A Type System for Safe Intermittent Computing

MILIJANA SURBATOVICH, Carnegie Mellon University, USA
NAOMI SPARGO, Carnegie Mellon University, USA
LIMIN JIA, Carnegie Mellon University, USA
BRANDON LUCIA, Carnegie Mellon University, USA

Batteryless energy-harvesting devices enable computing in inaccessible environments, at a cost to programmability and correctness. These devices operate intermittently as energy is available, using a recovery system to save and restore state. Some program tasks must execute atomically w.r.t. power failures, re-executing if power fails before completion. Any re-execution should typically be *idempotent*—its behavior should match the behavior of a single execution. Thus, a key aspect of correct intermittent execution is identifying and recovering state causing undesired non-idempotence. Unfortunately, past intermittent systems take an ad-hoc approach, using unsound dataflow analyses or conservatively recovering all written state. Moreover, no prior work allows the programmer to directly specify idempotence requirements (including allowable non-idempotence).

We present Curricle, the first *type system* approach to safe intermittence, for Rust. Type level reasoning allows programmers to express requirements and retains alias information crucial for sound analyses. Curricle uses information flow and type qualifiers to reject programs causing undesired non-idempotence. We implement Curricle's type system on top of Rust's compiler, evaluating the prototype on benchmarks from prior work. We find that Curricle benefits application programmers by allowing them to express idempotence requirements that are checked to be satisfied, and that targeting programs checked with Curricle allows intermittent system designers to write simpler recovery systems that perform better.

CCS Concepts: • **Computer systems organization** → **Embedded software**; • **Software and its engineering** → *Domain specific languages.*

Additional Key Words and Phrases: intermittent computing, energy harvesting, information flow, type systems

ACM Reference Format:
Milijana Surbatovich, Naomi Spargo, Limin Jia, and Brandon Lucia. 2023. A Type System for Safe Intermittent Computing. *Proc. ACM Program. Lang.* 7, PLDI, Article 136 (June 2023), 25 pages. https://doi.org/10.1145/3591250

1 INTRODUCTION

Batteryless energy-harvesting devices (EHDs) enable sensing and computing in harsh or inaccessible environments where battery maintenance is infeasible, such as space [Lucia et al. 2021; NASA 2019], civil infrastructure monitoring [Adkins et al. 2016], and in-/on-body health sensors [Curtiss et al. 2022; iota Biosciences 2022]. In lieu of a battery, an EHD harvests energy from its environment into a capacitor, powering on and operating only after collecting sufficient energy. As the device operates, it executes applications that sense, compute, and communicate, quickly consuming the energy. Once the energy is depleted, the device powers off until it can recharge and begin the cycle anew. To execute programs through frequent, arbitrarily-timed power failures, an EHD typically

Authors' addresses: Milijana Surbatovich, Carnegie Mellon University, Pittsburgh, USA, milijans@andrew.cmu.edu; Naomi Spargo, Carnegie Mellon University, Pittsburgh, USA, nspargo@andrew.cmu.edu; Limin Jia, Carnegie Mellon University, Pittsburgh, USA, liminjia@cmu.edu; Brandon Lucia, Carnegie Mellon University, Pittsburgh, USA, blucia@cmu.edu.

2475-1421/2023/6-ART136
https://doi.org/10.1145/3591250

uses a mixture of volatile (e.g., SRAM) and non-volatile memory (e.g., FRAM [TI Inc. 2014], STT-MRAM [Gobieski et al. 2019; Guo et al. 2010]). Mixed volatility requires intermittent system support to recover volatile execution state (e.g., registers) and to ensure non-volatile data remain consistent.

Maintaining correctness in an intermittent execution is challenging. Some program tasks, e.g., accessing sensor peripherals or timely data processing, must execute atomically w.r.t. power failures [Kortbeek et al. 2020b; Maeng and Lucia 2019; Surbatovich et al. 2021], re-executing if interrupted by power failure. For correctness, any re-execution should be *idempotent* unless otherwise specified by the programmer—a partial execution followed by a complete execution should have the same result as a single, complete execution. Past work identified that certain *write-after-read* (WAR) [Lucia and Ransford 2015; Van Der Woude and Hicks 2016] and *repeated-input-operation* (RIO) [Rodriguez Arreola et al. 2018; Surbatovich et al. 2020] access patterns cause re-execution to be non-idempotent. For example, if a variable x is stored in non-volatile memory, re-executing the WAR $x := x + 1$ will cause x to be incremented twice. If a program's paths branch on input, that input could produce a different value on re-execution, causing both paths to execute.

To enable idempotent execution, the recovery system must save and restore variables involved in WARs and RIOs, along with volatile state. Past work has followed two main strategies: conservatively restoring *all* written state [Kortbeek et al. 2020a,b; Maeng and Lucia 2020; Ruppel and Lucia 2019] or using dataflow analysis on the CFG to identify the variables involved in WAR and RIO patterns [Lucia and Ransford 2015; Maeng et al. 2017; Surbatovich et al. 2021, 2020; Van Der Woude and Hicks 2016]. A disadvantage of restoring all written state is its time and energy overhead. A disadvantage of the dataflow approach is that frequently system designers trade off correctness for performance, due to pointer aliasing. Most prior work in intermittent computing targets C, in which unsafe pointers make sound alias analysis extremely conservative. To counter this conservatism, many systems require programmers to adhere to informally-specified, unchecked restrictions (e.g., no pointer arithmetic [Kortbeek et al. 2020b], no function pointers [Maeng et al. 2017]). Moreover, no prior work allows the programmer to specify that data should be manipulated *non-idempotently*, which is a requirement in some applications (e.g., tracking partial peripheral interactions) and in intermittent systems code (e.g., counting reboots). EHDs are designed for remote, low-maintenance deployment, requiring reliability and adherence to a specific (i.e., formal) correctness definition. Unfortunately, today's intermittent systems programmer cannot express idempotence requirements and must depend on systems with needless overheads or unsound analyses.

To enable programmers to build correct systems that meet application requirements, we present Curricle. Curricle consists of the first *type system* for safe intermittent computing, a core calculus based on a subset of Rust, and an abstract requirements specification of an intermittent recovery system. Curricle enables a programmer to express idempotence requirements via types. Building Curricle on top of the typed, memory-safe language Rust allows precisely analyzing read and write accesses at the type level, without assuming the programmer follows unchecked restrictions. Providing an abstract recovery system specification separates reasoning about *what* state should be saved, identified by Curricle type inference, from low-level details about *how* state is saved; if a program type checks, it will run correctly on *any* intermittent system that meets Curricle's simple specification, regardless of the implementation mechanism (e.g., undo vs redo logging).

Curricle uses information flow typing and type qualifiers denoting *access modes* to reason about idempotence and input taintedness, inferring which variables must be recovered to satisfy the programmer's idempotence requirements. It augments data types with a pair of qualifiers denoting *idempotence level*– idempotent or non-idempotent–and *input dependence*, tainted or not-tainted. For example, a variable x of type $int@\langle \mathsf{Id}, \mathsf{Nt} \rangle$ is idempotent and not tainted. If the programmer wants x to be non-idempotent, they can type it as $int@\langle \mathsf{Nid}, \mathsf{Nt} \rangle$. As Curricle allows the programmer to specify variables as non-idempotent, its correctness theorem follows from non-interference.

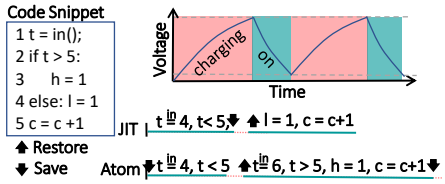

Fig. 1. Just-in-time and atomic intermittent execution

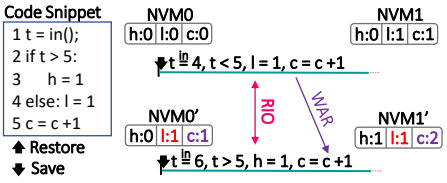

Fig. 2. Memory inconsistency caused by write-after-reads (WAR) and repeated-input-ops (RIO)

Non-interference requires equivalence between idempotent variables only, which is more general than and entails the full equivalence defined in past work [Surbatovich et al. 2020]. We implement the Curricle type system using Rust procedural macros and evaluate the prototype on benchmarks ported from prior work, finding that Curricle is beneficial to both intermittent system designers and application programmers. Targeting Curricle combines the advantages of prior approaches, enabling a recovery system to reduce dynamically logged locations up to $0.5x$ and memory footprint up to $0.6x$, compared to a system that logs everything, without requiring the system designer to provide complex dataflow analyses. Programmers can express sophisticated idempotence requirements and have assurance that their program will execute correctly. In summary, our contributions are:

- The first type system to check if programs will execute correctly intermittently, and a correctness theorem following from non-interference
- An abstract requirements specification for the underlying system that decouples correctness reasoning from the implementation mechanics
- An implementation of the type system for Rust, enabling programmers to specify their idempotence requirements without assuming unchecked program properties
- An evaluation showing that Curricle allows for simpler, more efficient system implementations, while giving application programers assurance their programs will execute correctly

For space, we put most definitions and proofs in the companion Appendix [Surbatovich et al. 2023].

2 BACKGROUND & MOTIVATION

We discuss intermittent computing basics and the weaknesses of current approaches, motivating the need to distinguish between application correctness requirements—i.e., which data to save/restore—and requirements of the underlying recovery system—i.e., how to save/restore state correctly.

2.1 Intermittent Computing: *When*, *What*, and *How* to Recover from Power Failures

An intermittently executing program makes progress as energy is available. Figure 1 shows how device energy (plotted as capacitor voltage) varies with time. A device harvests energy from its environment (e.g., sun, radio waves), buffering energy in a capacitor. When fully charged, the device begins operating, consuming energy (green). When depleted, the device powers off (red), volatile state clears, and non-volatile state persists. To make progress and run correctly intermittently, a device requires a *recovery system* to handle power failures. The recovery system design determines *when* to recover, *what* state to recover, and *how* to recover that state.

Intermittent Execution Models: the *When*. Just-in-time (JIT) checkpointing [Balsamo et al. 2016, 2015] and atomic regions (aka tasks) [Hester et al. 2017; Lucia and Ransford 2015; Maeng et al. 2017; Maeng and Lucia 2018; Ruppel and Lucia 2019; Van Der Woude and Hicks 2016] are two common intermittent execution models. The bottom of Figure 1 illustrates these two models with the aid of the code snippet on the left. The code reads an input into `t`, sets `h` if the value is too high, and otherwise sets `l`. The program then increments the counter `c`.

A JIT checkpoint scheme saves state just before power failure, which the system detects using voltage monitoring hardware. After reboot, the system restores execution to just before the failure. The top of Figure 1 shows such an execution. The trace sees an input value of four and checks the branch. Then, energy is exhausted and the system saves state. On reboot, the trace sets the low flag and updates the counter. The advantage of JIT checkpoints is that they are simple and save a small amount of state, resulting in better performance and few code changes. The disadvantage of JIT checkpoints is that checkpoint placement is arbitrary, leading to an abrupt interruption and arbitrary delay before reboot, which may not be well-tolerated by code. Peripheral state may be left inconsistent, resulting in crashes [Berthou et al. 2017; Branco et al. 2019; Maeng and Lucia 2019; Rodriguez Arreola et al. 2018], and time-constrained sensor data may become stale or inconsistent [Hester et al. 2017; Kortbeek et al. 2020b; Surbatovich et al. 2021].

An atomic region scheme ensures that if power fails, execution resumes from the start of the atomic region, potentially leading to the re-execution of some operations, as shown in Figure 1, lower. After gathering input and checking the branch, energy is exhausted. On reboot, execution resumes from line 1, re-executing the input and seeing the value six. The trace sets `h` and increments `c`. Compared to JIT checkpoints, atomic regions may impose higher time overhead, due to state management and re-execution. Some systems ask the programmer to define region boundaries [Hester et al. 2017; Maeng et al. 2017; Ruppel and Lucia 2019], while others [Maeng and Lucia 2018; Van Der Woude and Hicks 2016] automatically place boundaries to minimize recovery cost. Manual placement requires more programmer effort, but avoids problems related to peripherals and timely use of data, as a programmer is likely to know which code must execute atomically (e.g., spin-loops awaiting peripheral results). The example in Figure 1 illustrates this benefit of atomic region placement. Atomic execution makes its branching decision at line 2 using fresh data always. In contrast, JIT execution may incur a power failure between lines 1 and 2, causing an arbitrary delay and potentially rendering the input value `t` stale before its use on line 2.

Recent work [Maeng and Lucia 2019; Surbatovich et al. 2021] combines these two models, using atomic regions only for correct and timely interaction with peripherals and inputs, and low-overhead JIT checkpoints otherwise. Curricle adopts this "JIT + atomics" intermittent execution model.

Maintaining Idempotent Execution: the *What*. A system should produce only intermittent executions that are equivalent to some continuous execution [Surbatovich et al. 2020]. A JIT scheme must save enough state to restore to the point of power failure, typically the registers and stack. Atomic schemes must save enough state to restart the current atomic region *and* to prevent re-execution from producing *non-idempotent* results, i.e., the registers and stack at the start of the region, as well as a set of memory locations that includes those involved in WAR or RIO access patterns. Figure 2 illustrates how a WAR dependence or repeated input operation causes non-idempotence, showing an execution trace and values of non-volatile variables. All variables start with value 0 (NVM0). The top trace almost completes the region before energy is exhausted, setting both `l` and `c` to 1. The re-execution starts with these values persisted in non-volatile memory (NVM0'). As the input value on re-execution is 6, the lower trace takes the if branch and sets `h`, causing a RIO bug; both flags (`h` and `l`) are set, which is impossible in a continuous execution. On incrementing `c`, the execution reads the prior value 1 and updates to 2, causing a WAR bug.

Undo, Redo, Up-front, or On-Demand: the *How*. Prior work has explored a variety of recovery implementations, including undo logging (revert changes at reboot) [Kortbeek et al. 2020b; Lucia and Ransford 2015; Surbatovich et al. 2021], redo logging (commit changes at region completion) [Maeng et al. 2017; Ruppel and Lucia 2019], and channels (dataflow edges) [Colin and Lucia 2016; Hester et al. 2017]. Some identify all potential locations to restore ahead of time [Lucia and Ransford 2015],

others at runtime [Kortbeek et al. 2020b]. These designs offer different overhead trade-offs and optimizing logging overhead is orthogonal to identifying the set of data to recover for correctness.

2.2 Past Approaches Limit Programmers' Choices, Incurring Unnecessary Overheads

Few systems take the same approach to recovery, and systems do not formally express their recovery actions, leaving programmers with little assurance their program will execute as intended.

Lacking Principled Non-Idempotence. Some programs *need* non-idempotence to function correctly. A system might count re-executions, or an application might track partial sensor readings, even if they were interrupted by power failures. However, most prior systems assume that *everything* must re-execute idempotently, providing no explicit mechanism for non-idempotent operations. Prior efforts [Ma et al. 2017; Maioli and Mottola 2020] explore using data from partial executions for optimization, but lack a way to express non-idempotence and lack a correctness definition for programs that use data manipulated non-idempotently.

Monolithic Design Leads to Unnecessary Overheads. Today, intermittent system designers must determine what should be recovered for correctness and then design actual recovery mechanisms. Coupling these concerns creates an undesirable trade-off of correctness for performance. The simplest approach taken by prior work, including state-of-the-art systems, recovers all written data in an atomic region [Kortbeek et al. 2020b; Maeng and Lucia 2020; Ruppel and Lucia 2019]. This approach is simple and often correct (although *in*correct in code that requires non-idempotent operations) but has high overhead due to unnecessary logging of some written data. To reduce overheads, some systems use dataflow analysis [Lucia and Ransford 2015; Maeng et al. 2017; Surbatovich et al. 2021, 2020] to identify and log only variables involved in WARs and RIOs. To be correct, these analyses must be conservative w.r.t. to memory aliasing, which is challenging in C code that allows arbitrary pointer manipulation. To avoid needlessly conservative data recovery, some systems disallow or limit use of pointers, but do not actually check that the programmer is following the these restrictions, allowing silent failures. Samoyed [Maeng and Lucia 2019] offloads reasoning to the programmer, requiring annotations on WAR variables. However, the underlying system assumes the programmer is correct and does not check the annotations against the program. If the programmer under-specifies, the program will not run correctly. If the programmer over-specifies, it will cause overhead. Moreover, Samoyed does not consider RIOs, allowing idempotence violations.

3 AN OVERVIEW OF CURRICLE

Curricle overcomes the above-mentioned shortcomings of prior approaches. Curricle allows application programmers to specify idempotence policies via types and provides clean abstractions to relieve system designers from the burden of reasoning about idempotent accesses, as illustrated in Figure 3. Along with checking that non-idempotent (Nid) variables do not interfere with idempotent (Id) ones, Curricle provides a type inference algorithm that detects which Id variables have WAR and RIO patterns, forming a *recovery list*. Curricle passes this list to the underlying recovery system, which is connected by an abstract interface consisting of recovery APIs and requirements on these APIs. Moreover, Curricle leverages Rust's support for tracking unique ownership to ensure analysis soundness, as Rust enforces alias restrictions that past intermittent systems assume, but do not check, in C-based programs. We next provide a high-level intuition of Curricle's type system design and describe how Curricle benefits application programmers and intermittent system designers.

The Basics of Curricle Type System Components and Interactions. The Curricle type system has two key objectives: it must check that any data the programmer typed as non-idempotent does not flow to idempotent-typed data, and it must determine whether any idempotent-typed data must be recovered by the system to indeed be accessed idempotently. To realize these objectives,

Curricle relies on three key components: idempotence qualifiers, access mode qualifiers, and taintedness qualifiers. First, the top-level idempotence qualifiers Id and Nid are *checked* as standard information flow qualifiers, based on an integrity lattice [Biba 1977; Denning 1976]; non-idempotent data is untrusted, idempotent data is trusted, and any operation that causes untrusted data to influence trusted data, e.g., $x := y$, where x is type Id and y is type Nid, will be rejected. Secondly, each Id qualifier has an interior access mode qualifier that (a) describes why a variable is idempotent and (b) limits the accesses allowed to the variable. For instance, if a variable has a read-only qualifier Rd, this means that a variable is at most read, so cannot cause non-idempotence, and that a write to this variable should fail to type check. If a variable is read then written, as with the operation $c := c+1$ in Figure 2, the only way for the variable to be idempotently accessed is for it to be checkpointed, so it must have the access mode Ck to type check. As the access mode depends on potentially subtle dataflow and input dependencies and is not part of the higher level idempotence specification, Curricle provides a flow-sensitive *inference* algorithm to determine the access qualifier of each Id variable, along with a checking algorithm that confirms an access mode solution accurately describes the program. Finally, as the non-idempotent RIO access pattern involves input dependences, the type system needs the taintedness qualifiers Tnt and Nt to *track* whether a variable is currently dependent on an input operation (i.e., tainted). Defining the interactions between the checking, inferring, and tracking algorithms forms some of the key challenges and novelty behind Curricle.

Curricle Gives Programmers Correctness and Control. Curricle enables programmers to specify *intended* non-idempotence with its information flow types and prevents *unintended* non-idempotence from WAR and RIO bugs by capturing a variable's access pattern in the type with access modes. Using Curricle thus benefits programmers as Curricle checks that a programmer's idempotence requirements are satisfied, while automatically determining the necessary recovery list of each atomic region. To use Curricle, the programmer specifies which functions denote atomic regions and which variables should be *non-idempotent* (all others are typed as idempotent, the common case). Curricle next infers the interior access mode of each Id variable, only inferring Ck if there is no other solution, minimizing the recovery list. Curricle then reports this inferred recovery list to the programmer, as well as any type errors caused by information flowing from Nid types to Id types. Allowing the programmer to declare variables as non-idempotent while ensuring that non-idempotence does not effect idempotent variables is a novel feature of Curricle and is a key advantage of the language-level approach. Once a program compiles with no type errors, the program will run correctly intermittently on any runtime that meets the properties of the defined system abstraction.

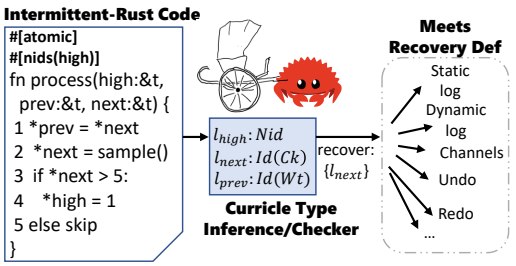

Fig. 3. Curricle defines an abstraction between programs and the recovery system

Targeting Curricle Simplifies Recovery System Design. Curricle separates reasoning about a program's idempotence requirements from the mechanics of restoration, reducing burden on intermittent system designers and enabling them to write simpler, more performant systems. Curricle's correctness relies on the existence of a recovery system that meets a set of correctness properties. We extract three core capabilities of a recovery system: Setup, Recover, Finalize. Setup creates a recovery context that the system will use to restore state power failure, aided by Recover. Finalize ensures that a previous recovery context can be safely discarded and a new one created. These routines should ensure that for some given variable, the first access on re-execution matches

Values	v	::=	$n \mid \text{true} \mid \text{false} \mid \&x \mid \&p.n$	*Restore list*	$ckPts$	::=	$\cdot \mid ckPts, x \mid ckPts, x.n$
Expressions	e	::=	$p \mid v \mid e_1 \odot e_2 \mid \oslash e$	*Non-id list*	$nIds$	::=	$\cdot \mid nIds, x \mid nIds, x.n$
Place expr.	p	::=	$x \mid *p \mid p.n$	*Programs*	$prog$	::=	$\text{halt} \mid seg;; prog$
Segments	seg	::=	$c \mid \text{start}_{\text{atom}}(aID, ckPts, nIds, \varsigma); c; \text{end}_{\text{atom}}$				
Commands	c	::=	$\text{skip} \mid p := e \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid c_1; c_2 \mid \text{let } x = \{v_0, ..., v_n\} \text{ in } c \mid \text{let } x = f(v) \text{ in } c$				
		|	$\text{let } x = e \text{ in } c \mid \text{let } x = \text{IN}() \text{ in } c \mid \text{let } f : \text{ft} = \text{cap}(\Delta)\ \lvert arg \rvert \{c'; ret := e\} \text{ in } c$				

Fig. 4. Syntax

what the access would be on the initial execution, i.e., is idempotent. Which variables need to be recovered to satisfy the idempotence requirements of the *entire program execution* is dependent on the programmer's desires combined with the access patterns of the program. Curricle gives this recovery list, freeing the system designer from doing complex analyses and requiring them only to provide mechanisms to back-up and restore variables.

4 LANGUAGE SYNTAX AND SEMANTICS

We present a core language for the JIT + Atomic region model of intermittent execution, with basic Rust features, such as references, structs, and closures. The language semantics rely on a recovery system, abstractly specified as a set of APIs and properties that the APIs must satisfy.

4.1 Syntax and Runtime Constructs

The syntax of our core calculus is shown in Figure 4. A value v is an integer, boolean, or reference. An expression e can be a place expression, value, or an operation on further expressions. A place expression p names a memory location by variable, a dereference of p, or a field access on p. A program *prog* consists of segments (*seg*) sequenced by the double semi-colon ;;, which are either atomic regions or commands. An atomic region is parametrized with an ID, a recovery list (*ckPts*), a set of variables that can be non-idempotent (*nIds*), and an access mode solution ς (used for typing and proofs). A command c includes assignments, branches, function calls, mutable let bindings, input operations, struct initialization, closure definitions, and further commands, sequenced by a single semi-colon. A closure definition is written as $f = \text{cap}(\Delta)\ \lvert arg \rvert \{c'; ret := e\}$, where Δ is the set of captured variables, and *arg* is the function parameter. When Δ is empty, this reduces to a function definition. To simplify our model, our syntax does not allow unbounded loops or recursion, which do not add interesting features to the type system and are disallowed in atomic regions by many intermittent systems due to their unpredictable energy cost. Furthermore, while unbounded loops are common in general embedded systems programming (e.g., for blocking I/O), recursion can be expensive to support on memory constrained devices due to the unpredictability of the stack size. Our model also does not *explicitly* model Rust features like mutability and ownership tags, as they are orthogonal to intermittent computation, which our type system targets. Such features are supported by the implementation, however, as it targets real Rust programs, which also pass Rust's type checking. We further discuss supporting Rust code and limitations in Section 8.

Variables	var	::=	$x \mid x.n$	*Mem. loc.*	loc	::=	$\ell \mid \ell.n$
Mem. value	val	::=	$n \mid \text{true} \mid \text{false} \mid loc$	*NV-mem*	N	::=	$\cdot \mid N, loc \mapsto val@(q, wtflag)$
Stack-mem	V	::=	$\cdot \mid V, loc \mapsto val@q$	*Prog. counter*	pc	::=	$\langle \text{Id}, qIO \rangle \mid \langle \text{Nid}, qIO \rangle$
PC stack	pcS	::=	$pc \rhd \cdot \mid pc \rhd pcS$	*Recovery ctx*	κ	::=	$(N, S, c) \mid (V, S, c, ckPts) \mid \cdots$
Mode counter	mc	::=	$\text{jit} \mid \text{atom}$	*Int. prog config.*	Σp	::=	$(\kappa, mc, N, M_N, S, prog, seg)$
Cont. config.	σc	::=	(N, S, M_N, seg)	*Int. cmnd config.*	Σc	::=	$(\kappa, mc, N, M_N, S, seg)$
Variable map	M	::=	$\cdot \mid M, var \mapsto loc \mid M, f \mapsto \text{cap}(\Delta)\ \lvert arg \rvert c; ret := e, M_f, V_f)$				
Exec. context	E	::=	$\cdot \mid \text{endIf} \rhd E \mid [\,]; c \rhd E \mid \text{let } x = [\,] \text{ in } c \rhd E$				
Stack	S	::=	$(pcS, M, V, E) \rhd_f \cdot \mid (pcS, M, V, E) \rhd_f S$				
Observation	o	::=	$\text{rb} \mid \text{in}(v@q) \mid \text{pwFail} \mid \text{atomB} \mid \text{jitB} \mid \text{mstate}(N, V)$				
Steps (Energy) Left	ε	$\in$	$\mathbb{N}$	*Logic Energy Inputs*	$\mathcal{E}$	::=	$\varepsilon :: \mathcal{E}$
Observations	O	::=	$\cdot \mid o :: O$	*Sensor Inputs*	$\mathcal{I}$	::=	$v :: \mathcal{I}$

Fig. 5. Runtime Constructs

The runtime constructs used in defining our operational semantics are shown in Figure 5. Non-volatile memory N maps a location to a value, a qualifier, and a write flag *wtflag*, while volatile memory V maps a location to a value and a qualifier. We delay explaining these qualifiers until the next section as they are only used for proofs. We assume that an n-element record is stored as a contiguous set of locations $\ell.1$ to $\ell.n$. A map M maps variables to their memory locations and function names to their definition. We write S to denote the stack. A stack frame is added at a function call and contains a pc stack pcS (for nested if statements), stack memory V and the local map M. To map variables to nonvolatile locations, we use a pre-initialized map M_N that is given as a separate parameter in the intermittent configuration. This way, the stack pertains entirely to volatile memory. The embedded devices we target statically allocate variables in non-volatile memory, so we assume N and M_N are initialized with data and the correct variable-to-location mappings. The last member of the stack tuple is a stack of execution contexts E for keeping track of execution order. let $x = [\,]$ in c denotes the continuation of a function call. The result of the function call will be bound to x and the execution will continue from c. $[\,]; c$ is the context for evaluating sequences. The program always starts from the stack frame of the main function, where the bottom of the stack is $S^0 = (\langle \mathsf{Id}, \mathsf{Nt} \rangle, \cdot, \cdot, \cdot)$ with empty contexts.

An intermittent configuration Σp contains the runtime state of the program. Σc denotes a command configuration (abbreviated as Σ), which contains a recovery context κ, the current execution mode mc (JIT or atomic), and the current command c. The contents of the recovery context depends on the underlying recovery system implementation—an instantiation is shown in Section 7. A continuous configuration σc is (N, S, M_N, seg) as recovery metadata is not needed. We provide the program with a stream of sensor inputs: $\mathcal{I}$, which gets a value removed and read on an input command execution, and a *logical* energy input stream $\mathcal{E}$. Each element of $\mathcal{E}$, denoted ε, is a natural number indicating how many steps remain before power fails. Finally, an execution generates observations, which include power failures pwFail, reboots rb, sensor inputs, transitions between atomic and jit modes, and memory snapshots at the end of each program segment.

4.2 Operational Semantics

Continuous and intermittent execution semantic rules are of the form $\mathcal{I} \mid \sigma \xrightarrow{O} \mathcal{I}' \mid \sigma'$ and $\mathcal{E}, \mathcal{I}, \varepsilon \mid \Sigma \overset{O}{\Longrightarrow} \mathcal{E}', \mathcal{I}_1', \varepsilon_1' \mid \Sigma'$, respectively. Their semantic rules are almost the same, except that intermittent executions use logical energy and have additional rules for recovery from power failures. All the rules are standard and can be found in Section F of the supplementary material.

An intermittent execution relies on a recovery system R to handle power failure. We formally define a recovery system R as a pair of an API list and a property list, written $R = (\mathsf{API}, \mathsf{Prop})$. We say R is correct if all APIs in API satisfies Prop, detailed in Section 4.3. We identify the following APIs for Curricle: {lookUpN, updateN, Setup, Recover, Finalize}. When accessing the non-volatile memory, the functions $\mathsf{lookUpN}((\kappa, N'), loc)$ and $\mathsf{updateN}((\kappa, N), loc \mapsto v)$ account for the memory state potentially kept in the recovery context κ. When the rest of the functions are invoked depends on the current execution mode. Executing in jit mode recovers to the point immediately before the last power failure, whereas executing in atomic mode recovers to the start of the atomic region. We explain the key rules shown in Figure 6, which use the mode counter mc to distinguish between jit and atomic execution. On transitioning from jit to atomic mode, the system finalizes the previous recovery point and sets up a new recovery point at the start of the atomic region (rule StartAtom). Each command-level transition decreases, ε, the number of energy steps remaining, by 1, with a power failure represented by ε reaching 0. If power fails during atomic mode execution, the system directly transitions to the reboot command (rule PowerFailAtom). When transitioning from atomic mode to jit mode, the system finalizes the atomic region recovery point. While executing in jit

mode, the system finalizes the previous point and sets up a new recovery point on low power (rule PowerFailJit). Both rules clear the volatile memory, denoted S^0, M_N^0. On reboot in either mode (rule Reboot), the system recovers to the stack, memory, and command returned by the Recover routine, and "recharges" by popping the next segment of energy ε from the energy input stream $\mathcal{E}$.

$$\frac{\mathsf{Finalize}(N,\kappa) = N' \qquad \mathsf{Setup}(\mathsf{atom}, \mathit{ckPts}, \mathit{nIds}, N', M_N, S, \mathit{prog}, c) = (\kappa', N'')}{\mathcal{E}, \mathcal{I}, \varepsilon \mid \kappa, \mathit{jit}, N, M_N, S, \mathit{prog}, \mathsf{start}_{\mathsf{atom}}(\mathit{aID}, \mathit{ckPts}, \mathit{nIds}, \varsigma); c; \mathsf{end}_{\mathsf{atom}} \overset{\mathsf{atomB}}{\Longrightarrow} \mathcal{E}, \mathcal{I}, \varepsilon - 1 \mid \kappa', \mathit{atom}, N'', M_N, S, \mathit{prog}, c; \mathsf{end}_{\mathsf{atom}}} \textsc{StartAtomic}$$

$$\frac{\mathsf{Recover}(\mathit{mc}, \kappa, N, S) = \kappa', N', M_N', S', \mathit{prog}, c}{\varepsilon :: \mathcal{E}, \mathcal{I}, 0 \mid \kappa, \mathit{mc}, N, M_N, S, \mathsf{reboot}, \mathsf{reboot} \overset{\mathsf{rb}}{\Longrightarrow} \mathcal{E}, \mathcal{I}, \varepsilon \mid \kappa', \mathit{mc}, N', M_N', S', \mathit{prog}, c} \textsc{Reboot}$$

$$\frac{}{\mathcal{E}, \mathcal{I}, 0 \mid \kappa, \mathit{atom}, N, M_N, S, \mathit{prog}, c; \mathsf{end}_{\mathsf{atom}} \overset{\mathsf{pwFail}}{\Longrightarrow} \mathcal{E}, \mathcal{I}, 0 \mid \kappa, \mathit{atom}, N, M_N^0, S^0, \mathsf{reboot}, \mathsf{reboot}} \textsc{PowerFailAtom}$$

$$\frac{\mathsf{Finalize}(N,\kappa) = N' \qquad \mathsf{Setup}(\mathsf{jit}, \emptyset, \emptyset, N', M_N, S, \mathit{prog}, c) = (\kappa', N'')}{\mathcal{E}, \mathcal{I}, 0 \mid \kappa, \mathit{jit}, N, M_N, S, \mathit{prog}, c \overset{\mathsf{pwFail}}{\Longrightarrow} \mathcal{E}, \mathcal{I}, 0 \mid \kappa', \mathit{jit}, N'', M_N^0, S^0, \mathsf{reboot}, \mathsf{reboot}} \textsc{PowerFailJit}$$

Fig. 6. Selected semantic rules, simplified to elide the typed memory

4.3 Requirements of the Recovery System

The correct execution of well-typed Curricle programs also relies on the correctness of the recovery system $R = (\mathsf{API}, \mathsf{Prop})$. We now explain the properties in groups of related operations.

We discuss what Prop includes below. The setup and recovery routines should guarantee that the system recovers to the last setup point, restoring execution context and the values for checkpointed variables. In other words, given the pair of functions $\mathsf{Setup}(\mathit{mc}, \mathit{ckPts}, \mathit{nIds}, N, M_N, S, \mathit{prog}, c) = \kappa', N'$ and $\mathsf{Recover}(\mathit{mc}, \kappa_1, N_1, S_1) = \kappa_r, N_r, M_{nr}, S_r, \mathit{prog}_r, c_r$ the following properties must hold:

- **(SP1)** $\forall \mathit{loc} \in \mathsf{dom}(\kappa', N'), \mathsf{lookUpN}((\cdot, N), \mathit{loc}) = \mathsf{lookUpN}((\kappa', N'), \mathit{loc})$
- **(SP2)** $c = c_r$, $\mathit{prog} = \mathit{prog}_r$, $S = S_r$ and $M_n = M_{nr}$
- **(SP3)** $\forall \mathit{loc} \in \mathit{ckPts}, \mathsf{lookUpN}((\kappa_r, N_r), \mathit{loc}) = \mathsf{lookUpN}((\kappa', N'), \mathit{loc})$
- **(SP4)** $\forall \mathit{loc} \in \mathsf{dom}(\kappa_r, N_r) \notin \mathit{ckPts}$, if $\mathsf{lookUpN}((\kappa_1, N_1), \mathit{loc}) = v_1$ and $\mathsf{lookUpN}((\kappa_r, N_r), \mathit{loc}) = v_r$ where $v_1 \neq v_r$ then $v_r = \mathsf{lookUpN}((\kappa', N'), \mathit{loc})$
- **(SP5)** $\forall \mathit{loc} \in \mathit{nIds}, \mathsf{lookUpN}((\kappa_1, N_1), \mathit{loc}) = \mathsf{lookUpN}((\kappa_r, N_r), \mathit{loc})$

Intuitively, these properties mean that (SP1) the setup routine cannot arbitrarily change values, and after recovery (SP2) the command, stack, and memory mappings return to their values at the last setup point, (SP3) looking up the value stored at locations in *ckPts* must be equivalent to looking up the location from the initial state, (SP4) if the value for any other location has changed, the new value must be equal to the value of the initial state, and (SP5) the value for location in *nIds* should not be reset. Notice for (SP3) that the definition does not enforce *where* the value is stored, e.g., in a log or in the main non-volatile memory, allowing for many implementations. Property (SP4) allows the recovery system to change locations other than those in *ckPts*, necessary for more conservative implementations, but constrains it from arbitrarily changing values. Property (SP5) means that variables declared as non-idempotent should not be reverted by the system.

A finalize routine must ensure that after finalizing a recovery context, the system no longer needs the past recovery context κ to look up values, and the functions $\mathsf{lookUpN}((\kappa, N), \mathit{loc})$ and $\mathsf{updateN}((\kappa, N), \mathit{loc} \mapsto v)$ must ensure that after an update, looking up returns the updated value:

Base type	b	::=	int | bool	*Qualified type*	tq	::=	$b@q \mid st@q \mid rt@q$
Type qual.	q	::=	$\langle qId, qIO \rangle$	*Struct type*	st	::=	$\{s_0 \ldots s_n\}$
Taint qual.	qIO	::=	Tnt | Nt	*Simple type*	s	::=	$b@q \mid rt@q$
Idemp. qual.	qId	::=	Id($qAcc$) | Nid | Id | Id(α)	*Reference type*	rt	::=	$\&b@q \mid \&rt@q \mid \&st$
Access qual.	$qAcc$	::=	$\mathsf{Wt} \mid \mathsf{Rd} \mid \mathsf{Ck} \mid \mathsf{Wt} \oplus \mathsf{Rd} \mid \mathsf{Wt}^t \oplus \mathsf{Rd} \mid \mathsf{Wt} \oplus \mathsf{Rd}^t \mid \mathsf{Wt}^t \oplus \mathsf{Rd}^t$				
Types	τ	::=	$\mathsf{fn}(tq) \xrightarrow{pc} tq' \mid tq$				

Fig. 7. Type Constructs

(FP1) $\mathsf{Finalize}(N, \kappa) = N'$ implies $\forall loc \in dom(N, \kappa), \mathsf{lookUpN}((\kappa, N), loc) = \mathsf{lookUpN}((\cdot, N'), loc)$
(RW1) $\mathsf{lookUpN}(\mathsf{updateN}((\kappa, N), loc \mapsto v), loc) = v$

5 THE CURRICLE TYPE SYSTEM

Curricle's type system rules out programs that introduce non-idempotence when running intermittently by augmenting data with idempotence and taint qualifiers, using type rules to determine allowed information flow and the access mode of variables. First, we explain the types in more detail. Then, we show how to type check a program with idempotence qualifiers and access modes to solidify the intuition behind the types. Finally, we explain the type inference algorithm.

5.1 Typing Constructs and Running Example

Figure 7 shows the typing constructs, which include base types int and bool, and struct and reference types. The right column shows the nesting of the types when creating structs and references. To check idempotence and input taintedness, Curricle qualifies the type of expressions with a pair $\langle qId, qIO \rangle$. Note that the interior types of the struct carry their own qualifier pair.

The *qIO* qualifier denotes whether data is tainted by an input value: tainted data is typed Tnt and un-tainted Nt. The *qId* qualifier denotes whether a variable can have different (non-idempotent) values across re-executions (Nid), or must remain idempotent (Id). In addition, the idempotent qualifier *qId* denotes *why* a variable is idempotently accessed with qualifier *qAcc*. A variable can be idempotent because it must first be written to on any execution, thereby clearing any result from a partial execution, Wt. Alternatively, a variable could be at most read, so the value never changes, Rd. If a variable has a WAR or RIO access pattern, the only way for this variable to be idempotent is for the original value to be checkpointed, Ck. Finally, if a variable is either first written or at most read, but the decision is deterministic (i.e., not input dependent), the variable has access mode $\mathsf{Wt} \oplus \mathsf{Rd}$. This $\oplus$ type can have one or both sides tagged to indicate whether a leading read or write access has occurred. Note that the type $\mathsf{Wt}^t \oplus \mathsf{Rd}^t$ is only possible statically, as the type system must consider all possible first accesses from a branch. At runtime, only one path (and one initial access) can be ever be taken. Id can also contain a type variable α, used by Curricle's inference algorithm to determine access modes. Finally, the top level type τ consists of qualified types tq and function types. The *qId* in a function type does not contain any access mode or type variable.

A Running Example. To explain type checking and inference, we will use the running example in Figure 8. The example shows an atomic region processing a sensor read, using the syntax of the implementation, which hews closer to Rust syntax. The programmer has marked the atomic region but not declared any variables as Nid. The program copies `*next` into `*prv` and gathers new input into `*next`. If the input value was greater than 5, it sets `*high`. Finally, if `*prv` was less than some floor, `*prv` gets set to `*fl`, which otherwise gets cleared. We refer to the dereferenced locations of `next`, `prv`, `high` and `fl` as n, p, h, and f, henceforth. Because n is involved in a WAR pattern and h is involved in a RIO, both locations must be checkpointed to maintain idempotence (access mode Ck). On the other hand, p is always first written, so it has mode Wt, even though lines 6–7 read then write to p. Finally, f is either always written or always read, based on a deterministic branch value, so it has mode $\mathsf{Wt} \oplus \mathsf{Rd}$.

Information Flow Lattices. Curricle uses two information flow lattices and defines a partial order for access modes. The first lattice is for idempotence qualifiers and is used to prevent non-idempotent values from influencing idempotent ones, with the partial order defined as Id(_) ⊏ Nid. The second lattice is for *qIO* with Nt ⊏ Tnt, to track input tainting through the program execution. The partial order $qAcc_1 \prec_a qAcc_2$ indicates that the type system poses less access restrictions on a location qualified by $qAcc_1$ than $qAcc_2$. For example, if an access mode is Ck, any access, in any order, is allowed, but if an access mode is Rd, the location can only be read and not written. It is defined as $\mathsf{Ck} \prec_a \mathsf{Wt} \sim_a (\mathsf{Wt}^t \oplus \mathsf{Rd}) \prec_a (\mathsf{Wt} \oplus \mathsf{Rd})$ and $\mathsf{Ck} \prec_a \mathsf{Rd} \sim_a (\mathsf{Wt} \oplus \mathsf{Rd}^t) \sim_a (\mathsf{Wt}^t \oplus \mathsf{Rd}^t) \prec_a (\mathsf{Wt} \oplus \mathsf{Rd})$. The type Wt ⊕ Rd is the top element because it cannot have had any accesses made to it; when an initial write or read access occurs, the respective mode gets tagged and the type changes. A tagged Rd makes the type equivalent to Rd, regardless of whether the write tag is set, and an exclusively tagged Wt is equivalent to Wt.

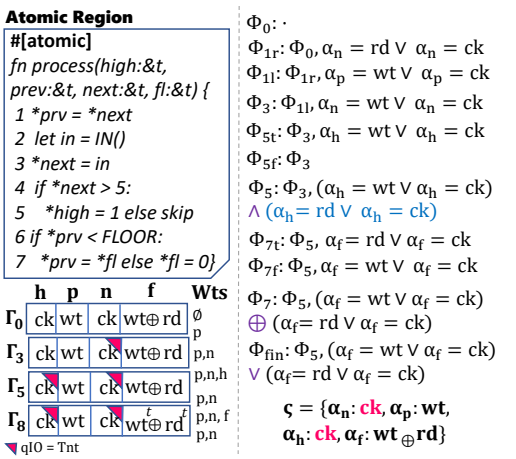

Fig. 8. Code with type contexts and constraints

5.2 Type Checking Access Modes

Type checking confirms that an access mode assignment describes the program. We first describe the type checking judgments and then step through the checking algorithm for the example in Figure 8, using the selected rules in Figure 9.

Typing Contexts and Judgments. The memory typing context Γ maps abstract locations l and function names f to their respective types. The points-to map Π maps place expressions (e.g., variable names) to abstract locations, as well as function names to their declaration. A variable's type can be looked up as $\Gamma(\Pi(x))$. Assuming a linear type system like Rust's ownership typing beneath Curricle, a place-to-location mapping can be computed syntactically and is almost always a singleton set.[1] As an example, if a variable is borrowed, $x := \&y$, the points-to map updates the entry of $*x$ to be $\Pi(y)$. The points-to map can change if, e.g., the value of a reference type changes. Additionally, the checking rules calculate a must write set of locations, Wts.

$$\begin{array}{llll} \textit{Type Context} & \Gamma & ::= & \cdot \mid \Gamma, l : \tau \mid \Gamma, f : \tau \\ \textit{Points-to Map} & \Pi & ::= & \cdot \mid \Pi, p : l \mid \Pi, f : \mathrm{cap}(\Delta)\ |arg|\{c; ret := e\} \end{array} \qquad \begin{array}{llll} \textit{Writes} & \mathsf{Wts} & ::= & \cdot \mid \mathsf{Wts}, l \end{array}$$

Expression typing judgments are of the form $\Gamma, \Pi \vdash_e e : \tau \Rightarrow \Gamma'$, meaning given a typing context and a points-to map, evaluating expression e yields a value of type τ and an updated typing context Γ'. Command typing judgments are of the form $\Gamma, \Pi, \mathsf{Wts}, pc \vdash c \Rightarrow \Gamma', \Pi', \mathsf{Wts}'$. The program counter pc indicates the qualifiers of the outer branching condition. Executing c changes the points-to map to Π' and the must-write set to Wts′. The lower left of Figure 8 shows the typing contexts for the dereferenced locations, with Γ_n denoting the context after line n, showing only the access mode for space. A pink tag in the corner means that the location's taintedness qualifier is set to Tnt.

Checking Reads and Writes. Rules LocUnwritten and LocWritten check read accesses. The notation $qId \uparrow^{qAcc}$ retrieves the access mode of a type, if one exists. Reading a location l of type

[1]The rules in Appendix Sections C and D handle the multi-element case; since it complicates the typing rule presentation and is rare in practice (it does not occur in our benchmarks), we defer the discussion.

Rd will type check as long as l is not written ($l \notin \mathsf{Wts}$). A read to l of type Wt, however, only type checks if l has already been written, preventing a variable with a WAR access pattern from being typed as Wt. Reading l of type Ck type checks regardless of whether l is in Wts. Rule Assgn checks write accesses. The rule checks that the location's type is one that allows writes ($qAcc \preceq_a \mathsf{Wt}$) and adds the location to Wts. No rule allows writing to locations of type $\sim_a$ Rd, preventing a variable with a WAR access pattern from being typed as Rd. Consider line 1 of the example, which reads n and writes to p. Since n has type Ck and $n \notin$ Wts, the right hand side checks with rule Unwritten. The type of p is Wt, so rule Assgn checks the write and adds p to Wts. Any reads to p, as on line 6, will now type check with rule LocWritten. Continuing, line 2 gets an input, storing it into a stack variable. Rule Let-In sets the qualifier of `in` to $\langle \mathsf{Id}, \mathsf{Tnt} \rangle$ since its value is always reinitialized, but is input dependent. Checking line 3, rule Assgn type checks the write to n as Ck $\preceq_a$ Wt, and updates n's taint qualifier to Tnt, as the expression `in` is tainted.

$$\frac{\Gamma(l) = \langle qId, qIO \rangle \qquad l \notin \mathsf{Wts} \qquad qId \uparrow^{qAcc} \preceq_a \mathsf{Rd}}{\Gamma, \Pi, \mathsf{Wts} \vdash^{atom}_{e} l : t@\langle qId \downarrow, qIO \rangle \Rightarrow \Gamma} \textsc{LocUnwritten}$$

$$\frac{\Gamma(l) = \langle qId, qIO \rangle \qquad l \in \mathsf{Wts} \qquad qId \uparrow^{qAcc} \preceq_a \mathsf{Wt}}{\Gamma, \Pi, \mathsf{Wts} \vdash^{atom}_{e} l : t@\langle qId \downarrow, qIO \rangle \Rightarrow \Gamma} \textsc{LocWritten}$$

$$\frac{\begin{array}{c}\Gamma, \Pi, \mathsf{Wts} \vdash^{atom}_{e} e : \mathsf{bool}@\langle qId_e, qIO_e \rangle \qquad \langle qId_{\mathrm{pc}}, \mathsf{Tnt} \rangle = \langle qId_e, qIO_e \rangle \sqcup pc \\ \Gamma, \Pi, \mathsf{Wts}, \langle qId_{\mathrm{pc}}, \mathsf{Tnt} \rangle \vdash^{atom}_{c} c_1 \Rightarrow \Gamma_1, \Pi_1, \mathsf{Wts}_1 \qquad \Gamma, \Pi, \mathsf{Wts}, \langle qId_{\mathrm{pc}}, \mathsf{Tnt} \rangle \vdash^{atom}_{c} c_2 \Rightarrow \Gamma_2, \Pi_2, \mathsf{Wts}_2 \\ \Gamma_3 = \Gamma_1 \sqcup_T \Gamma_2 \qquad \mathsf{Wts}_{\mathsf{EMW}} = (\mathsf{Wts}_1 \cup \mathsf{Wts}_2) \setminus (\mathsf{Wts}_1 \cap \mathsf{Wts}_2) \qquad \forall l \in \mathsf{Wts}_{\mathsf{EMW}}, \Gamma_3(l) \uparrow^{qAcc} \preceq_a \mathsf{Ck}\end{array}}{\Gamma, \Pi, \mathsf{Wts}, pc \vdash^{atom}_{c} \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \Rightarrow \Gamma_1 \sqcup_T \Gamma_2, \Pi_1 \uplus \Pi_2, \mathsf{Wts}_1 \cap \mathsf{Wts}_2} \textsc{IfTnt}$$

$$\frac{\begin{array}{c}\Pi(p) = l_p \qquad \Gamma, \Pi, \mathsf{Wts} \vdash^{atom}_{e} e : t@q_e \Rightarrow \Gamma' \\ \Gamma', \Pi, \mathsf{Wts} \vdash^{atom}_{elft} p : @q_l \Rightarrow \Gamma'_l \qquad \langle qId_e, qIO_e \rangle = q_e \sqcup \langle qId_{pc}, qIO_{pc} \rangle \sqcup q_l \\ \Gamma'_l(l_p) = t@\langle qId_p, qIO_p \rangle \qquad qId_e \sqsubseteq qId_p \downarrow \qquad qId_p \uparrow^{qAcc} \preceq_a \mathsf{Wt} \qquad \mathsf{UpdatePoint}(\Pi, p, e : t) = \Pi'\end{array}}{\Gamma, \Pi, \mathsf{Wts}, \langle qId_{pc}, qId_{pc} \rangle \vdash^{atom}_{c} p := e \Rightarrow \Gamma'_l[\Pi'(p) : \langle qId_p, qIO_e \rangle], \Pi', (\mathsf{Wts}, \Pi'(p))} \textsc{Assgn}$$

$$\frac{\mathsf{fresh}(l_x) \qquad \Gamma[l_x : \mathsf{Int}@\langle \mathsf{Id}, \mathsf{Tnt} \rangle], \Pi[x : l_x], (\mathsf{Wts}, l_x), pc \vdash^{atom}_{c} c \Rightarrow \Gamma_c, \Pi_c, \mathsf{Wts}_c}{\Gamma, \Pi, \mathsf{Wts}, pc \vdash^{atom}_{c} \mathsf{let}\ x = \mathsf{IN}()\ \mathsf{in}\ c \Rightarrow \Gamma_c, \Pi_c \backslash x, \mathsf{Wts}_c \backslash l_x} \textsc{Let-In}$$

Fig. 9. Selected typing rules, showing checks for reads, writes, branches, and input operations

Branches, Selects, and Joins. Type checking must ensure that if a variable is non-deterministically written to based on input, such a variable must be checkpointed. Rule IfTnt checks branches where the conditional expression is tainted. It sets the pc for the check of each branch arm to the least upper bound of the current pc and the taint of the branching expression and confirms that any variable written on only one of the paths (i.e., the set difference between the union of writes sets and their intersection) has access mode Ck. On lines 4–5, n is tainted, so rule IfTnt applies. As the current pc is tainted, checking the write to h updates its taint qualifier to Tnt and adds h to the write set. The `else` branch does not access h, keeping the write set $\{p, n\}$. Since h is not in both write sets, it must have access mode Ck. Of course, if the programmer decides that h should be set for any seen input, not just the one of the final execution, they can give h the type Nid.

Lines 6–7 contain another branch, this time on a read of p, which is not tainted. Because it does not depend on non-deterministic sensor values, at run time p will always evaluate deterministically on any partial execution. Thus, even though f is only written on one path, and read on the

other, it need not be checkpointed, a situation that cannot be described with the basic Wt and Rd access modes. To avoid throwing out programs unnecessarily, we create the special Wt ⊕ Rd type; handling this type is a key challenge of designing the type checking algorithm. Checking a write of an untagged Wt ⊕ Rd updates the type to $\mathsf{Wt}^t \oplus \mathsf{Rd}$, allowing f to be treated as type Wt within the `else` branch (AssgnSelect, not shown). Checking the `if` branch sets the read tag, treating the type as Rd (LocSelect, not shown). Crucially, joining these now differing type context *accumulates* any tags; after the branch f has type $\mathsf{Wt}^t \oplus \mathsf{Rd}^t$. This type is $\sim_a$ Rd, capturing that the remainder of the atomic region should at most read to f. If the program had a following write to f, then taking the `if` path would result in a WAR access and would not type check.

Finally, the top level rule for an atomic region checks that all variables with type Wt, in this case p, are in Wts, confirming that they *must* be written.

5.3 Inferring Access Modes

Curricle's inference algorithm infers access modes for programmers, generating a per-atomic-region access mode solution ς that will type check if used to initialize the typing context of a region. We describe the inference algorithm at a high level and provide the intuition for why inferred solutions are sound w.r.t. the type checking judgments. The inference judgments require the additional contexts listed below. The type variable context A maps an abstract location l to a type variable α, where $\Gamma(l) = \mathsf{Id}(\alpha)$. The constraint context Φ is a list of constraints φ, interpreted as being joined by $\wedge$, which can be an equality between a type variable and an access mode, a $\wedge$ of two constraints, a $\vee$, or a special "lazy" operator and constraint $\oplus_\alpha$ and $\lfloor\alpha\rfloor$, which we explain shortly. The inference judgments are then of the form $\Phi, A, \Gamma, \Pi, \vdash_e^{inf} e : t@q \Rightarrow \Phi'$ and $\Phi, A, \Gamma, \Pi, pc \vdash_c^{inf} cmd \Rightarrow \Phi', \Gamma', \Pi'$ where Φ' records any of the constraints imposed by expression e or command c. The key operations that constrain the access mode of a variable are reads, writes, and branch joins. We describe the constraints generated by each operation, showing the Φ_n on the right of Figure 8.

$$\begin{array}{llll} \textit{TVar Context} & A & ::= & \cdot \mid A, l : \alpha \quad \textit{Constraint Context} \quad \Phi ::= \cdot \mid \Phi, \varphi \\ \textit{Constraints} & \varphi & ::= & \alpha = qAcc \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \oplus_\alpha \varphi_2 \mid \lfloor\alpha\rfloor \end{array}$$

Reads and Writes. On the read of n on line 1, α_n is not in the domain of Φ_0, so the judgments determine it is the *first* access and constrain α_n to be Ck or Rd (Φ_{1r}). On the write to p, α_p is constrained to be Ck or Wt (Φ_{1l}). On line 3, n is written to, similarly restricting α_n to be Ck or Wt (Φ_3). Note now that the only way to solve the constraints on α_n, joined by $\wedge$, is α_n = Ck. In this way, any variable with a WAR can only have access mode Ck. Next, when n is read on line 4, α_n is already in the domain of Φ_3; this read cannot be the first access, and the judgments impose no further constraints. Looking to the read of p on line 6, the fact that a read imposes no constraints when α_p is already in the constraint set matches the intuition that a Wt-typed variable can freely be read once written; α_p can still solve to Wt after the read as p need not be checkpointed, unlike n.

Branches and Joins. Just as checking uses a different rule depending on the taintedness of the branch expression, inference uses two different joins, $\wedge^{\mathsf{Tnt}}$ and $\wedge^{\mathsf{Nt}}$. For example, the write to h on line 5 constrains α_h to be Ck or Wt (Φ_{5t}), but the `else` branch imposes no constraints (Φ_{5t}). As the expression on line 4 is tainted, Φ_{5t} and Φ_{5f} are joined with $\wedge^{Tnt}$. This join differs from a standard $\wedge$ only in that, if α_h is not in the domain of one operand (Φ_{5f}), the join adds in the constraint that α_h is Ck or Rd to fill in the empty side. The effect of this join is that if only one operand contains a write constraint (α_h = Ck or Wt), the only solution for the resulting set (Φ_5) is Ck. However, this join is too strict for the non-tainted branch on lines 6–7. While taking the `if` branch constrains α_f to be Ck or Rd (Φ_{7t}), and taking the `else` constrains it to be Ck or Wt (Φ_{7f}), f need not be checkpointed. Thus, we introduce a lazy operator, $\oplus_\alpha$, which allows solving to the Wt ⊕ Rd type in certain situations. The intuition of $\oplus_\alpha$ is that if the atomic region maintains the invariant that

a variable must be first written or is at most read, then the constraints can be joined with $\vee$, not $\wedge$. To handle cases where a non-tainted branch does not access the variable at all on one side, we introduce a lazy constraint, $\lfloor\alpha\rfloor$. The intuition of $\lfloor\alpha\rfloor$ is that it forms a "hole" that should be filled on the next access to α, which would be the first access on some path. If that access is a write, $\oplus_\alpha$ is replaced with the standard $\wedge$, but if the inference reaches the end of the atomic region with no further writes, then $\oplus_\alpha$ is replaced with $\vee$ (Φ_{fin}). To solve the constraints, Curricle actually does not return the most general solution according to the access mode lattice, as such as solution would return Ck, the bottom element, wherever possible, e.g., for $\alpha_p = \mathsf{Ck} \vee \alpha_p = \mathsf{Wt}$. Instead, Curricle returns the highest solution possible, to minimize the recovery list. Notice that for α_f, which can be Ck or Wt or Rd, due to the final operator being $\vee$, this solves to the top-most element of the lattice $\mathsf{Wt} \oplus \mathsf{Rd}$, successfully recording that f is exclusively first written or at most read.

Soundness. We implement the type inference algorithm but prove non-interference over the checking algorithm. To connect the two, we prove a soundness theorem for the inference algorithm. The theorem states that if the inference algorithm generates an access mode assignment ς, then that assignment will type check. The example gives the intuition for why this is true; at each step where a typing rule would reject a program, e.g., having type Rd on a write access, the generated constraint set already cannot solve to that access mode. The proof relies on defining a correspondence between a partially generated constraint set and the concrete typing context (Appendix Section E).

6 CORRECTNESS AS NON-INTERFERENCE

To state our correctness theorem, we first formalize system inputs, outputs, and their equivalences, using these notions to define noninterference and subsequently correctness of intermittent systems. We then state and prove that well-typed Curricle programs run correctly on intermittent systems. Detailed definitions and auxiliary lemmas are in Sections G–K of the supplementary material.

6.1 Definitions for Correctness

To formally state that nondeterministic power failures do not interfere with the program execution (noninterference), we first define notions of input, output, and memory equivalence.

We define two helper functions: *lastInputs*(O) to extract the sensor inputs for the completed execution of each atomic region and all sensor inputs of a JIT segment; and *mstatesOf*(O) to extract all $\mathsf{mstate}(N, V)$ observations out of the trace. We say two input streams are equivalent w.r.t. idempotence, written $I_1 \approx^{\mathsf{in}} I_2$, if all sensor inputs taken under a low (Id) context are the same. This allows traces T_1 and T_2 to differ on sensor readings taken during partial executions of atomic regions (which are thrown out by lastInputs) and during the execution of branches that depend on non-idempotent conditional expressions. We define two output sequences as equivalent w.r.t. idempotence, written $O_1 \approx^{\mathsf{out}} O_2$, if the memory state observations at the completion of each atomic region and JIT segment agree on idempotent locations (those have type qualifier Id).

We define our first correctness definition, one of noninterference, below; it states that, starting from a program's initial state, any two intermittent executions of the program produce equivalent memory states, as long as their idempotent sensor inputs are equivalent. Figure 10 illustrates the definition for an atomic region. The left side shows multiple partial executions of two traces and the right shows the last, completed execution. As the inputs are equivalent, for noninterference to hold, the memory state observation at each transition of this final execution must be equivalent.

Definition 1 (Noninterference). *Nondeterministic power failures do not interfere with the intermittent execution of a program prog from Σp iff given Σp s.t. it is the initial state of prog and two execution traces T_1 and T_2, $T_1 = \mathcal{E}_1, \mathcal{I}_1, \varepsilon_1 \mid \Sigma p \overset{O_1}{\Longrightarrow}{}^* T_2 = \mathcal{E}_2, \mathcal{I}_2, \varepsilon_2 \mid \Sigma p \overset{O_2}{\Longrightarrow}{}^*$ s.t. lastInputs$(O_1) \approx^{\mathsf{in}}$ lastInputs(O_2), it is the case that mstatesof$(O_1) \approx^{\mathsf{out}}$ mstatesof(O_2).*

6.2 Noninterference Proof

We prove the following main correctness theorem, which states that, starting from a well typed state, then for all intermittent executions of this program, all intermittent executions with power-failure equivalent inputs will produce equivalent outputs.

Theorem 2 (Noninterference). *Given any initial configuration Σp for prog, s.t. $\vdash \Sigma p$: ok, nondeterministic power failures do not interfere with the intermittent execution of prog from Σp.*

We prove this theorem by inducting over the JIT and atomic regions in the traces and then over the power failures in each atomic region execution. Much of the complexity comes from proving partial executions of atomic regions eventually yield equivalent results (illustrated in Figure 10). To show this, we define a *low*-equivalence ($\approx$) relation between memory states and configurations that holds at every execution step during the completed execution. The high level idea of the equivalence definition for memory states is that the value of Nid locations can differ, while the value of Id locations should match. However, a partial execution updates persistent memory, and only reverts checkpointed locations, introducing differences in the Id locations of the two traces. Thus, the memory state relation uses the access qualifiers, which indicate whether a location could be written to on prior partial executions, and write flags, which indicate whether the location is written to in this execution, to constrain where these differences can occur. If a location has a read-only type, e.g., Id(Rd), this value can never differ. If a location has a first write type, e.g., Id(Wt), the value can differ until the first write occurs, i.e., the write flag is set, as this write must overwrite the difference. Assuming a well-typed program, such a write must always occur. As non-idempotence can also cause commands executed to differ if a branch decision is made on non-idempotent values, our equivalence definition for configurations states that only low context stack content and commands must match.

To reason about partial executions, we define a relation between an intermittent configuration and the entry point configuration, $\Sigma_1 \prec \Sigma_1^1$, showing that as a trace advances it does not change the value of Id(Rd) or never accessed locations. We then show that after a reboot, the resulting state Σ_{1r} is low equivalent to the entry state; checkpointed variables must revert to their values at the entry point by the definition of a recovery system, and other written locations switch to an unwritten flag and are allowed to differ. In this way, if two traces start from equivalent states, no matter the power failures each trace experiences, any partial execution always "collapses" back to an equivalent state. Then, we show that for the last, complete execution, at each step, the configurations maintain $\approx$ equivalence. Since the programs are well-typed, there can never be a command that attempts to read the differing locations written on a past execution. As all low reads must be equivalent, any low writes will get the same value, overwriting values from past executions. If a trace switches to a high context, any written variables must be high, or the program would not have type checked. After returning from the high context, the traces continue in lock-step. Finally, we show that all potentially different Id(Wt) or Id($\mathsf{Wt}^t \oplus$ Rd)-typed locations must have been written by the end of the atomic region. Maintaining that only Nid typed locations are different by region end allows us to switch between execution modes in a top level induction between program segments.

6.3 Correctness w.r.t. Continuously Powered Execution

To connect our noninterference result to continuous executions, we also prove the following corollary, stating that an intermittent execution with zero power failure events is equivalent to a continuous execution. We write $\sigma = \Sigma^-$ if $\Sigma = (\kappa, mc, N, M_N, S, prog, seg)$ and $\sigma = (N_c, M_N, S, prog^-, seg)$, $N_c = \mathsf{Finalize}(\kappa, N)$, and $prog^-$ is *prog* with atomic region boundaries stripped.

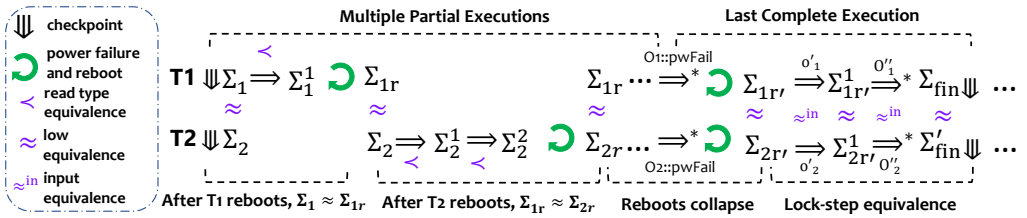

Fig. 10. Overview of the proof structure for atomic regions

Corollary 3 (Intermittent-Erase-Cont). *Given Σp and $\sigma p = \Sigma p^{-}$, O_1 is power failure free, $T_1 = \mathcal{E}, \mathcal{I}, \infty \mid \Sigma p \overset{O_1}{\Longrightarrow} \mathcal{E}, \mathcal{I}', \infty \mid \Sigma p', \sigma p \overset{O_1^{-}}{\Longrightarrow} \sigma p'$, implies $O_1 = O_2$ and $\sigma' p = \Sigma p'^{-}$*

The key differences in the two semantics are that the intermittent execution will start and end atomic regions, and it will use the recovery context when reading and writing memory. Thus, the proof relies on the recovery system properties (SP1, RW1, FP1).

7 RECOVERY SYSTEM AND UNDERLYING HARDWARE

To take advantage of the correctness guarantees provided by Curricle, the recovery system implementation, which is in charge of providing an idempotent view of any location in the given recovery list by the time the location is read, needs only to satisfy the properties in Section 4.3. Similarly, Curricle makes few assumptions about the underlying hardware, hiding architectural details beneath the system abstraction layer. We show an example undo-logging instantiation and briefly discuss how hardware impacts Curricle.

7.1 An Undo Logging Instantiation

Below are the definitions of the recovery system routines of an up-front undo logging system. To set up an atomic mode recovery point, the current stack, map M_n, program, and value of the non-volatile memory for the locations in *ckPts* are saved. In JIT mode, the stack, map, and program are saved. Neither routine changes the value of any variables (SP1).

$$\frac{\kappa' = (N|_{ckPts}, S, M_n, prog, c)}{\mathsf{Setup}(\mathsf{atom}, ckPts, nIds, N, M_n, S, prog, c) = (\kappa', N)} \qquad \frac{\kappa' = (\emptyset, S, M_n, prog, c)}{\mathsf{Setup}(\mathsf{jit}, \emptyset, \emptyset, N, M_n, S, prog, c) = (\kappa', N)}$$

$$\frac{\kappa = (N_c, S_c, M_{nc}, prog_c, c_c)}{\mathsf{Recover}(mc, \kappa, N, S) = \kappa, N \lhd N_c, M_{nc}, S_c, prog_c, c_c} \qquad \frac{}{\mathsf{Finalize}(N, \kappa) = N}$$

On reboot, regardless of mode, the system recovers by updating the stack, map and program to the saved versions, and applies any saved values back into the memory. This ensures that locations in the recovery list and all the execution state return to their values at the setup point (SP2, 3), while other state remains the same (SP4). As *ckPts* and *nIds* must be disjoint, SP5 (*nIds* should not be modified) is also satisfied. The `lookup` and `update` functions (omitted) directly access the nonvolatile memory [2], even for locations in *ckPts*, satisfying RW1 and making finalizing a no-op, as any updates already reside in memory. Thus the old recovery context can be discarded (FP1).

7.2 How Does Hardware Affect Curricle?

Curricle's recovery system abstraction hides memory model and micro-architectural details from the type system, making Curricle applicable even as the underlying hardware changes.

[2]The `update` function for an *on-demand* undo-logging system also saves initial values to the log.

Memory Models & Re-ordering. Most current intermittent systems are sequential, with a simple memory model. Furthermore, caches are write-through, and compilers do not re-order instructions past region boundaries. As such, instructions can be assumed to execute and persist in program order, within the region they appear in the source. However, more complex hardware, such as a write-back cache or volatile scratchpad, can cause instructions to persist out of order, requiring reasoning about the persistent memory model [Pelley et al. 2014, 2015; Raad and Vafeiadis 2018; Raad et al. 2019]. We designed Curricle so that persistence reasoning applies to the system abstraction layer (e.g., proving that Finalize persists values in a cache) and not the type system, which is agnostic to these lower level details. Proving that this abstraction layer is sufficient is future work.

Volatility Assumptions. The semantics of the core calculus treat the stack S as volatile and all other memory N as non-volatile. Some intermittent system designs place most or all of the stack in the non-volatile memory [Kortbeek et al. 2020b; Maeng and Lucia 2018], and some emerging processors are completely non-volatile, including the register file [Ma et al. 2018]. Curricle is still useful on such systems and devices. The execution model with pure non-volatile memory is similar to the JIT execution model described in Section 2.1, as the system directly resumes execution from the point of power failure, which remains arbitrary. Just as with JIT execution, atomic regions remain a necessary language construct to satisfy timing constraints, which may require input operations to be re-executed freshly after a power failure. Any re-execution introduces the possibility of non-idempotent updates, the key concern of Curricle.

8 CURRICLE IMPLEMENTATION

We implemented Curricle using Rust's procedural macro feature, which operates on a program's abstract syntax tree (AST). The programmer annotates necessary types, leaving the rest to be inferred. Curricle's analysis visits AST nodes, applying syntax-node-specific type inference and checking rules to each operation. After Curricle's checking pass, Rust's type checker runs, validating the type linearity upon which Curricle relies, despite Curricle not explicitly modelling ownership.

Using the Curricle Implementation. To use Curricle, a programmer annotates their Rust code with attributes. For example, the Rust annotation `#[nids(high)]` marks `high` as non-idempotent in Figure 3. To reduce annotation burden, Curricle requires specifying which variables are Nid only, assuming the rest are Id. The programmer additionally marks which functions execute atomically and annotates the signature of any function called from such an atomic function with taint types. In our prototype, the programmer provides these taint types to match our formal model. An alternative would be to use existing intermittent taint-tracking analyses [Surbatovich et al. 2020]; nothing fundamental prevents building these analyses into our prototype, but we opted not to because doing so would require porting from LLVM to Rust procedural macros. While our syntax for functions/closures explicitly lists captured variables, the programmer need *not* provide the list. Similarly, functions need not be declared with a `let` binding.

Curricle infers the access modes of all Id variables and checks that there are no information flow violations. The analysis reports both information flow type errors and variables that have access mode Ck, which must be added to the recovery list. The goal of this report is to help the programmer understand their types and identify variables that should be Nid, as well as to guide refactoring to avoid non-idempotent access patterns and reduce the recovery list.

Connecting Curricle to the Underlying Recovery System. Curricle exposes the recovery list that it produces through an interface to recovery system implementations. For each atomic region, Curricle reports variables in the recovery list. For a system that, e.g., saves variables at region start [Lucia and Ransford 2015; Surbatovich et al. 2021], this list suffices, because the variable name corresponds to a non-volatile memory location. For systems that log data *on demand* at a write

or read [Kortbeek et al. 2020b; Maeng et al. 2017], Curricle must refer to a particular non-volatile memory location, potentially through an alias. Curricle passes this points-to information to the recovery system by instrumenting each read and write of a recovery list variable with calls to `tagRead` and `tagWrite` interface functions, respectively. A recovery system designer can implement these functions to log a location using an on-demand logging strategy.

Limitations & Differences from the Formal Model. Curricle's implementation closely matches its formal model, but there are a few differences. The prototype requires atomic regions to be functions. In some code, Rust allows type elision and not all type information is available to our procedural macros. Writing atomic regions as functions ensures sufficient information for our type analysis without demanding extra work for the programmer, because function arguments require explicit types. Our implementation checks real Rust code, which includes more syntactic constructs than our core calculus. Often these constructs map to a command in our calculus. For example, a `match` expression can be treated as an `if` with additional branches. Similarly, an immutable `let` binding can be treated as our generic `let`; every program must pass the Rust type checker after passing Curricle's checks, confirming that an immutable `let` was never written to, even though Curricle does not explicitly model mutability. One excluded feature from the model that must be handled specially by the Curricle implementation is arrays. Precise static array reference analysis is difficult and often impossible, so our type inference treats arrays conservatively. After a leading write to an array, a read of some array element could still be a first access, and a subsequent write should require the access mode of the array to be Ck. To capture this conservatism, the inference constraint must record that after, e.g., a write, any element still has the potential to be not yet accessed, leveraging the "lazy" constraint discussed in Section 5.3.

Curricle does not support unsafe Rust, which allows syntactically untrackable pointer manipulations, or interior mutability (e.g., `RefCell`). Determining what properties unsafe Rust blocks must satisfy for sound Curricle analysis, as well as how to specify and enforce them, is a rich future research direction that could integrate with Rust formal models [Jung et al. 2019, 2017].

9 EVALUATION

Our evaluation shows that Curricle benefits both intermittent system designers and application programmers; Curricle enables the recovery system to reduce dynamic logging time and memory overheads compared to approaches that recover all written data, without the designer writing complex, ad hoc analyses, compared to approaches that use dataflow analysis passes. Application programmers can express idempotence requirements with a type-level assurance of correctness.

Table 1. Benchmark characteristics and effort to use Curricle

		Benchmark Characteristics			Using Curricle			
Origin	**App**	**LoC**	**# Atomic Reg.**	**# Fn/Clsrs called in Atomics**	**# Anno.**	**# Explicit typed Var.**	**LoC Overhead**	**# Ck**
Ocelot	**Activity**	518	2	8	10	29	1.9%	2
	CEM	320	1	1	2	3	0.6%	1
	Greenhouse	234	1	4	5	10	2.1%	1
	Tire	424	3	10	13	33	3.1%	3
Curricle	**Vax**	374	2	10	(1 Nid anno.)13	41	3.5%	7
	Vax-2	393	2	10	(1 Nid anno.)13	37	3.3%	5

9.1 Benchmarks and Comparison System Designs

We evaluated Curricle using benchmarks from prior work [Surbatovich et al. 2021], tweaked to compile with Curricle by converting atomic regions to functions and adding taint information to function signatures. Table 1 characterizes our benchmarks and the code changes Curricle requires.

At left is provenance, number of lines of code (LoC), atomic regions, and functions called in atomic regions. `Activity` classifies human activity, `CEM` is a sensor logging and compression app, and `Greenhouse` and `Tire` are sensor-driven alarm apps. We add two new benchmarks, `Vax` and `Vax-2`, which record and compress multi-sensor data about a cold storage environment (e.g., vaccine transport); `Vax-2` is `Vax` refactored to avoid a WAR on array data. The first three columns on the right show LoC for annotations, how many variables require type annotations, and percent increase in LoC for annotations. Each application needs one annotation per atomic function, one to declare Nid types, and one per function/closure called during atomic region execution, to provide taint types. LoC overhead scales mainly with the number of functions called during atomic regions, not total benchmark size, ranging from 0.6% to 3.5%. The last column shows the variable count of Curricle's inferred recovery list.

To explore the benefits of Curricle, we compare to systems that allow the programmer to express atomicity requirements. We exclude systems that automatically place recovery points—e.g., by periodic or energy-level triggered interrupts [Balsamo et al. 2016, 2015] or software analysis [Maeng and Lucia 2018; Van Der Woude and Hicks 2016]—because these systems do not provide timeliness and consistency [Hester et al. 2017; Kortbeek et al. 2020b; Surbatovich et al. 2021]. There are two main design axes for atomic regions: *which variables* to recover and *how*. Table 2 organizes prior systems on these axes. Systems either recover all written state or recover only potentially non-idempotent state. Most systems use redo- or undo-logging mechanisms to recover state, but some use channels [Colin and Lucia 2016; Hester et al. 2017; Yildirim et al. 2018] that double buffer data to eliminate WARs. Logging all written data [Kortbeek et al. 2020a,b; Maeng and Lucia 2020; Ruppel and Lucia 2019] is popular as the design choice does not need dataflow analyses or assumptions about pointers [Branco et al. 2019]. Systems that identify variables manipulated non-idempotently do so using dataflow analyses [Lucia and Ransford 2015; Maeng et al. 2017; Surbatovich et al. 2021, 2020] or manual annotation [Maeng and Lucia 2019]. While Chain [Colin and Lucia 2016] allows the programmer to manually identify which channels to double buffer, no channel-based system uses dataflow analysis for automatic identification. We evaluate whether Curricle benefits the systems in Table 2, which we refer to categorically as **Channels**, **All-Writes**, **CFG-pass**, and **Id-Manual**.

Table 2. Recovery System Designs

	Channels	Logging
All-Writes	Mayfly	TICS Coati
	Ink	Catnap BFree
Non-Id Set	$\emptyset$	Alpaca Ocelot
CFG Pass		Dino
Manual	Chain	Samoyed

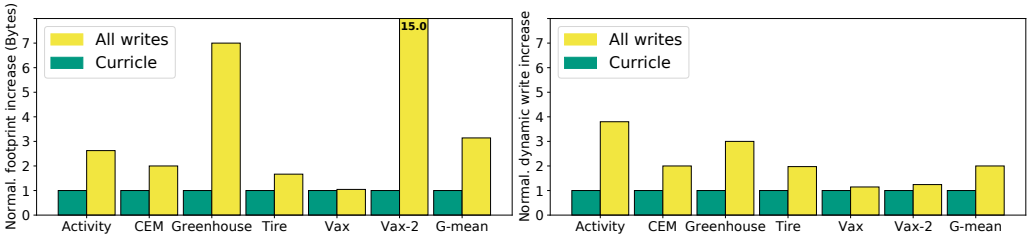

Fig. 11. Normalized increase in allocation footprint and dynamic writes from logging all writes

9.2 Using Curricle's Abstraction Lessens Overheads While Providing Correctness

Curricle alleviates the conservatism inherent in **All-Writes** systems without requiring the intermittent system designer to provide complex compiler analyses or to make the unchecked assumptions of **CFG-pass** systems. We quantify the cost of the conservatism, comparing the amount of storage required to store all written variables versus only Curricle-identified ones, as well as comparing

dynamic write counts. Figure 11 shows these results. The left plot shows the increase in memory footprint, and the right plot shows the increase in dynamic writes, normalized to the Curricle-identified recovery list with a geometric mean increase at right. In all cases, the program writes some data that need not be recovered for correctness. The data show that recovering all written variables imposes on average a 2x overhead in write logging operations and a 3.14x overhead in logging memory footprint. Curricle eliminates these overheads by reducing the total amount of data to log, a benefit that is complementary to optimizing the logging mechanism itself.

Curricle's reduction in logging overheads provides the quantitative benefits shown in Figure 11, but also provides a qualitative benefit, sparing the system designer from implementing system-specific WAR and RIO analyses (which prior work noted is undesirable [Branco et al. 2019]). These analyses are brittle and complex for prior systems targeting C (see Section 2.2). Curricle instead leverages Rust's memory safety, allowing precise idempotence analysis as the Rust compiler enforces necessary pointer restrictions. An intermittent system designer building a recovery system targeting Curricle benefits from the reduced log overhead, has lower compiler complexity by depending on Curricle's analysis only, and need not impose unchecked pointer restrictions. Thus, as summarized in Table 3, Curricle provides a *performance* benefit to **All-Writes** system designs, a qualitative benefit of simplicity to **CFG-pass** designs, and *correctness* benefits to all the design configurations we consider.

9.3 Curricle Enables Programmers to Express Idempotence Requirements Explicitly

Curricle gives programmers abstractions to specify complex idempotence requirements that are inexpressible in prior systems and checks that the requirements are upheld. For example, `Vax` and `Vax-2` record a maximum temperature sensor value. An **All-write** or **CFG-pass** system will, by default, restore the max flag on re-execution, as its update *is* non-idempotent, allowing the application not to record the actual highest sensed temperature. Samoyed [Maeng and Lucia 2019] requires the programmer to specify what parameters into an atomic region have WARs, only logging those variables (Samoyed ignores RIOs). A clever programmer could (ab-)use this unchecked annotation approach to allow non-idempotence by purposely leaving WAR variables unlabeled. Samoyed cannot, however, check whether an omission is purposeful or a bug. For systems that log all writes or perform CFG analysis, a programmer *may* be able to avoid checkpointing some data with low-level hacks. For example, a redo log instrumentation pass might identify variables to log by examining the linker section in which they reside. A programmer can hack the linker script to create a "non-idempotent" section and force a variable to be stored there, causing the analysis pass to ignore it. This type of low-level approach requires an *application* programmer to delve into system code, reason about information-flow manually, and rely on behaviour that is not part of the recovery system's specification. Curricle provides the programmer an abstraction with which to specify variables as non-idempotent via its type system and *checks* that the program accesses cannot violate the specification. Curricle furthermore defines the properties the system implementation must provide to uphold the specification at runtime, providing a interface to the recovery system.

Table 3. Benefits of using Curricle

(a) Summary of benefits to system designers

Benefit	All-writes	Id-Manual	CFG-pass	Curricle
Optimized set	×	✓	✓	✓
Simpler Back-end	✓	✓	×	✓
No Restrictions	✓	×	×	✓
Checked Correct	N/A	×	×	✓

(b) Summary of benefits to application programmers

Benefit	All-writes	Id-Manual	CFG-pass	Curricle
Express Spec	×	×	×	✓
Checked Correct	N/A	×	×	✓

10 RELATED WORK

Curricle is primarily related to work in intermittent computing, information flow and idempotence analysis, and work leveraging the ownership properties of Rust.

Intermittent Computing. How the memory consistency reasoning of past works [Colin and Lucia 2016; Hester et al. 2017; Maeng et al. 2017; Maeng and Lucia 2019, 2020; Ruppel and Lucia 2019; Surbatovich et al. 2021, 2020; Yildirim et al. 2018] relates to Curricle is discussed in the motivation and evaluation. Curricle is the first intermittent computing work to provide a type system for reasoning about idempotence. Ocelot [Surbatovich et al. 2021] is the only other intermittent computing work targeting Rust, to the best of our knowledge. However, its correctness analysis is still in LLVM and does not allow the programmer to express idempotence requirements. Coati [Ruppel and Lucia 2019] and Ink [Yildirim et al. 2018] address interrupt driven concurrency for intermittent systems and Immortal Threads [Yıldız et al. 2022] provides multi-threading. Expanding the type-system for concurrency is future work. Karma [Branco et al. 2019], Restop [Rodriguez Arreola et al. 2018], and Sytare [Berthou et al. 2017] address retaining consistent state of peripheral input devices. Work [Berthou et al. 2020] has been done in formalizing and proving this peripheral correctness. This peripheral reasoning is complimentary to Curricle and could be integrated into the system abstraction. [Maioli and Mottola 2020] argue that allowing programs to be intermittence aware (i.e., use non-idempotent data) exposes new design pattern and performance optimizations, though they do not define correctness in those scenarios. Curricle supports and formalizes this point of view by providing the non-idempotent type to programmers, though future work should explore endorsement of non-idempotent types. [Surbatovich et al. 2020] provide a formal framework for reasoning about intermittent computing, defining a correct intermittent execution as one that refines a continuous execution. Unlike Curricle, they do not allow any non-idempotence or provide constructs for the programmer to express requirements. Curricle builds upon their language model but supports more features, such as references and functions. Moreover, Curricle's correctness theorem is more general, showing non-interference.

Information Flow and Idempotence Analysis. Information flow type systems are a well-studied technique in reasoning about program security [Heintze and Riecke 1998; Rajani et al. 2017; Sabelfeld and Myers 2006; Zdancewic and Myers 2001]. Curricle's type system draws inspiration from integrity reasoning [Biba 1977], as it ensures that untrusted (non-idempotent) data does not not flow to trusted (idempotent) data. Curricle's checking of idempotence qualifiers for information flow violations and taint propagation via taint qualifiers follow standard techniques. However, Curricle does not check for security violations, but rather the presence of idempotence violations on intermittent systems, requiring additional type qualification for access modes. Future work extending Curricle to reason about distributed intermittent systems can integrate with work using information flow types to reason about interactions between consistency models on geo-distributed systems [Milano and Myers 2018].

Idempotence analysis has frequently been used to make systems more robust to failures, including in idempotent processing [De Kruijf and Sankaralingam 2013; de Kruijf et al. 2012], logging mechanisms for persistent memory systems [Liu et al. 2018], and for fault tolerance in distributed systems [Ramalingam and Vaswani 2013]. These approaches do not use types or allow the programer to specify custom idempotence requirements. Our support for non-idempotence draws inspiration in spirit from Safe Nondeterminism [Bocchino et al. 2011], which provides language support for controlled non-determinism in concurrent programs.

Ownership Types and Rust. Curricle's type checking analysis relies on Rust's linear types to correctly identify accessed locations, as they prevent arbitrary aliasing. Curricle does not directly model lifetimes and ownership, instead relying on the Rust compiler in the implementation. The

Rustbelt [Dang et al. 2019; Jung et al. 2017] and Oxide [Weiss et al. 2019] projects provide formal, Rust-like semantics, and Curricle could integrate more directly with these models in future work.

Many recent works take advantage of Rust's strict typing to do static analysis that would be prohibitively difficult with unrestricted pointer behaviour, including for functional verification with constrained horn clauses [Matsushita et al. 2021], functional verification using pre- and post-conditions of functions [Astrauskas et al. 2019], and in safely implementing compiler optimizations around unsafe Rust code [Jung et al. 2019]. Closer to our work, [Balasubramanian et al. 2017] and [Njor and Gústafsson 2021] demonstrate that Rust's type system indeed makes more precise static taint tracking possible, providing prototype information flow control and analysis tools. Most recently, Flowistry [Crichton et al. 2022] shows that information flow in Rust can be analyzed modularly, as the types are powerful enough that only function signatures are needed. The authors formally prove their non-interference theorem using Oxide's model. Unlike these works, Curricle does not aim to provide information flow analysis for general Rust programs, but instead uses information flow along with type-state to check if programs are safe for intermittent execution.

11 CONCLUSION & FUTURE WORK

This work presents Curricle, a type system for safe intermittence, and a requirements specification for intermittent recovery systems. Curricle uses information flow reasoning and type qualifiers to identify recovery lists and to disallow programs that cause unintentional non-idempotence. Curricle obviates the need for intermittent system designers to write complex back-end analyses while enabling them to benefit from the reduced recovery sets such an analysis would provide. A programmer using Curricle can express more complex idempotence requirements than past work allows, while simultaneously having assurance that their program will execute correctly.

Formally reasoning about future intermittent systems provides interesting directions for Curricle:

Concurrency. Some intermittent systems provide interrupt-driven concurrency [Branco et al. 2019; Ruppel and Lucia 2019; Yildirim et al. 2018], and future systems should support multi-tenancy. Type checking that programs are safe to run intermittently *and* concurrently requires additions to the model and type system to reason about shared memory dependencies and the non-determinism of power failures and preemption together.

Outputs and Endorsement. To support distributed intermittent systems, desirable for smart agriculture or smart city applications, Curricle should be extended with output operations, to reason about communication protocols. Additionally, reasoning about interactions between non-idempotent data and idempotent data in the context of concurrent systems and communication—e.g., a reboot counter used by a scheduler or a replayed output—requires extending Curricle with endorsement capabilities (i.e., lowering Nid types to Id types).

Studying User Interactions. This work argues that providing abstractions for non-idempotence via types increases programmer expressivity and simplifies correctness reasoning, relying on the general power of type systems. Further quantifying the usability of Curricle and future language abstractions for concurrent and distributed intermittent systems requires dedicated user studies.

ACKNOWLEDGMENTS

We thank the reviewers for their feedback and Chao Wang for shepherding this work. We also thank the Abstract Research Lab at CMU, the Foundations of Programming group at MPI-SWS, and the PLDI SRC 2022 judges for insightful feedback on early versions of Curricle. 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

Joshua Adkins, Bradford Campbell, Branden Ghena, Neal Jackson, Pat Pannuto, and Prabal Dutta. 2016. The Signpost Network: Demo Abstract. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM* (Stanford, CA, USA) *(SenSys '16)*. https://doi.org/10.1145/2994551.2996542

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 147 (oct 2019), 30 pages. https://doi.org/10.1145/3360573

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

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

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

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* (London, United Kingdom) *(LCTES '20)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/3372799.3394365

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)*. IEEE. https://doi.org/10.1109/giots.2017.8016243

K. J. Biba. 1977. *Integrity Considerations for Secure Computer Systems* (esd-tr-76-372 ed.). Technical Report. MITRE Corp. 66 pages.

Robert L. Bocchino, Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. 2011. Safe Nondeterminism in a Deterministic-by-Default Parallel Language. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 535–548. https://doi.org/10.1145/1926385.1926447

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* (New York, New York) *(SenSys '19)*. Association for Computing Machinery, New York, NY, USA, 55–67. https://doi.org/10.1145/3356250.3360033

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

Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. 2022. Modular information flow through ownership. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM. https://doi.org/10.1145/3519939.3523445

Alexander Curtiss, Blaine Rothrock, Abu Bakar, Nivedita Arora, Jason Huang, Zachary Englhardt, Aaron-Patrick Empedrado, Chixiang Wang, Saad Ahmed, Yang Zhang, Nabil Alshurafa, and Josiah Hester. 2022. FaceBit: Smart Face Masks Platform. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 5, 4, Article 151 (dec 2022), 44 pages. https://doi.org/10.1145/3494991

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt Meets Relaxed Memory. *Proc. ACM Program. Lang.* 4, POPL, Article 34 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371102

Marc De Kruijf and Karthikeyan Sankaralingam. 2013. Idempotent code generation: Implementation, analysis, and evaluation. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/cgo.2013.6495002

Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static Analysis and Compiler Design for Idempotent Processing. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. https://doi.org/10.1145/2254064.2254120

Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (may 1976), 236–243. https://doi.org/10.1145/360051.360056

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* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 670–684. https://doi.org/10.1145/3352460.3358277

Xiaochen Guo, Engin Ipek, and Tolga Soyata. 2010. Resistive Computation: Avoiding the Power Wall with Low-leakage, STT-MRAM Based Computing. *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 371–382. https://doi.org/10.1145/

1816038.1816012

Nevin Heintze and Jon G. Riecke. 1998. The SLam Calculus: Programming with Secrecy and Integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 365–377. https://doi.org/10.1145/268946.268976

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*. https://doi.org/10.1145/3131672.3131673

iota Biosciences. 2022. iota Biosciences. https://iota.bio/.

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (dec 2019), 32 pages. https://doi.org/10.1145/3371109

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

Vito Kortbeek, Abu Bakar, Stefany Cruz, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. 2020a. BFree: Enabling Battery-Free Sensor Prototyping with Python. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 4, Article 135 (Dec. 2020), 39 pages. https://doi.org/10.1145/3432191

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

Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. 2018. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 258–270. https://doi.org/10.1109/MICRO.2018.00029

Brandon Lucia, Brad Denby, Zachary Manchester, Harsh Desai, Emily Ruppel, and Alexei Colin. 2021. Computational Nanosatellite Constellations: Opportunities and Challenges. *GetMobile: Mobile Comp. and Comm.* 25, 1 (June 2021), 16–23. https://doi.org/10.1145/3471440.3471446

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* (Portland, OR, USA) *(PLDI 2015)*. https://doi.org/10.1145/2737924.2737978

Kaisheng Ma, Xueqing Li, Mahmut Taylan Kandemir, Jack Sampson, Vijaykrishnan Narayanan, Jinyang Li, Tongda Wu, Zhibo Wang, Yongpan Liu, and Yuan Xie. 2018. NEOFog: Nonvolatility-Exploiting Optimizations for Fog Computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/3173162.3177154

Kaisheng Ma, Xueqing Li, Jinyang Li, Yongpan Liu, Yuan Xie, Jack Sampson, Mahmut Taylan Kandemir, and Vijaykrishnan Narayanan. 2017. Incidental Computing on IoT Nonvolatile Processors. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. ACM, New York, NY, USA, 204–218. https://doi.org/10.1145/3123939.3124533

Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 96 (Oct. 2017), 96:1–96:30 pages. https://doi.org/10.1145/3133920

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* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, Berkeley, CA, USA, 129–144. http://dl.acm.org/citation.cfm?id=3291168.3291178

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

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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1005–1021. https://doi.org/10.1145/3385412.3385998

Andrea Maioli and Luca Mottola. 2020. Intermittence Anomalies Not Considered Harmful. In *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems* (Virtual Event, Japan) *(ENSsys '20)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3417308.3430266

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-Based Verification for Rust Programs. *ACM Trans. Program. Lang. Syst.* 43, 4, Article 15 (oct 2021), 54 pages. https://doi.org/10.1145/3462205

Mae Milano and Andrew C. Myers. 2018. MixT: A Language for Mixing Consistency in Geodistributed Transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 226–241. https://doi.org/10.1145/3192366.3192375

NASA. 2019. What is KickSat-2? https://www.nasa.gov/ames/kicksat. Visited April 15th, 2022.

Emil Njor and Hilmar Gústafsson. 2021. Static Taint Analysis in Rust. *Master's thesis* (2021).

Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. Piscataway, NJ, USA. https://doi.org/10.1109/isca.2014.6853222

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

Azalea Raad and Viktor Vafeiadis. 2018. Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 137 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276507

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

Vineet Rajani, Iulia Bastys, Willard Rafnsson, and Deepak Garg. 2017. Type Systems for Information Flow Control: The Question of Granularity. *ACM SIGLOG News* 4, 1 (feb 2017), 6–21. https://doi.org/10.1145/3051528.3051531

G. Ramalingam and Kapil Vaswani. 2013. Fault Tolerance via Idempotence. *Principles of Programming Languages (POPL)* (January 2013). https://www.microsoft.com/en-us/research/publication/fault-tolerance-via-idempotence/

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. https://doi.org/10.3390/s18010172

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

A. Sabelfeld and A. C. Myers. 2006. Language-Based Information-Flow Security. *IEEE J.Sel. A. Commun.* 21, 1 (sep 2006), 5–19. https://doi.org/10.1109/JSAC.2002.806121

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* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 851–866. https://doi.org/10.1145/3453483.3454081

Milijana Surbatovich, Brandon Lucia, and Limin Jia. 2020. Towards a Formal Foundation of Intermittent Computing. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 163 (Nov. 2020), 31 pages. https://doi.org/10.1145/3428231

Milijana Surbatovich, Naomi Spargo, Limin Jia, and Brandon Lucia. 2023. Technical Report: A Type System for Safe Intermittent Computing. https://doi.org/10.1184/R1/22583920

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

Joel Van Der Woude and Matthew Hicks. 2016. Intermittent computation without hardware support or programmer intervention. In *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/vanderwoude

Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2019. Oxide: The Essence of Rust. https://doi.org/10.48550/ARXIV.1903.00982

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* (Shenzhen, China) *(SenSys '18)*. ACM, New York, NY, USA, 41–53. https://doi.org/10.1145/3274783.3274837

Eren Yıldız, Lijun Chen, and Kasim Sinan Yıldırım. 2022. Immortal Threads: Multithreaded Event-driven Intermittent Computing on Ultra-Low-Power Microcontrollers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 339–355. https://www.usenix.org/conference/osdi22/presentation/yildiz

Steve Zdancewic and Andrew C. Myers. 2001. Secure Information Flow and CPS. In *Proceedings of the 10th European Symposium on Programming Languages and Systems (ESOP '01)*. Springer-Verlag, Berlin, Heidelberg, 46–61. https://doi.org/10.1007/3-540-45309-1_4

Received 2022-11-10; accepted 2023-03-31