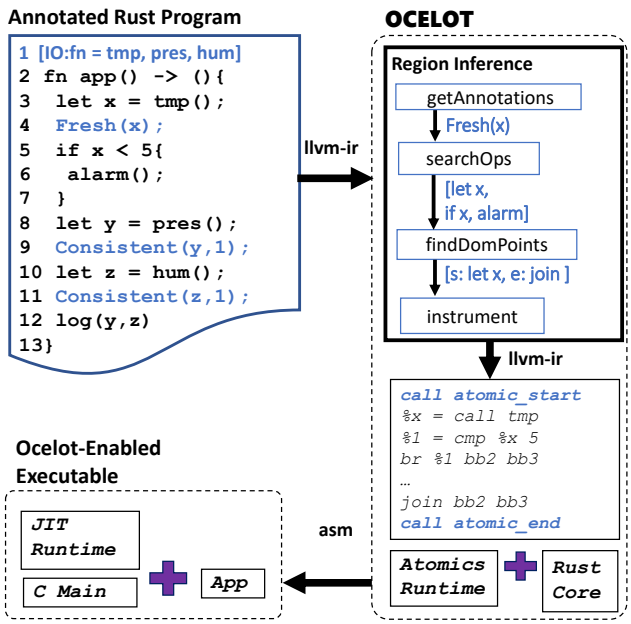

Figure 3. Visualization of the Ocelot toolchain.

3.3 Benefit of Targeting Rust

To our knowledge, Ocelot is the first intermittent computing toolchain to target Rust. Enabling correct intermittent execution of Rust programs is valuable to the community. Further, Rust provides memory safety, which contributes to correct intermittent execution in the following two ways. First, as energy-harvesting devices are often deployed to inaccessible or remote environments, a memory-unsafe program that corrupts non-volatile memory may be difficult or impossible to patch, making the device useless. Second, current intermittent systems, including Ocelot, rely on the *soundness* of static analyses for their correctness guarantees. These static analysis identify variables to checkpoint [31, 32, 51, 52] or where to place checkpoint bounds [53]. Pointer alias analysis is a hard problem in C. Missing an alias leads to memory corruption if the compiler fails to checkpoint an aliased memory location that must be checkpointed. Rust's ownership and immutability properties make alias analysis more precise [1]. Sections 5 and 6 describe how this precision benefits Ocelot's analyses. Finally, combining Ocelot with emerging formalisms and frameworks for Rust, such as Rustbelt [14, 24] and Iris [25] creates a path toward fully formally verified intermittent system implementations.

4 Formalizing Freshness and Consistency

We define a simple modeling language and introduce annotations for freshness and temporal consistency as discussed in Section 2. Then, we define their meaning by reference to allowed *correct* intermittent executions.

4.1 A Simple Language

This language includes accesses to references and arrays. A program p consists of a set of function declarations. We assume that the program starts at the main function. We show key syntax below—the rest is in Appendix A.

$$
\begin{array}{llcl}
\textit{func. decls} & FD & ::= & \cdot \mid FD, f(arg) = c; \mathsf{ret}\ e \\
\textit{instructions} & \iota & ::= & \mathsf{skip} \mid x := e \mid a[i] := e \mid *x := e \\
\textit{commands} & c & ::= & \iota \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid c_1; c_2 \\
 & & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ c \\
 & & \mid & \mathsf{let}\ x = f(v)\ \mathsf{in}\ c \mid \mathsf{let}\ x = \mathsf{IN}()\ \mathsf{in}\ c \\
 & & \mid & \mathsf{start}_{\mathsf{atom}}(aID, \omega); c; \mathsf{end}_{\mathsf{atom}}
\end{array}
$$

Commands include if statements, sequencing, variables bindings, function calls, input operations, and atomic regions, which are parameterized with a unique ID *aID* and set of checkpointed locations ω. For simplicity, we assume that let bound variables are immutable and their uses obey Rust's type system, which is the case in our benchmarks. Commands use values v, which are numbers, booleans, or references, and expressions e, which are variables, values, or operations on sub-expressions. A command can also be an instruction ι, which includes assignments to a dereferenced variable and skip. We do not have a general loop construct as bound loops can be unrolled to if statements. Unbounded loops do not introduce technical difficulties, but complicate the presentation. We do not support recursive functions, which many intermittent systems disallow.

The operational semantic rules for continuous executions are of the form: $(\tau_1, N_1, S_1, c_1) \longrightarrow (\tau_2, N_2, S_2, c_2)$, where τ is the logic time stamp, N is the nonvolatile memory, S is the calling stack, and c is the command to be executed. Intermittent executions are of the form $(\tau, \kappa, N, S, c) \Longrightarrow (\tau', \kappa', N', S', c')$, where κ is the saved execution context. Appendix H details these rules. These intermittent semantics model Ocelot's runtime. Continuously powered execution traces are sequences of $\longrightarrow$ transitions and intermittently powered execution traces are sequences of $\Longrightarrow$ transitions. The difference is that the latter saves and restores context at power failure and reboots, as described in Section 3.1.

4.2 Annotations for Freshness and Consistency

Ocelot introduces two annotations: Fresh and Consistent(id).

$$
\begin{array}{llcl}
\textit{commands} & c & ::= & \cdots \mid \mathsf{let\ fresh}\ x = e\ \mathsf{in}\ c \\
 & & \mid & \mathsf{let\ consistent(n)}\ x = e\ in\ c
\end{array}
$$

Here, let fresh x and let consistent(n) x create immutable variables. The annotation for Rust code is shown in the first box of Figure 3. On Line 4, Fresh(x), declares that any input operations x could depend on and any uses of x must not be interleaved with a power failure. The Fresh(x) annotation is violated if the input on which x depends executes before a power failure and a use of x executes after that failure. The Consistent annotation specifies temporal consistency. The annotation associates a group of variables together into a

consistent set. For any variable in the consistent set dependent on an input operation, those input operations must have executed together with no interleaving power failures. The annotation takes an ID as a parameter. All variables annotated as Consistent with the same ID are in the same set, such as y and z in Figure 3. Any input operations that y and z depend on must execute together as if they were in a continuously powered execution.

4.3 The Meaning of Freshness and Consistency

Figure 4 illustrates freshness and temporal consistency by relating the intermittent and continuously-powered executions. A double arrow is an intermittent execution trace, and a solid single arrow is a continuous execution trace. The vertical lines mark the transitions (steps) at operations. A dashed arrow denotes a dependence between operations (i.e., control- and data-flow). Each operation occurs at a logical time τ, which increases with each instruction executed.

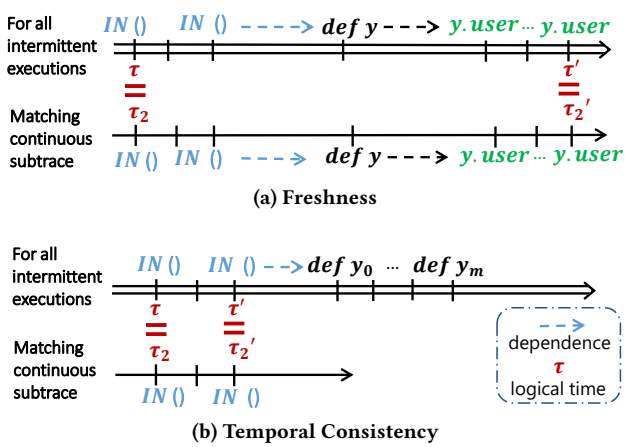

Figure 4. Illustrating the properties via execution traces

The definition of freshness is in Figure 4 (a). An intermittent system ensures that a variable y is fresh if, for *all* intermittent execution traces that include input operations on which y depends (in blue), the definition of y (in black), and dependents of y (in green), there *exists* a possible continuous execution of the program that has the same sequence of operations from the first input to the final dependent operation. Furthermore, the time span between the first input to final dependence on the intermittent execution—marked by the red timestamps—must match that of this continuous execution. In this illustration, a user of y is any instruction or command using an expression e where one of the terms of e is y. Consider a power failure between y's definition and its first use. A JIT checkpointing execution resumes from that point on reboot, but after an arbitrary period of time. The freshness property does not hold: there is no continuous execution with the same operation sequence *and* times.

The definition of temporal consistency is in Figure 4 (b). A set of variables $y_0 \ldots y_m$ is consistent if, for all intermittent traces with a set of input operations that y_i depends on (in blue), there exists a continuous execution with the same sequence of input operations and the same time difference between the first and last input operation. A power failure between input operations violates the property: an arbitrary duration may pass during power failure, and no continuous execution could have the same time difference between operations. The definitions of y_i do *not* need to be in an intermittent subtrace matching a continuous trace for temporal consistency to hold.

Formal definitions are in Appendix C. The key is to augment the semantics with taint tracking and store the input dependency information in memory so we can identify the input operations on which an annotated variable depends.

5 Ocelot Design

Ocelot's design generates programs that satisfy freshness and consistency constraints and we describe how to prove the correctness of the Ocelot design.

5.1 Ocelot Components

Ocelot has two key components to generate programs that are correct-by-construction. First, given an annotated program, Ocelot needs to identify the instructions that are relevant to each annotation; we call this record of an annotation and relevant instructions a *policy*. To construct a policy, Ocelot must identify the inputs on which an annotated variable depends, and the uses of any fresh variable. Ocelot constructs a policy using a static taint analysis to track data and control flow originating at input operations, and builds a taint summary for each function. Second, given a set of constructed policies, Ocelot adds atomic regions to the program so that all instructions in a policy are within a single atomic region. To add an atomic region for a policy, Ocelot identifies each program point that dominates all instructions in the policy and inserts the start of an atomic region at those points. The analysis inserts the end of the atomic region after the last of the instructions in the policy.

We formalize policies and summaries of input dependence in Figure 5. We assume that each instruction inside a function is given a unique label; consequently, a function name and label pair uniquely identifies an instruction. To be context sensitive, we use provenance, the sequence of calls ending in an input operation, to distinguish different calls to the same input operation (example to follow). A freshness policy is a record containing the declaration, a list of input operations and their provenance, and a list of uses. A temporal consistency policy contains a list of declarations and a list of input operations and their provenance.

The purpose of provenance information is to disambiguate multiple calls to the same function in a policy. We show an

$$
\begin{array}{llcl}
\textit{provenance} & \rho & ::= & \mathsf{nil} \mid (f_1, \ell_1) :: \rho \\
\textit{policy decls} & PD & ::= & \cdot \mid PD, pID \mapsto pol \\
\textit{policy map} & PM & ::= & \cdot \mid aID \mapsto \overrightarrow{pID} \\
\textit{taint map} & tmap & ::= & \mathsf{ret} \hookleftarrow inInfo \\
 & & \mid & \&\mathsf{arg} \hookleftarrow inInfo \\
 & & \mid & \mathsf{arg} \hookleftarrow inInfo \\
\textit{func sum.} & fsum & ::= & \overrightarrow{lSum}, \overrightarrow{CSum}
\end{array}
\qquad
\begin{array}{llcl}
\textit{policy} & pol & ::= & \mathsf{fresh}(decl : (f, \ell), inputs : \mathcal{P}uses : \overrightarrow{(f_1, \ell_1)}) \\
 & & \mid & \mathsf{consistent}(decls : \overrightarrow{(f_1, \ell_1)}, inputs : \vec{\rho}) \\
\textit{type of taint} & fromtp & ::= & \mathsf{local}(\ell) \mid \mathsf{retBy}(f, \ell) \mid \mathsf{pbr}(f, \ell) \mid \mathsf{argBy}(fromtp) \\
\textit{Inputs} & inInfo & ::= & \emptyset \mid inInfo, (input : (f, \ell), fromTp : fromtp) \\
\textit{local sum.} & lSum & ::= & \mathsf{local}(\overrightarrow{tmap}) \\
\textit{caller sum.} & CSum & ::= & \mathsf{call}(caller : (f, \ell), \overrightarrow{tmap}) \\
\textit{func sums.} & FS & ::= & \cdot \mid FS, f \mapsto fsum
\end{array}
$$

Figure 5. Syntax for policies and taint maps

example in Figure 6 (b). The main function *app* calls *confirm*. *confirm* calls the pressure sensor twice consistently. Both calls to *pres* must occur in the same atomic region. To reflect this in the policy declaration, each input is associated with its call chain (indicated in purple) to distinguish the same input with different calling contexts.

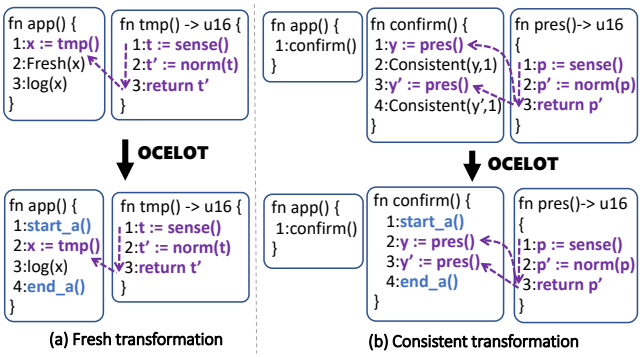

Figure 6. Policies for longer call chains

To present the results of both components, we define policy declarations *PD*, which map policy IDs to policies; a policy map *PM*, which maps atomic region IDs to policies that it enforces; and function summaries *fsum*, which are lists of local and caller summaries. A function summary contains a taint map entry, which is a link in a call chain describing how tainted information flowed into and out of the function. The entry records if taint flows through the return ($\mathsf{retBy}(f, \ell)$), into a pass-by-reference parameter ($\mathsf{pbr}(f, \ell)$), or is passed in by an argument ($\mathsf{argBy}(fromtp)$). A local summary *lsum* is used if the taint was generated within the function, in which case taint flows to any caller. For example, input is generated within the function *pres* and passed through the return, so *pres* has a local summary $\mathsf{local}(\mathsf{ret} \hookleftarrow (input{:}(sense,\ 0), fromTp{:}\mathsf{local}(1))$. A caller summary *CSum* is used when taint was passed in, in which case taint flows back only to that calling context. For example, *norm* is called by *pres* with a tainted argument which flows to the return, so *norm* has a summary including the taint map $\mathsf{call}(caller : (pres, 2), \mathsf{arg} \hookleftarrow (input{:}(sense,\ 1), fromTp{:}\mathsf{local}(1))), \mathsf{ret} \hookleftarrow (input{:}(sense,\ 1), fromTp{:}\mathsf{argBy}((pres,\ 2))))$. Linking taint map entries uncovers the entire provenance.

The two components of Ocelot are: BUILDSUMMARY(*FD*) = (*FS*, *PD*) and INFERATOMIC(*FD*, *FS*, *PD*) = (*PM*, *FD*'). We show more implementation details in Algorithm 1 in Section 6.

5.2 Sanity Checks for Results

Instead of directly proving the algorithms correct, we show a set of sanity checking rules for the results and prove that programs that pass these checks can be executed correctly intermittently. These rules resemble how the algorithms work and can additionally serve as a validation tool.

Checking Summary and Policy Declarations We first check that a function summary is correct and that the correct sets of operations are in policy declarations. The judgment is of the form: $FD; PD, FS; (g, \ell); f; M; I \Vdash c : M'; I'$. *FS* is the summary for all functions. We are checking the summary for when f is called from g on line ℓ. M and I denote the may-alias and input-dependence information prior to executing c. M' and I' are updated with any may-alias and input dependence information from c.

$$
\textsc{Call-nr}\quad
\frac{\begin{array}{l}
v \text{ not a ref.} \quad checkUse(PD, v) \quad ins = I(v) \quad FS(g) = s \\
ins \subseteq s(\mathsf{call}, f, \ell, \mathsf{arg}) \quad outs = s(\mathsf{local}, \mathsf{ret}) \cup s(\mathsf{call}, f, \ell, \mathsf{ret}) \\
outs' = outs[fromTp \mapsto \mathsf{retBy}(g, \ell)] \\
FD; PD, FS; c; f; M; I \cup (x \hookleftarrow outs') \Vdash c : M'; I'
\end{array}}{FD; PD, FS; c; f; M; I \Vdash \ell : \mathsf{let}\ x = g(v)\ \mathsf{in}\ c : M' \backslash x; I' \backslash x}
$$

$$
\textsc{Let-fresh}\quad
\frac{\begin{array}{l}
ins = I(e) \quad callChain(FS, ins) \subseteq PD(\mathsf{fresh}, f, \ell).\mathsf{ins} \\
FD; PD, FS; c; f; M \cup (x \mapsto M(e)); I \cup (x \hookleftarrow ins) \Vdash c : M'; I'
\end{array}}{FD; PD, FS; c; f; M; I \Vdash \ell : \mathsf{let\ fresh}\ x = e\ \mathsf{in}\ c : M' \backslash x; I' \backslash x}
$$

The rule CALL-NR shows an example of checking function summaries. When calling g with an argument v (not a reference), if v depends on inputs, there must be a caller summary for g that records that f propagates taint to g. Furthermore, if g returns tainted information, either locally-generated or due to f, those outputs must be propagated to x when checking the sub-command. We update the provenance information in the outputs to reflect the fact that the taint from f's perspective comes from g. Further, the second premise $checkUse(PD, v)$ checks that if v is a use of fresh policy, it has to be in the policy declaration.

The rule Let-fresh checks the fresh annotation. Any input provenance that the expression of an annotated variable depends on must be in policy associated with that annotation. We use *callChain*(*FS*, *ins*) to reconstruct the call chain. In Figure 6 the policy for the freshness example in (a) must contain the input operation $sense$ and its call chain through the return into x indicated in purple. The rule to check the consistent annotation (omitted) is similar. For our example, the two inputs are (app, 1)::(confirm, 2)::(pres, 1)::(sense(), 0) and (app, 1)::(confirm, 3)::(pres, 1)::(sense(), 0), showing two different calls to pres.

To check the entire program, we write $FD; PD, FS \vdash FS : \mathsf{ok}$ to mean that all the functions are checked under all specified calling contexts in the summary *FS*.

Finally, propagating input dependence information is simple in this modeling language as there are no mutable aliases allowed. The may-alias set for a location is always a singleton set. We can easily find out whether we are writing to a reference that is passed from the caller, which is difficult for C and thus the reason why we use Rust.

Atomic Region Checking This check is to make sure that *all* the instructions and their call chains mentioned in the policy declaration *only* appear in the correct atomic region. We write $FD; PD, PM; f; \rho; pols; aID \Vdash c : pols'$ to mean that command c in function f is currently called from the call chain ρ, within atomic region *aID*. *pol*s are the polices that *aID* enforces. After c is checked, instructions in $pols'$ still need to appear in this atomic region. When the c is not in an atomic region, *pol*s and *aID* are empty and the end of the judgment is : ok. These rules follow each call chain. For each instruction, the rule checks whether the call chain and instruction is mentioned in *PD*. If so, the current atomic region ID must match that in the *PM*. Then, this instruction is marked as reached. At the end of an atomic region, the rule checks that all instructions in *pol*s are reached. Key rules are shown in Appendix D. For a program consisting of function declarations *FD*, we say it passes the atomic region check if $FD; PD, PM; \mathsf{main}; \cdot; \emptyset; \cdot \Vdash FD(\mathsf{main}) : \mathsf{ok}$.

5.3 Correctness

We prove the following correctness theorem.

Theorem 1. *Given a program p consisting of functions in FD,* $FD; PD, FS \Vdash FS : \mathsf{ok}$*, and PM covers all policies in PD, and* $FD; PD, PM; \mathsf{main}; \emptyset; \cdot \Vdash FD(\mathsf{main}) : \mathsf{ok}$*, then p satisfies all the policies.*

The proof relates the static checking rules to the execution traces, showing that if a program p passes the checks then all input operations that an annotated fresh variable depends on, as well as any uses of the variable, will be in the same atomic region. Any input operations that any item in a consistent set depends on will be in the same region. As the committed execution of a region never experiences a power-failure, the committed execution always has the same timing-behaviour as a continuous execution for any operations in the region. Thus, w.r.t. to freshness and temporal consistency, any intermittent execution of p will preserve input freshness and temporal consistency.

To prove Ocelot correct, we only need to prove that Ocelot's algorithms produce results that pass those checks. This setup allows us to integrate seamlessly with prior work on proving memory consistency of intermittent systems [52].

Correctness of Region Size There are many possible region placements that could pass the policy check—trivially, $\mathsf{start}_{\mathsf{atom}}(aID, \omega); FD(\mathsf{main}); \mathsf{end}_{\mathsf{atom}}$. Another aspect to correct intermittent execution, however, is that any atomic region must be able to complete with the energy that can be stored in the buffer. Thus, Ocelot must infer the smallest regions that satisfy the checks to increase the likelihood that a program is also correct with respect to energy consumption. If the smallest possible region that guarantees correctness w.r.t. to timing policies is too large to complete, such a program fundamentally cannot run correctly.

6 Ocelot Implementation

Ocelot's implementation in LLVM uses the region inference algorithm to transform an annotated program *FD* into a program *FD*' that passes the checks of Section 5.2. The Ocelot implementation analyzes LLVM intermediate representation code generated from an annotated Rust program, determines the policy for each annotation, and infers and inserts atomic regions satisfying the policies. Ocelot then links with the JIT checkpointing and undo-logging atomic region runtimes.

6.1 Mapping Annotations to Policies

The implementation of the policy building component closely matches the checking rules in 5.2, except that instead of checking that an operation is in the policy declaration, as in rule Let-fresh, the algorithm starts with empty policy declarations and adds the operations to the policies at those points. The algorithm first finds all annotation instructions, which are implemented as calls to the empty functions $Fresh(var)$ and $Consistent(var, id)$. The algorithm builds a taint map associating variable definitions with inputs and the provenance of the input. Appendix I shows the map-building algorithm, which uses a taint tracking analysis that is inter-procedural, context-sensitive, and leverages Rust's ownership model to simplify pointer aliasing. We also assume no mutable globals, which are unsafe in Rust. Using the input-dependence map, the algorithm adds provenance information to the policies as described in Section 5.2. After computing the policies, the pass erases the annotations and starts region inference.

6.2 Inferring Atomic Regions

Algorithm 1 performs region inference. Given the function summary and policy declarations generated at lines 2 and 3, it

aims to generate regions that pass the policy check. The algorithm calculates a point that dominates all operations in the policy to begin the region and a point that post-dominates operations to end the region. The main challenge is that the policy operations may not be in the same function scope. The algorithm first finds a candidate function where all operations are either in the function or in a descendant of the function. It then associates each policy operation with the point in the candidate function that reaches the operation.

To find the candidate, the algorithm maps each policy operation to its basic block (Line 5) and calls FINDCANDIDATE with the block map and the root of the program. The function is recursive and tracks which basic blocks in the map execute in successor functions from the root. If all blocks in the map are executed in the current root or its successors and no candidate is set, then the root returns itself as the candidate. Consider example (b) in Figure 6. FINDCANDIDATE starts from *app* and calls itself on *confirm*. The invocation on *confirm* marks that it contains some blocks and calls itself on the calls to *pres*. These return the blocks that they called, but no candidate, as neither call to *pres* contains all blocks. Combining the results of its successors, *confirm* does contain all blocks. The invocation marks *confirm* as the candidate function, returning this to the invocation on *app*. While *app* is also a root of all the blocks, the candidate is already set, so the invocation returns *confirm*. Placing the region in *confirm* results in a smaller region than placing it in *app*.

```
1:  function INFERATOMIC(Cmd)
2:      map ← buildSummary(Cmd)
3:      pol ← buildPolicies(Cmd, map)
4:      for all set ∈ pol do
5:          ∀item ∈ set, blocks[item] ← item.basicBlock
6:          goalFunc ← findCandidate(blocks, Cmd.root)
7:          for all item ∈ set do
8:              while blocks[item].func ≠ goalFunc do
9:                  calls ← blocks[item].func.callers()
10:                 for all call ∈ calls do
11:                     if call ∈ set then
12:                         blocks[item] ← call.basicBlock
13:                     end if
14:                 end for
15:             end while
16:         end for
17:         startDom ← closestCommonDom(blocks)
18:         endDom ← closestCommonPostDom(blocks)
19:         (S, E) ← truncate(startDom, endDom, set)
20:         Cmd.insertRegAt(S, E)
21:     end for
22: end function
```

Algorithm 1. Atomic Region Inference

To find the points in the goal function that reach a policy operation, the algorithm traverses the call graph aided with the basic block map (lines 8-15). Until the function of each basic block in the map is the goal function, the algorithm gets the callers of the function and checks if the callsite is in the policy, as the policy includes the provenance. If it is, traversing this path will get the basic block closer to the goal function. The algorithm sets the map value to the basic block of the callsite. For the freshness example in Figure 6, the basic block of the assignments to t, t' is in the function *tmp*. *tmp* is called by *app* at the callsite $x := tmp$. This operation is in the policy, so the map values for t, t' are set to the basic block of the callsite. Now all blocks in the map are in *app*.

Once all blocks associated to the policy operations are in the same function, the algorithm uses LLVM's built-in CLOSESTCOMMONDOMINATOR and CLOSESTCOMMONPOSTDOMINATOR passes to return candidate *startDom* and *endDom* basic blocks (lines 17-18). Multiple returns in the source function do not cause the post-dominance analysis to break, as the compiled code has a return landing-pad that post-dominates all paths through the function. Taking these blocks, the algorithm calls TRUNCATE, which finds the latest point in the starting block that dominates everything in the set and the earliest point in the ending block that post-dominates everything in the set. Inserting region start and end instructions at these points creates an atomic region containing all the operations in the policy.

6.3 Runtime Implementation

To implement atomic regions with undo logging, we used WAR and EMW analysis code publicly available from prior work [32, 52], porting both to work for Rust code. The existing implementation has a *currentContext* variable that tracks region metadata. We add to it a *mode* field that is either *jit* or *atomic*. The value is *atomic* in an atomic region, and is *jit* otherwise. An atomic region's checkpoint also saves volatile execution context (registers, stack) along with performing undo-logging. The routines to save and restore volatile execution context are the same for both JIT checkpoints and atomic regions, and are similar to Hibernus [3]. The checkpoint routine copies registers and stack to a dedicated non-volatile memory region. Restoration copies values from non-volatile memory back into the context.

We target the Capybara energy-harvesting hardware platform [12], which has a built-in comparator to monitor energy, the only hardware needed for JIT checkpointing. The firmware triggers an interrupt on low energy. We raised the voltage level on which the interrupt triggers and modified the ISR to handle JIT mode and atomic mode. In JIT mode, the ISR checkpoints volatile state and shuts down. In atomic mode, the ISR only shuts down. Similarly to Samoyed [34], we assume that the extra energy gained from raising the trigger point will always be enough to complete the checkpoint. As pointed out in prior work [27, 33, 34], this assumption may not be true for programs with large and unpredictable stack sizes. None of our benchmarks have this behaviour

Table 1. Benchmark Characteristics. The origins: [27, 31, 34]

Origin	App	LoC	Sensors	Constraints
TICS	Activity	470	Accel*	Con, Fresh
	Greenhouse	170	Hum, Temp	Con
Samoyed	Photo	68	Photo	Con
	Send Photo	92	Photo	Fresh
DINO	CEM	292	Temp*	Fresh
Ocelot	Tire	338	Pres*, Temp*, Accel*	Fresh, Con, FreshCon

and our implementation is sufficient to demonstrate Ocelot's correctness improvements with low overhead.

7 Evaluation

We evaluate the performance and correctness of programs generated by Ocelot and the programmer effort of using Ocelot. We measure runtime overhead of a set of benchmarks compiled with Ocelot, with just JIT checkpoints, and with just Atomic regions (similar to the execution model of DINO [31]). We measure the runtimes on continuous power, showing the inherent performance overheads of Ocelot and Atomics even when energy is plentiful, and on intermittent power. While JIT is fastest, it is incorrect. Ocelot has a mean 7% runtime increase and is correct by construction. To show correctness empirically, we run the Ocelot programs with simulated power failure points chosen to be sufficient to uncover any timing violations and on real intermittent power. Finally, we compare the code changes needed to write correct programs with Ocelot, TICS [27], and Samoyed [34]. We further discuss the difference between annotating code and manually adding atomic regions in Section 8.

7.1 Benchmarks

We use the following 6 benchmarks that are representative of sensor applications in the intermittent computing domain. *Activity*, an activity recognition app, *Greenhouse*, a greenhouse monitor app, *CEM*, a compression logger, *Photo*, an app that takes the average of 5 photo-resistor readings, *SendPhoto*, an app that samples a photo resistor and sends a radio packet if the value is too high, and *Tire*, a tire pressure monitor. All benchmarks except for *Tire* were originally written in C, ported to Rust by us. *Activity* and *Greenhouse* were obtained from the TICS artifact [27], *Photo* and *SendPhoto* were microbenchmarks used in Samoyed [34] and were obtained from the authors, and *CEM* is originally from DINO [31]. *Tire* we wrote ourselves. We characterize the benchmarks in Table 1. The table shows the provenance of each benchmark, the lines of code, the sensors used or simulated (denoted with an asterisk), and the constraints used. Comma-separated values mean that the constraints apply to separate pieces of data. "FreshCon" means that both constraints were used on the *same* piece of data. Both unaltered and annotated benchmarks are located at https://github.com/CMUAbstract/ocelot.

7.2 Overheads

The goal of the performance evaluation is to make a generalizable comparison of Ocelot, which uses a JIT + Atomics execution model, to systems that use just Atomics [20, 31–33, 48, 53] or just JIT [2, 3, 23, 37]. We ran the benchmarks on the Capybara hardware platform [12], harvesting energy from a PowerCast [41] antenna placed 10 inches away.

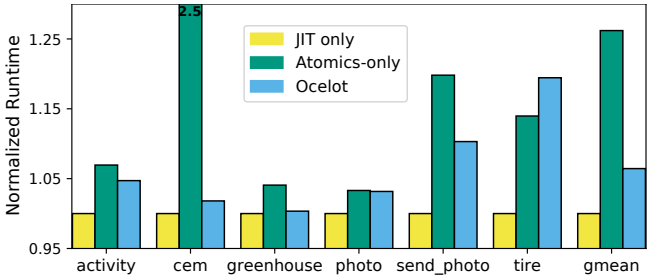

Figure 7. Continuous runtimes of JIT, Atomics, and Ocelot

Figure 7 shows the runtimes on continuous power of the benchmarks compiled with JIT checkpoints only (yellow columns), with Ocelot-inferred Atomic regions (blue columns), and with Atomic regions only (teal columns). To enable correct output, calls to the UART were guarded by a small atomic region, generating a constant overhead for all configurations. Runtimes are normalized to the JIT execution, which has the least overhead at the cost of correctness, both of timing constraints and of basic peripheral operation [5, 7, 34, 47].The y-axis shows the runtime increase, and the x-axis shows the benchmarks. The Atomics-only programs are entirely divided into atomic regions. We manually placed regions where Ocelot-inferred regions would go, to ensure that the correctness properties will still be satisfied, which is otherwise not guaranteed. If statically-placed checkpoints or tasks were used on the program in a prior work (*Greenhouse*, *Activity*, and *CEM*), we tried to place atomic regions as similarly as possible. *CEM* required a few code changes to run on the device, as the original program had a region with a WAR dependence on a large structure. Backing the entire structure to the undo log caused the program to be too large to flash to the device. We changed the code to remove any WAR dependences on that structure. Generally, atomic regions, whether manually placed or inferred add a reasonable amount of run time overhead. The geometric mean runtime increase of Ocelot programs to JIT is around 7%. Atomics-only experiences similar overheads, except for CEM which has a 2.5 runtime increase. CEM grabs a sensor value once and then performs lookup and insertion into a compressed log. The inferred atomic region is small and infrequently executed, resulting in an Ocelot runtime that is close to JIT. With Atomics-only, all lookup and insertion code is in regions even though re-execution is unnecessary for either timing or memory correctness, resulting in a large

overhead. *Tire*, in contrast, is slightly faster with Atomics-only than with Ocelot. The Atomics-only version nests a frequently executing inferred region within a larger, less frequently executing region. At runtime, only the outermost bounds are treated as an atomic region.

Next, we show the runtimes of the benchmarks on intermittent power in Figure 8. All bars are normalized to the JIT execution time on continuous power. Again, yellow represents JIT, blue represents Ocelot, and teal represents Atomics-only. For each benchmark, the lower, colored bar represents the time spent running the application, and the stacked grey bar represents the time spent off, charging. The lower sublot shows a closer view of the time spent running the application. Since JIT cannot execute peripheral operations correctly [5, 7, 34, 47], we changed *Greenhouse*, *Photo*, and *Send – Photo* to simulate sensors . Generally, the intermittent overheads have the same proportion as the continuous ones. A notable difference between the plots is that the runtime is dominated by charging time. The benchmarks were run on real hardware and harvested energy; the off, charging times are dictated by the physical environment.

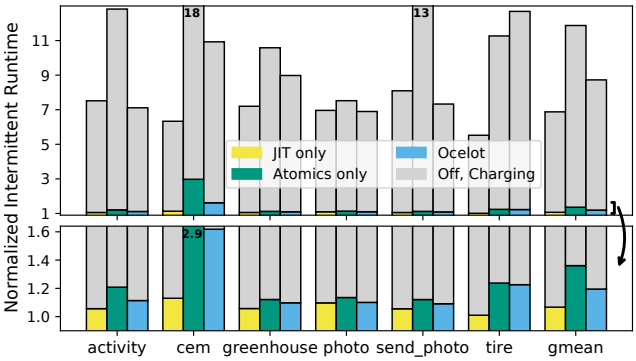

Figure 8. Intermittent runtimes of Ocelot, JIT, and Atomics

7.3 Correctness

We showed how to check Ocelot's correctness on the simple modeling language in Section 5. Here we empirically show the correctness of the implementation. Power can potentially fail at any instruction in an intermittent execution. To determine if a program will violate freshness and consistency policies, however, we need only to consider power failures at a few key locations; there must not be a visible power-failure between the inputs and the dependencies of a fresh variable, and there must not be a visible power failure between the inputs of a consistent set. Power failures outside these sub-traces do not affect if the policy is upheld. We insert simulated power failures immediately before the use of a fresh variable and between input operations in a consistent set. Power-failing at each instruction is unnecessary, as these failure points are sufficient to expose if the atomic region is placed correctly and will re-execute all necessary inputs.

Table 2. Correctness comparison of Ocelot to JIT

(a) Violating % with pathological power failure points

	Percentage Violating					
Exec. Model	**Activity**	**CEM**	**Greenhouse**	**Photo**	**Send Photo**	**Tire**
Ocelot	0%	0%	0%	0%	0%	0%
JIT	100%	100%	100%	100%	100%	100%

(b) Violating % while running intermittently

Exec. Model	**Activity**	**CEM**	**Greenhouse**	**Photo**	**Send Photo**	**Tire**
Ocelot	0%	0%	0%	0%	0%	0%
JIT	50%	0%	24%	77%	50%	3%

To determine if an input is gathered before a power failure, we add bit vector in nonvolatile memory. Each sensor operation has a unique position in the bit vector. On an input operation, the sensor's position in the bit vector is set to 1. On power failure, the bit vector is cleared. On the use of a fresh variable, the bits of any dependent sensors are checked. On an input operation in a consistent set, the bits of any preceding operations in the set are checked. If the sensor has not been re-executed, the checked bit will be zero, generating an error. Table 2 (a) shows the results of injecting these simulated power failures. Ocelot programs did not experience any violations, whereas JIT programs always did.

The previous experiment shows that a policy violation cannot occur on Ocelot programs. To show that violations do occur practically as well as theoretically on JIT programs, we ran the programs with the added bit vector on intermittent power, using the simulated sensor versions of *Greenhouse*, *Photo*, and *SendPhoto*. We ran each benchmark for a fixed time of 100 seconds and recorded the percentage of complete runs of the benchmark that contained a policy violation. Each benchmark completed between 50 - 450 times, depending on the program runtime. The results are in Table 2 (b). All benchmarks except *CEM* experienced a policy violation within that window. *CEM* is a compute heavy benchmark, and the freshness constraint only applies for a few instructions. A policy violation is possible, but experiencing a power failure at exactly the right point is rare. Benchmarks like *Activity*, *Photo*, and *SendPhoto* have time constraints that cover much of the program, so violations are frequent.

7.4 Code Changes of Ocelot vs. TICS and Samoyed

We characterize the effort of using Ocelot. We compare the JIT baseline, Ocelot, and Atomics-only, plus the prior works TICS and Samoyed in Table 3. The first column of the table shows the system. Column **Constructs** shows the language constructs each system provides to the programmer to enable correct execution with inputs. Column **Strategy** lists in brief the method to use the constructs. Column **LoC Changes** estimates the lines of code needed to implement the strategy. The last column indicates if the methods succeed in providing fresh and temporally consistent intermittent executions.

Table 3. Characterizing the Strategy of Using Ocelot

System	Constructs	Strategy	LoC Changes	Correctly Upholds Freshness and Consistency
Ocelot	Time-constraint Types	Annotate inputs, time-constrained data	1*(num inputs) + 1*(data with constraint)	**Correct. Intermittent execution must match the continuous specification**
JIT	None	Do nothing	0	**Incorrect**
Atomics	Atomic Regions	Annotate inputs, manually place regions. Reason about control, data flow.	1*(num inputs) + 2*(num atomic regions)	**Programmer-dependent could place regions incorrectly**
TICS	Timestamp alignment, Expiration Catch, Timely Branches	Add real-time expiry date, timestamp alignment operations, and expiration/branch points. Write exception handlers.	3*(time-sensitive data) + $\Sigma_{i=0}^{n}(LoC\ of\ handler_i)$	**Real-time timeliness No clear mapping to temporal consistency**
Samoyed	Atomic Functions	Reason about control, data flow. Rewrite code to be function, (opt) provide software fallback, (opt) scaling rules.	$\Sigma_{i=0}^{n}(rewrite\ cost\ of\ f_i)$ + $\Sigma_{i=0}^{n}(LoC\ of\ scalingRule_i)$ + $\Sigma_{i=0}^{n}(LoC\ of\ fallback_i)$	**Programmer-dependent could put wrong code in atomic function**

Ocelot requires only a small, bounded amount of code changes. The programmer must declare which functions generate input and apply Fresh and Consistent annotations to variables. Each annotation requires adding a single line of code, and the programmer *never* has to write new program logic. The resultant program is correct by construction.

JIT checkpointing provides nothing to the programmer, requiring no effort but offering no correctness. Atomics-only requires the programmer to reason about the dataflow and relationships of input operation to each other and place the regions. Since undo-logging backs up EMW sets [52], the programmer must also specify inputs. If the programmer reasons correctly, the resultant program will be correct.

TICS [27] offers the programmer annotations that require reasoning about real time. It provides expiry times, a timestamp alignment operator, an expiration check, and a timely branch check. The latter checks also allow the programmer to specify an exception-like handler to execute if the check fails. Handlers impose an unknown burden on the programmer as they have to write new logic. If the original program has explicit real-time checks and exception handling, the process is straightforward and is a good match for TICS. Otherwise, the programmer must generate these from scratch. TICS ensures that stale data is not processed, similar to freshness, though it does not guarantee the existence of a continuous execution with the same behaviour. If the programmer chooses an expiration time poorly, the program could behave in undesired ways. The TICS concept of timeliness does not cover temporal consistency.

Samoyed [34] focuses on safe peripheral operations and provides the programmer with atomic functions. Samoyed requires more rewriting work than simple atomic regions, as the code to be executed atomically must be a function. The programmer can also specify scaling rules and fallbacks, if the function takes too much energy to execute within a power cycle. If the programmer carefully reasons about the dependencies and relationships of input operations, they can use atomic functions to uphold freshness and consistency.

Table 4. Using Ocelot vs. TICS and Samoyed

	LoC Changes						Real-time	Data-flow
Sys	**Act**	**CEM**	**G-house**	**Photo**	**S-Photo**	**Tire**	**Reasoning**	**Reasoning**
Ocelot	5	2	7	2	4	9	**No**	**No**
TICS	20	8	12	8	8	32	**Yes**	**No**
Samoyed	18	4	6	12	4	24	**No**	**Yes**

In Table 4, we estimate the concrete lines of code needed to enable correct execution on each of our benchmarks for Ocelot, TICS, and Samoyed. For TICS, we estimate that each handler will take five lines of code. For consistent sets, we estimate that each variable incurs 2 LoC changes (expiry and timestamp alignment), but that there is only one expiration check and accompanying handler per set. For Samoyed, we estimate that restructuring into atomic functions will take a fixed 3 LoC (creating the atomic function signature, adding the callsite), plus an additional line for each parameter to the function. Scaling rules take 3 LoC, fallbacks take 5 LoC, and these are provided for any atomic function with a loop. For all benchmarks, Ocelot requires the fewest annotations. Moreover, Ocelot does not require reasoning about real-time values, about information flow from inputs, or writing exception handling, instead enforcing correctness by construction.

8 Discussion of Annotation Benefits

In this section we discuss the benefits of Ocelot annotations as compared to manually adding atomic regions. Instead of using Ocelot annotations and allowing the system to infer atomic region placement, programmers can carefully place atomic region constructs to uphold timing constraints, but such an approach has several drawbacks.

Annotation Simplicity and Meaning While adding Ocelot annotations and manually adding atomic regions both require the programmer to be aware of timing invariants in their program, programmers must use additional reasoning to correctly place atomic regions. Figure 9 shows a code snippet from the tire benchmark. The snippet describes the decision whether or not to send out a burst tire alarm. This

```
1    FreshConsistent(avgDiff, 1);
2    FreshConsistent(&currMotion,1);
3    if isMoving(&currMotion) && avgDiff > 0 {
4        sendData("urgent_burst_tire!\r\n\0");
5        *urgentWarningCount +=1;
6    }
```

Figure 9. Tire code snippet

```
1  fn main ()  {                fn confirm() {
2    //should be consistent       let y = pres();
3    confirm();                   Consistent(y,1);
4  }                              let y' = pres();
5                                 Consistent(y,1);
6                                 ... //more processing
7                              }
```

Figure 10. The intuitive atomic region around CONFIRM could be too expensive

decision should happen on a fresh sensor reading, and variables in the branch should be consistent with each other. Such a level of knowledge about program behaviour is sufficient to add Ocelot annotations – `currMotion` and `avgDiff` should be marked Fresh and Consistent as in lines 1-2.

To manually place an atomic region, the programmer has to examine the data each of the variables depends on and make sure any inputs in that data flow are included in the atomic region. The programmer must know the invariants in either case, but adding an atomic region that includes every input the variables depends on and every use of the variables requires more work than annotating the variables at the declaration point only. Even knowing the invariants, the programmer could make a mistake when manually adding a region, which would not be detected by the system as added atomic regions do not carry any specification information. The program has no explicitly declared guarantees of what properties are met. When using Ocelot annotations, however, the programmer is explicitly giving a specification of the timing properties that must be upheld, and the Ocelot-generated program will correctly uphold that specification.

Region Size As discussed in Section 5.3, Ocelot's implementation aims to find the smallest region that satisfies the specified timing constraints. A programmer-added region may be uncessarily large. Consider the programming pattern in Figure 10. The function *main* calls function *confirm* which has a temporal consistency constraint on the assignments to y, y'. Programs with this pattern will likely do more processing on y, y' in *confirm*. If a programmer manually adding regions knows that *confirm* calls sensors that need to be consistent, they may simply wrap the entire function in an atomic region. While such a region placement does preserve the timing constraints, it uncessarily includes any processing in *confirm*, while the Ocelot region would not. If sampling the sensors *and* processing the data takes more energy than can fit in the buffer, the program with manually-added regions would fail to complete, while the Ocelot program would not. If an Ocelot program fails to complete, the specified timing constraints are fundamentally unsatisfiable with the energy capacity of the device.

Using added regions and Ocelot together Programmers may have programs that already have atomic regions placed, e.g., if they used Samoyed [34] to write programs with safe peripheral operations, or otherwise want manual control over atomic region placement (that they are sure will run to completion). Ocelot can be used with programs that already have atomic regions. In this use case, Ocelot's analysis confirms that the region placement meets a program's annotated timing constraints. If the input to Ocelot is a program that already has atomic regions as well as annotations, Ocelot adds regions to enforce the annotations. While these added regions may overlap or duplicate existing ones, only the outermost bounds of nested regions execute (see Appendix H). The resultant program respects the atomicity of both programmer-specified and inferred regions without extra runtime overhead. Thus, Ocelot in conjunction with manually added regions can give the programmer control and correctness. Additionally, extending Ocelot with a true checker mode is straightforward. After generating the policy sets, Ocelot could merely check that all instructions in each set are dominated by existing region boundaries, instead of inferring and placing the region boundaries.

9 Related Work

Areas related to Ocelot are intermittent systems with timeliness and reactivity, work on persistent memory correctness and crash-consistency, and data-centric concurrency.

Intermittent Systems with Inputs MayFly [20] introduced the concept of timeliness, but its solution is complicated, requiring programmers to write programs as dataflow graphs with expirations on the edges. TICS [27] is most similar to this work, providing timely intermittent computation through annotations on existing programs. In contrast to Ocelot, both these works require reasoning about real-time, do not examine temporal-consistency, and require additional hardware to keep time through power failures [15, 56]. TICS also presents an architecture for constant-time checkpoints, which is complementary and can be used with Ocelot.

Samoyed [34], RESTOP [47], Sytare [5] and Karma [7] all address the problem of safe peripheral manipulation on intermittent systems, but do not consider application-level time-constraints. Samoyed provides atomically executing functions which can be used to ensure freshness and consistency, though at more effort than Ocelot. Samoyed also provides fallbacks if an atomic function is too large, which Ocelot does not. Karma additionally considers asynchronous inputs, i.e., from interrupts, which Ocelot does not.

Capybara [12] is a hardware platform with a reconfigurable energy buffer, allowing for larger atomic regions to

be executed when needed. HomeRun [26] also explores hardware support for atomicity in I/O events. Accumulative Display Updating [36] explores relaxing atomicity constraints for long-running peripheral operations, such as updating displays, which does not meet the correctness definitions of Ocelot. Coati [48] and InK [57] focus on event-driven execution and are task-based. Tasks can be used with programmer effort to ensure freshness and consistency.

Correctness Reasoning Prior works [4, 13, 52] model the correctness of intermittent systems. Intermittent computing correctness is also similar to correctness of persistent memory [22, 39, 40, 42–44] and to file system crash consistency [6, 10, 28, 50]. Our notion of correctness follows most closely from [6, 28, 52], which define intermittent (or crashy) executions as correct if they are a refinement of some continuous (or non-crashy) execution. However, all these works define correctness in terms of memory consistency, and this continuous execution may pause arbitrarily. In this work, we show that these pauses introduce behaviour in the intermittent execution that is undesirable, even though memory is consistent. Our definitions of fresh and temporal consistency impose constraints on where these pauses are allowed.

Transactions and Data-Centric Concurrency Atomic regions are similar to transactions [19, 49], though transactions use atomicity for concurrency, not timely processing of inputs. We draw the concept of consistent sets from the line of *data-centric* concurrency control research [8, 9, 18, 54, 55]. The data-centric approach is that programmers should indicate data that need to be synchronized, rather than onerously reasoning about operations and trying to place synchronization constructs accordingly. Data-coloring [9] is a programming model to automatically infer transaction placement for data consistency, but it does so dynamically, requiring hardware support. [54, 55] use types and static analysis to automatically infer synchronization operation placement, such as locks, that guarantees correctness for specified atomic sets, but the meaning of correctness is different. An atomic set is correct if it is serializeable; intermittent programs may experience timing violations even when memory safe.

10 Conclusion and Future Work

We present the properties of freshness and temporal consistency for intermittent executions, linking the correct timing behaviour of an intermittent execution to that of a continuous execution. Using these definitions, we observe that atomicity can be used to provide correct timing behaviour as well as memory consistency. To help enforce timing constraints, we develop Ocelot, which is lightweight, and unlike prior work does not require external hardware or complex reasoning about real-time expiration or dataflow. Ocelot uses simple annotations indicating which data should be fresh or temporally consistent to infer atomic regions placement that automatically enforces correct behaviour at runtime. The development of Ocelot additionally leads to several avenues of future work.

Integration with Rust formalisms Ocelot is the first intermittent computing toolchain to target Rust programs. Rust is an attractive target for intermittent computing systems as Rust programs are memory safe, reducing the likelihood that memory bugs will make a device inoperable after deployment to a remote environement. To prove that intermittence does not subvert the safety guarantees of Rust, however, future work should integrate intermittent computing semantics into existing Rust formalisms [14, 24].

User Studies on Programmer Effort We discuss the stategies and model the lines of code needed to use Ocelot, TICS, and Samoyed in Section 7.4. Truly comparing programmer effort and usability, however, needs to be done via user study. Ocelot raises the usability questions of real-time versus implicit annotations, as well as annotations versus manually added regions. Carrying out a comprehensive user study on such features would benefit future system designers.

Reasoning about Forward Progress Along with memory consistency and timing-constraints, another key issue of correctness for intermittent computing systems is ensuring that programs can execute to completion. Analyses that identify the minimum atomic regions necessary for correct execution, such as Ocelot's, can serve as a foundation for developing tools and formalisms to reason about forward progress and the inherent energy constraints of a program.

Acknowledgements

We thank the anonymous reviewers for their feedback and Martin Rinard for shepherding this work. We also thank members of the Abstract Research Lab for their insightful comments on initial drafts. This work was generously funded in part through National Science Foundation Award 2007998, National Science Foundation CAREER Award 1751029, and the CMU CyLab Security & Privacy Institute.

References

[1] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. *SIGOPS Oper. Syst. Rev.* 51 (Sept. 2017). https://doi.org/10.1145/3139645.3139660

[2] D. Balsamo, A. Weddell, A. Das, A. Arreola, D. Brunelli, B. Al-Hashimi, G. Merrett, and L. Benini. 2016. Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* PP, 99 (2016), 1–1. https://doi.org/10.1109/TCAD.2016.2547919

[3] Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. 2015. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters* 7, 1 (2015), 15–18. https://doi.org/10.1109/LES.2014.2371494

[4] Gautier Berthou, Pierre-Évariste Dagand, Delphine Demange, Rémi Oudin, and Tanguy Risset. 2020. Intermittent Computing with Peripherals, Formally Verified. In *The 21st ACM SIGPLAN/SIGBED Conference*

on Languages, Compilers, and Tools for Embedded Systems (LCTES '20). 85–96. https://doi.org/10.1145/3372799.3394365

[5] Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. 2017. Peripheral state persistence for transiently-powered systems. In *2017 Global Internet of Things Summit (GIoTS)*. https://doi.org/10.1109/giots.2017.8016243

[6] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and Checking File System Crash-Consistency Models. *SIGARCH Comput. Archit. News* 44, 2 (March 2016), 83–98. https://doi.org/10.1145/2980024.2872406

[7] Adriano Branco, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. 2019. Intermittent Asynchronous Peripheral Operations. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems (SenSys '19)*. 55–67. https://doi.org/10.1145/3356250.3360033

[8] Luis Ceze, Pablo Montesinos, Christoph von Praun, and Josep Torrellas. 2007. Colorama: Architectural Support for Data-Centric Synchronization. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*. 133–144. https://doi.org/10.1109/HPCA.2007.346192

[9] Luis Ceze, Christoph von Praun, Calin Cascaval, Pablo Montesinos, and Josep Torrellas. 2008. Concurrency control with data coloring. In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08), Seattle, Washington, USA, March 2, 2008*, Emery D. Berger and Brad Chen (Eds.). 6–10. https://doi.org/10.1145/1353522.1353525

[10] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. 18–37. https://doi.org/10.1145/2815400.2815402

[11] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. https://doi.org/10.1145/2983990.2983995

[12] 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)*. https://doi.org/10.1145/3173162.3173210

[13] Manjeet Dahiya and Sorav Bansal. 2018. Automatic Verification of Intermittent Systems. In *Verification, Model Checking, and Abstract Interpretation*, Isil Dillig and Jens Palsberg (Eds.). https://doi.org/10.1007/978-3-319-73721-8_8

[14] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt Meets Relaxed Memory. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 34 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371102

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

[16] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. 2019. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 199–213. https://doi.org/10.1145/3297858.3304011

[17] Graham Gobieski, Amolak Nagi, Nathan Serafin, Mehmet Meric Isgenc, Nathan Beckmann, and Brandon Lucia. 2019. MANIC: A Vector-Dataflow Architecture for Ultra-Low-Power Embedded Systems. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. 670–684. https://doi.org/10.1145/3352460.3358277

[18] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. 2008. Dynamic Detection of Atomic-Set-Serializability Violations. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. 231–240. https://doi.org/10.1145/1368088.1368120

[19] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*. 289–300. https://doi.org/10.1145/165123.165164

[20] Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17)*. Article 17. https://doi.org/10.1145/3131672.3131673

[21] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. https://doi.org/10.1145/3079856.3080238

[22] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing*, Cyril Gavoille and David Ilcinkas (Eds.). 313–327.

[23] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. 2014. QUICKRECALL: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers. In *Proceedings of the 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems (VLSID '14)*. 330–335. https://doi.org/10.1109/VLSID.2014.63

[24] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

[25] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[26] Chih-Kai Kang, Chun-Han Lin, Pi-Cheng Hsiu, and Ming-Syan Chen. 2018. HomeRun: HW/SW Co-Design for Program Atomicity on Self-Powered Intermittent Systems. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '18)*. Article 29. https://doi.org/10.1145/3218603.3218633

[27] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. 2020. Time-Sensitive Intermittent Computing Meets Legacy Software. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. 85–99. https://doi.org/10.1145/3373376.3378476

[28] Eric Koskinen and Junfeng Yang. 2016. Reducing Crash Recoverability to Reachability. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. 97–108. https://doi.org/10.1145/2837614.2837648

[29] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[30] Y. Lin, P. Hsiu, and T. Kuo. 2019. Autonomous I/O for Intermittent IoT Systems. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 1–6. https://doi.org/10.1109/ISLPED.2019.8824923

[31] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proceedings*

of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015). https://doi.org/10.1145/2737924.2737978

[32] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 96 (Oct. 2017), 96:1–96:30 pages. https://doi.org/10.1145/3133920

[33] Kiwan Maeng and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. 129–144. http://dl.acm.org/citation.cfm?id=3291168.3291178

[34] Kiwan Maeng and Brandon Lucia. 2019. Supporting Peripherals in Intermittent Systems with Just-in-Time Checkpoints. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. 1101–1116. https://doi.org/10.1145/3314221.3314613

[35] Kiwan Maeng and Brandon Lucia. 2020. Adaptive Low-Overhead Scheduling for Periodic and Reactive Intermittent Execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. 1005–1021. https://doi.org/10.1145/3385412.3385998

[36] Hashan Mendis and Pi-Cheng Hsiu. 2019. Accumulative Display Updating for Intermittent Systems. *ACM Transactions on Embedded Computing Systems* 18 (10 2019), 1–22. https://doi.org/10.1145/3358190

[37] Azalia Mirhoseini, Ebrahim M Songhori, and Farinaz Koushanfar. 2013. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs. In *Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on*. https://doi.org/10.1109/PerCom.2013.6526735

[38] Matteo Nardello, Harsh Desai, Davide Brunelli, and Brandon Lucia. 2019. Camaroptera: A Batteryless Long-Range Remote Visual Sensing System. In *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems (ENSsys'19)*. 8–14. https://doi.org/10.1145/3362053.3363491

[39] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*.

[40] Steven Pelley, Peter M Chen, and Thomas F Wenisch. 2015. Memory Persistency: Semantics for Byte-addressable Nonvolatile Memory Technologies. *IEEE Micro* 35, 3 (2015), 125–131. https://doi.org/10.1109/MM.2015.46

[41] PowerCast Inc. 2020. PowerCast Antennae Information. https://www.powercastco.com/products/powercaster-transmitter/. Visited November 19th, 2020.

[42] Azalea Raad and Viktor Vafeiadis. 2018. Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 137 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276507

[43] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019. Persistency Semantics of the Intel-X86 Architecture. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 11 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371079

[44] Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak Persistency Semantics from the Ground Up: Formalising the Persistency Semantics of ARMv8 and Transactional Models. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, Article 135 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360561

[45] Amir Rahmati, Mastooreh Salajegheh, Dan Holcomb, Jacob Sorber, Wayne P. Burleson, and Kevin Fu. 2012. TARDIS: Time and Remanence Decay in SRAM to Implement Secure Protocols on Embedded Devices without Clocks. In *21st USENIX Security Symposium (USENIX Security 12)*. 221–236. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/rahmati

[46] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. https://doi.org/10.1145/1950365.1950386

[47] Alberto Rodriguez Arreola, Domenico Balsamo, Geoff V Merrett, and Alex S Weddell. 2018. RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems. *Sensors* 18, 1 (2018), 172.

[48] Emily Ruppel and Brandon Lucia. 2019. Transactional Concurrency Control for Intermittent, Energy-Harvesting Computing Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. 1085–1100. https://doi.org/10.1145/3314221.3314583

[49] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. https://doi.org/10.1145/224964.224987

[50] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 1–16. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson

[51] Milijana Surbatovich, Limin Jia, and Brandon Lucia. 2019. I/O Dependent Idempotence Bugs in Intermittent Systems. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, Article 183 (Oct. 2019), 31 pages. https://doi.org/10.1145/3360609

[52] Milijana Surbatovich, Brandon Lucia, and Limin Jia. 2020. Towards a Formal Foundation of Intermittent Computing. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 163 (Nov. 2020), 31 pages. https://doi.org/10.1145/3428231

[53] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. 17–32.

[54] Mandana Vaziri, Frank Tip, and Julian Dolby. 2006. Associating Synchronization Constraints with Data in an Object-Oriented Language. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. 334–345. https://doi.org/10.1145/1111037.1111067

[55] Mandana Vaziri, Frank Tip, Julian Dolby, Christian Hammer, and Jan Vitek. 2010. A Type System for Data-Centric Synchronization. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*. 304–328.

[56] Harrison Williams, Xun Jian, and Matthew Hicks. 2020. Forget Failure: Exploiting SRAM Data Remanence for Low-Overhead Intermittent Computation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. 69–84. https://doi.org/10.1145/3373376.3378478

[57] Kasim Sinan Yildirim, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SenSys '18)*. 41–53. https://doi.org/10.1145/3274783.3274837