Towards a Formal Foundation of Intermittent Computing

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

Intermittently powered devices enable new applications in harsh or inaccessible environments, such as space or in-body implants, but also introduce problems in programmability and correctness. Researchers have developed programming models to ensure that programs make progress and do not produce erroneous results due to memory inconsistencies caused by intermittent executions. As the technology has matured, more and more features are added to intermittently powered devices, such as I/O. Prior work has shown that all existing intermittent execution models have problems with repeated device or sensor inputs (RIO). RIOs could leave intermittent executions in an inconsistent state. Such problems and the proliferation of existing intermittent execution models necessitate a formal foundation for intermittent computing.

In this paper, we formalize intermittent execution models, their correctness properties with respect to memory consistency and inputs, and identify the invariants needed to prove systems correct. We prove equivalence between several existing intermittent systems. To address RIO problems, we define an algorithm for identifying variables affected by RIOs that need to be restored after reboot and prove the algorithm correct. Finally, we implement the algorithm in a novel intermittent runtime system that is correct with respect to input operations and evaluate its performance.

CCS Concepts: • **Theory of computation** → *Operational semantics*; • **Computer systems organization** → *Embedded systems.*

Additional Key Words and Phrases: intermittent computing, operational semantics

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

1 INTRODUCTION

Battery-less, energy-harvesting devices (EHDs) are an emerging class of embedded computing device that operate entirely using energy extracted from their environment, such as light energy from a solar panel or energy from radio waves using an antenna. Free from a battery, these devices enable new applications in IoT [Colin et al. 2018; Fraternali et al. 2018; Hester and Sorber 2017; Jackson et al. 2019], civil infrastructure sensing [Nardello et al. 2019], in-body medical sensing [Proteus Digital Health 2015], and space exploration [Colin et al. 2018; Denby and Lucia 2020; Zac Manchester 2015]. We study EHDs that compute *intermittently* as energy is available. The device slowly harvests energy into a capacitor. After storing sufficient energy to make meaningful progress, the device operates, quickly consuming the energy. After exhausting the stored energy, the device powers off, awaiting more energy. Software executes according to an *intermittent execution*

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

2475-1421/2020/11-ART163
https://doi.org/10.1145/3428231

model, where programs make progress during an active period that is preceded by and followed by an inactive recharge period and a reboot [Balsamo et al. 2015; Colin and Lucia 2016; Jayakumar et al. 2014; Lucia and Ransford 2015; Maeng et al. 2017; Ransford et al. 2011; Woude and Hicks 2016]. A reboot clears volatile state (registers and SRAM) and preserves non-volatile state (FRAM [TI Inc. 2020a] and Flash). This execution pattern is illustrated in Figure 1 (a).

Unpredictably-timed power failures create several challenges for an intermittent execution model, including how to maintain forward progress, ensure memory consistency, manage I/O and concurrency, and correctly interface with hardware. Figure 1 (a) illustrates how executing a program intermittently can result in a memory state inconsistent with any continuous execution of a program, if the program contains certain memory access patterns or repeats input operations. As energy-harvesting devices have matured, an increasing variety of new programming models and runtime systems for intermittent execution, with varying technical approaches to addressing these key challenges, have emerged [Balsamo et al. 2016, 2015; Colin and Lucia 2016; Colin et al. 2018; Ganesan et al. 2019; Gobieski et al. 2019; Jayakumar et al. 2014; Kortbeek et al. 2020; Lucia and Ransford 2015; Ma et al. 2017; Maeng et al. 2017; Maeng and Lucia 2018, 2019; Ransford et al. 2011; Ruppel and Lucia 2019; Woude and Hicks 2016].

The proliferation of diverse intermittent execution models presents a developer with a confusing space of implementation options and raises the question of how to specify and compare the behavioral properties of different systems. Models differ subtly, and a program written for one model may make assumptions not met by another. Moreover, no existing software or hardware system has clear, formally defined behaviour. Such a characterization and formalism is a key building block for defining and proving correctness properties, developing tools to find and fix intermittence-specific bugs, and understanding the fundamental similarities and differences between models. A primary motivation for using intermittent computing systems is their deployability in remote, inaccessible environments. Updating the device may be difficult or even impossible, so a buggy runtime that corrupts the memory state can make the device practically useless. The lack of specifications of intermittent systems is a key impediment to their deployment, particularly in applications that demand high reliability or security.

In this work, we lay the groundwork for provably correct intermittent computing by formalizing the semantics of several classes of intermittent execution models, focusing on the foundational correctness issue of *ensuring memory consistency* in the presence of non-deterministic input operations. Figure 1 (b) provides an overview of our contributions. An intuitive correctness property is that an intermittent execution's behavior should be equivalent to some continuously-powered execution. Prior work [Surbatovich et al. 2019] has shown (confirmed by our formalism) that a majority of existing intermittent systems *do not satisfy a reasonable correctness condition* in the presence of input operations that may change as a program runs intermittently. We articulate the changes to the execution model that are necessary to correctly handle the behavior of input operations in an intermittent execution. We start by formalizing a *checkpoint-based intermittent execution model* based on DINO [Lucia and Ransford 2015]. A checkpoint model saves important state during execution and, after a power failure, restarts from that point in the execution by restoring the checkpoint on reboot. We then formalize the behavior of variants of checkpoint-based systems [Woude and Hicks 2016], including both redo- and undo-logging checkpoint strategies, and a task-based execution model [Maeng et al. 2017]. We relate these systems via bi-simulation, showing that the correctness properties of one system hold for the others. Finally, we propose a new checkpoint-based intermittent execution model that provably handles input behavior correctly, while allowing re-execution of inputs (crucial for data freshness). As far as we know, we are the first to propose such an execution model.

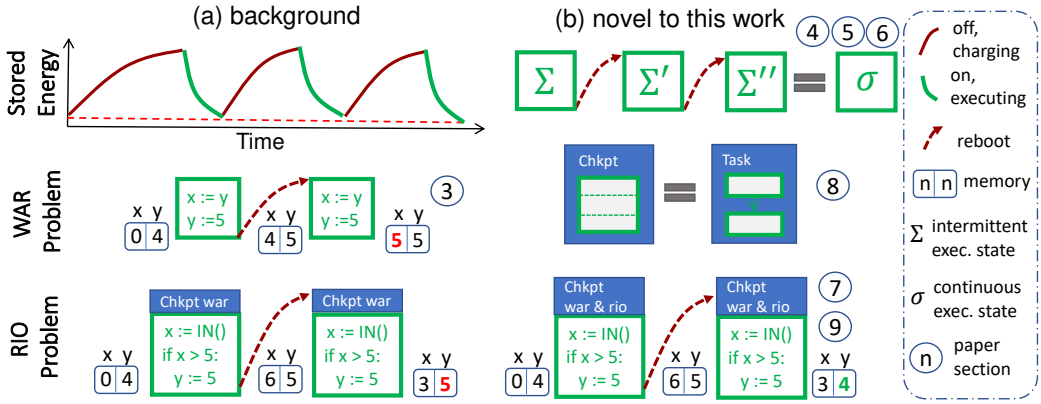

Fig. 1. (a) The intermittent execution model. Re-executions can result in an inconsistent memory state due to write-after-read dependencies and repeated input operations. (b) We provide a formal correctness theorem, prove equivalence between systems, and implement a runtime with a provably correct algorithm

We acknowledge that intermittent systems are similar to systems that handle crashes and failures in file systems or databases [De Kruijf and Sankaralingam 2013; Izraelevitz et al. 2016a; Koskinen and Yang 2016], but there are key differences. As intermittent execution targets highly resource-constrained, embedded systems, and failure is inevitable and frequent, the checkpoint and recovery mechanisms are typically implemented to save and restore the least state possible. Additionally, the correctness of the system depends on the state of both the volatile and non-volatile memory, and programs are often driven by non-deterministic sensor inputs. Existing formalisms for verifying crash consistency typically do not model the internal state of the recovery mechanism [Koskinen and Yang 2016], which is necessary to capture the behaviour of the systems we model, model nonvolatile state only [Chen et al. 2015; Sigurbjarnarson et al. 2016], or do not explicitly model peripherals. These differences mean that existing automatic verifiers or logic frameworks [Chen et al. 2015; Koskinen and Yang 2016; Ntzik et al. 2015; Sigurbjarnarson et al. 2016] are not directly suitable for analyzing intermittent execution models, though the correctness specifications and invariants that we identify are useful for extending such tools to work for intermittent computing. Identifying such invariants for the correctness proofs is nontrivial. These invariants deepen our understanding of intermittent systems. This work is the foundation for formalizing more complex intermittent behaviour, including event-driven and reactive execution [Ruppel and Lucia 2019; Yildirim et al. 2018], or guaranteeing forward progress. We make the following contributions:

- A novel, formal intermittent execution model with inputs, a correctness theorem, and sufficient conditions for correctness.
- A provably sound algorithm to collect variables to checkpoint and an implementation for an existing intermittent system.
- Formalized variants of intermittent-execution models and proofs of the equivalence of these systems via bi-simulation.
- An experimental evaluation of our algorithm implementation on real hardware showing that our technique has low time and space overheads while requiring little to no programmer effort.

Due to space, we relegate detailed formalism and proofs to the companion technical report (TR) located at https://arxiv.org/abs/2007.15126.

2 SCOPE AND RELATED WORK

Fully specifying intermittent system behaviour requires reasoning about diverse properties. These properties include memory consistency, forward progress, timeliness, and correct handling of

concurrency. Our paper addresses memory consistency. An intermittent program that is guaranteed to finish and always processes only fresh data is incorrect if it operates on a memory state inconsistent with any continuous execution of the program. While we focus on memory consistency, we can extend the formalism to cover other correctness properties in the future. In the remainder of this section, we describe these other properties at the high level and sketch what extensions are necessary for our framework to capture these properties. We then discuss how our framework relates to existing research, particularly in intermittent computing and verified crash consistency.

2.1 Scope: What the Paper Is *Not* about

We focus on memory consistency with re-executed inputs. Reasoning about other desirable properties presupposes that the underlying system memory is correct. As intermittent applications are often sensor-driven, ensuring correctness with input operations is paramount. The full set of properties mentioned above is a long term goal that can be reached by building on top of our current framework. Each additional piece requires nontrivial theoretical and implementation components.

Forward progress To make forward progress, a program executing intermittently must be able to execute the region between any two adjacent checkpoints (or any task) with the amount of energy in the device's buffer. Otherwise the program will get stuck, partially executing the region, recharging energy, and rebooting forever. Current intermittent systems assume that the largest task or checkpoint region will be cheap enough to finish [Maeng et al. 2017; Woude and Hicks 2016]. Our correctness theorem is sound relative to this assumption and does not itself prove forward progress. Formalizing and *guaranteeing* forward progress requires a persistent energy model, which is likely to be complex because the amount of energy a sequence of instructions consumes depends on the state of the entire board, not the processor alone. A region between checkpoints could finish on a processor in isolation, but may not if, e.g., the radio is enabled. CleanCut [Colin and Lucia 2018] is a compiler tool that provides a probabilistic energy model to guide programmers in sizing tasks, but it offers no guarantees and does not consider the full state of the board. Samoyed [Maeng and Lucia 2019] allows programmers to specify cheaper alternatives to algorithms that the system can switch to at runtime, if it seems a program is not making progress.

Timeliness As power can be off for an arbitrary period of time, sensor data collected before a power failure can be stale and useless after a reboot. To avoid processing stale data, prior systems have either required external persistent timekeepers [de Winkel et al. 2020; Hester et al. 2016] so that a programmer can specify explicit timing annotations that the system can check at runtime [Hester et al. 2017; Kortbeek et al. 2020], or required that the programmer place sensor calls and uses requiring fresh data in the same checkpoint region or task. We assume the latter approach in this work, and find that while it allows timely processing of data, it also introduces memory inconsistencies, making current systems that take this approach incorrect. Our formalism aids us in developing a runtime that allows consistent re-execution of inputs.

Guaranteeing timely consumption of data requires additional language and type constructs to specify which inputs and uses are time-critical, along with either static checking algorithms to disallow programs that may incorrectly consume stale data or additional runtime mechanisms to ensure that a program will not consume stale data.

Concurrency While most current intermittent systems use single-core micro-controllers and have no parallelism, recent work [Ruppel and Lucia 2019] supported interrupt-based concurrency through transactions. Modeling concurrency requires modeling interrupts and asynchronous events and updating the language with synchronization commands.

2.2 Related Work

The ideas presented in this paper are related to work in intermittent systems, fault tolerance and crash consistency in files systems, and formal persistent memory models. We first discuss the most related works in verified crash consistency and persistent memory models, and then how our work relates to existing intermittent systems, particularly those that deal with inputs or reactivity.

Crash consistency The failure and recovery problems of intermittent systems are similar to those of crash consistency on concurrent programs and file systems. A file system can crash at any time and must not exhibit unspecified behaviour after recovering. Developing formal specifications and verifiable file systems is an important research goal [Joshi and Holzmann 2007], particularly to guarantee correctness in the presence of crashes [Bornholt et al. 2016; Chen et al. 2015; Ernst et al. 2016; Ntzik et al. 2015; Schellhorn et al. 2014; Sigurbjarnarson et al. 2016].

Bornholt et al. [Bornholt et al. 2016] create a framework for generating crash consistency models. A crash consistency model specifies the allowed behaviour of a file system across crashes. Their crash consistency theorem relates the crashy fs trace to a canonical program trace with no crashes. In contrast, as we model non-deterministic sensor inputs, there is no single canonical trace even for executions with no crashes. The model consists of litmus tests and an operational semantics of the file system that models both volatile core state and durable disk state. Our semantics additionally model checkpoints.

The verification tool Yggdrasil [Sigurbjarnarson et al. 2016] uses crash refinement to aid programmers in developing verified file systems. Programmers must write specification and consistency invariants of their system. Then the verification is process is modular, allowing developers to swap in different implementation of system components as long as they meet the specification. The focus of our work is on defining correct specifications of intermittent system behaviour, including whether different implementations are in fact equivalent. While our correctness theorem is similar to crash refinement, we do not use Yggdrasil to verify our specifications as Yggdrasil uses file system abstractions, e.g., inode layouts and disc models, that don't apply to intermittent systems, which interact directly with memory. We additionally model the effects of inputs.

Crash Hoare Logic (CHL) [Chen et al. 2015] and fault-tolerant resource reasoning [Ntzik et al. 2015] are proof automation tools that extend Hoare triples with crash conditions to verify file system implementations. Using CHL has a high programmer proof burden because the programmer must specify the correctness invariants and recovery procedures and prove the recovery procedure correct. Thus, a large portion of our work is a prerequisite to using CHL; we define intermittent correctness invariants, which is non-trivial. We additionally show that existing recovery procedures are in fact *in*correct. Using CHL for proof automation once we have defined intermittent correctness is not immediately possible as CHL does not provide primitives for checkpoints and only explicitly models non-volatile state, not the mixed-volatility state typical on an intermittent system. Additionally, the crash conditions for the Hoare triples should capture the intermediate states at which a crash could occur, and must be specified for every procedure. For a set of file system procedures, capturing these intermediate states is not onerous, as each procedure interacts with only a few blocks of the disk. In contrast, we model intermittent execution traces of programs, which makes enumerating crash conditions complicated and time-consuming. Ntzik et al. [Ntzik et al. 2015] do consider both volatile and non-volatile resources, though they also require enumerating the non-volatile states possible after a procedure crashes. The authors use their framework to prove the soundness of an ARIES recovery mechanism. They model updates to the durable state at page granularity, and undoing a transaction requires rolling back all updates to pages modified by the transaction. In contrast, the intermittent systems we model are designed to roll back the minimum set of updates necessary to

(ostensibly) guarantee correctness. Neither of these frameworks model non-deterministic sensor inputs.

Unlike the works above, which deal with verifying the file system itself, Koskinen et al. [Koskinen and Yang 2016] automatically verify crash recoverability at the program level. Our approach is most similar to this work. Our correctness theorem is similar, defining correctness in terms of a simulation relation and observational equivalence between the continuous and intermittent executions. In contrast to our work, this model assumes that underlying system operations will be correct. Their method analyzes control-flow and reduces crash recoverability to reachability: if control-flow cannot reach an error state, it will be correct. The definition of a recovery mechanism is that given a state q_k in the original program, after a crash the mechanism brings the program back to q_k after transitioning through some recovery states. While this is clearly the desired behaviour of a recovery mechanism, this definition does not consider the internal state of the recovery mechanism and is not expressive enough to capture the behaviour of existing intermittent checkpoint systems. After fully executing the recovery procedure, an intermittent program is not in an equivalent state to before the crash. Identifying what differences are allowable in the recovered state so that further execution eventually brings the program to a consistent state is a key contribution of this work.

Persistent memory formalisms Persistent memory has been used for whole systems [Narayanan and Hodson 2012], entirely non-volatile processors [Ma et al. 2015a,b], and heap structures [Coburn et al. 2011; Volos et al. 2011]. There are formalisms exploring persistency models [Pelley et al. 2014, 2015] for reasoning about data on non-volatile systems and parallel persistency [Blelloch et al. 2018]. Other work looks at defining linearizability [Izraelevitz et al. 2016b] for persistent objects on concurrent systems. While these are useful correctness properties, current intermittent hardware is single-core and has no thread-level concurrency.

Weak persistency semantics have been formalized for TSO memory models [Raad and Vafeiadis 2018], for ARMv8 [Raad et al. 2019a], and for Intel x86 [Raad et al. 2019b]. In [Raad et al. 2019a], the authors introduce a declarative semantics for reasoning about persistency. Among memory persistency formalisms, our approach is most similar to this one, but we are at a higher level; there are differences in scope and the language features provided. These persistency models reason about the allowable differences between the order in which instructions execute and the order they persist to memory on multi-threaded programs. This scope introduces (needed) complexity into the declarative semantics, but the devices we target do not have multi-threading and expose no difference between execution and persist order. We do not currently benefit from this complexity, but in future work we may need to integrate with these models to guarantee assumptions, e.g., checkpoint atomicity, that are currently upheld by the simple hardware. Moreover, our modeling language provides inputs and checkpoints.

Runtime systems for crash consistency Runtime systems that attempt to provide crash consistency on database systems have similar functionality to runtime systems for intermittent execution, but generally do not provide re-execution of inputs, necessary for data freshness.

JustDo logging [Izraelevitz et al. 2016a] targets hybrid persistent systems. JustDo explicitly avoids re-executing code for better performance. iDo [Liu et al. 2018], also targeting hybrid systems, identifies idempotent instruction sequences to reduce the number of locations to be logged. Idempotence has also been used as a correctness criterion for fault tolerance in distributed systems [Ramalingam and Vaswani 2013]. Idempotent processing [De Kruijf and Sankaralingam 2013; de Kruijf et al. 2012] has been posed as an alternative recovery mechanism to checkpoint-logging and re-execution, but does not allow re-executing inputs, which sometimes is necessary for intermittent systems to provide fresh sensor readings.

Other work [Ben-David et al. 2019] provides a construction to automatically make accesses to shared memory and algorithms persistent. Intermittent systems need all executing code to be checkpointed or in transactions, not just shared data structures.

Runtimes for Intermittent Systems In this paper, we explicitly model DINO [Lucia and Ransford 2015] as a basic checkpointing system, Ratchet [Woude and Hicks 2016] for the idempotent region variant, Alpaca [Maeng et al. 2017] as an example of task-based redo logging, and Chinchilla [Maeng and Lucia 2018] for undo logging. Hibernus [Balsamo et al. 2016, 2015] is a *just-in-time* checkpoint system that dynamically inserts checkpoints and does not re-execute code, but suffers timeliness violations. Mayfly [Hester et al. 2017] is the first work to describe the timeliness problem and implements a programming model to enforce timeliness using an external timekeeper and explicit programmer annotations. Capybara [Colin et al. 2018] is a reconfigurable energy-harvesting platform that allows flexible atomicity and reactive events. Homerun [Kang et al. 2018] also explores atomicity for I/O events. Coati [Ruppel and Lucia 2019] and InK [Yildirim et al. 2018] explore event-driven intermittent systems. None of these works provide formal definitions or guarantees of correctness for either memory consistency or timely processing of inputs.

EDB [Colin et al. 2016] and Ekho [Zhang et al. 2011] are frameworks for debugging intermittent systems, and ScEpTIC [Maioli et al. 2019] is a tool for detecting bugs caused by write-after-read patterns. The EH Model [Miguel et al. 2018] provides a way of reasoning about the architectural and software consequences of energy availability and intermittent system design choices.

Dahiya et al. [Dahiya and Bansal 2018] create a formal model for verifying via translation validation that instrumented intermittent programs are equivalent to continuous ones. They do not consider repeated input operations or how checkpoints must behave for programs to be correct.

Inputs on intermittent systems IBIS [Surbatovich et al. 2019] identifies and characterizes bugs caused by repeated inputs in intermittent systems. The authors provide only a bug detection tool, not a correct runtime system, nor formal correctness invariants. We provide a formal proof of a sound version of the algorithm the authors use in their tool, as well as a correct runtime system. We discuss in detail the differences and similarities of the algorithm presented in this paper versus the algorithm in the IBIS tool in Section 9.3.

Developed most recently, TICS [Kortbeek et al. 2020] is a runtime system that uses an external timekeeper and programmer annotations to avoid consuming stale data. Rather than regather data, the runtime reruns expiration checks after rebooting, so that any stale data will not be processed. This approach avoids any consistency errors associated with re-executing inputs, but can also miss processing any input events if power failures are frequent. Moreover, this approach requires external time-keeping hardware.

Samoyed [Maeng and Lucia 2019], Sytare [Berthou et al. 2017] and RESTOP [Arreola et al. 2018] look at retaining the peripheral state of input devices, not at memory correctness issues caused by repeated input operations.

3 BACKGROUND AND MOTIVATION

Our work is motivated by emerging intermittent execution models and their varied correctness definitions. In this section, we review the fundamentals of checkpoint-based intermittent execution and show by example how existing models are not correct in the presence of input operations. Any intermittent execution model must ensure forward progress and preserve state. An intermittent execution progresses only when energy is available. Power fails when energy is exhausted, erasing the device's execution context and volatile state, including registers and all data stored in volatile memory. By default, the system then restarts from the start of `main()` and naively-written code makes no forward progress. To make progress, an intermittent system can periodically save its execution context and restart from that execution context on reboot; a common mechanism for

saving state is a statically placed *checkpoint*. A checkpoint is an operation that stores some memory state and execution context in non-volatile memory, preserving it across a power failure [Balsamo et al. 2016, 2015; Jayakumar et al. 2014; Lucia and Ransford 2015; Maeng and Lucia 2018; Mirhoseini et al. 2013; Ransford et al. 2011; Woude and Hicks 2016].

To be correct, an intermittent execution should generate the same result as a continuous execution. Multiple partial executions followed by a complete execution should have the same behaviour as *some* continuous execution. Code between checkpoints must execute *idempotently*. Unfortunately, a checkpoint system that saves only volatile execution context and volatile memory [Ransford et al. 2011] may be incorrect. Prior work [Lucia and Ransford 2015; Woude and Hicks 2016] identified that write-after-read (WAR) dependencies on non-volatile locations cause non-idempotent behavior and adjust the checkpointed data accordingly. We next discuss how WAR dependencies cause problems.

3.1 Write-After-Read (WAR) Dependencies

An intermittent execution may produce an incorrect result if the execution writes a value into non-volatile memory before a power failure and the execution reads that updated value after rebooting. Consider the example shown in Figure 2 (a). On the left of the figure is a small program. The next column shows execution traces illustrating the WAR problem. In the initial execution, the branch at line 1 is taken. After executing through line 4, there is a power failure. The column then shows the re-execution of the code. The re-execution assumes that the checkpoint restores only volatile state (i.e., control state) and retains non-volatile variables' values. This time, the execution completes and yields state N'_4. A continuous program would finish with the memory in state N_4. The state in N'_4 contains a different value for x. The re-execution does not idempotently update x because x depends on w. When the re-execution reads w into x at line 2, the read produces the (incorrect) value of w written before the power failure. The example shows that the re-execution is non-idempotent because w is involved in a WAR dependence, which we call a **WAR Variable**.

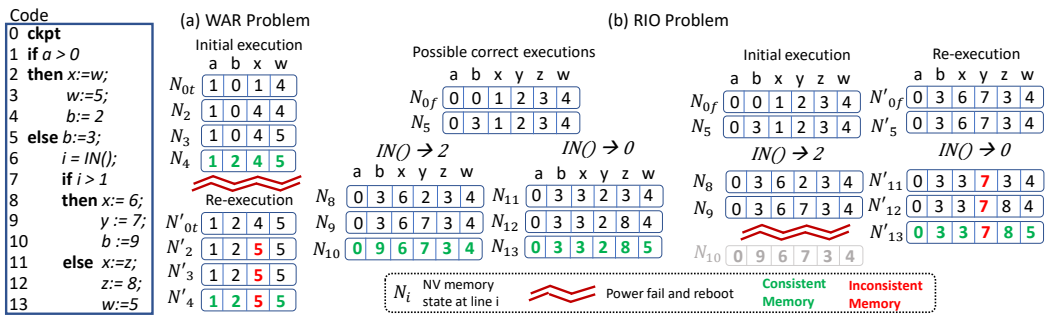

Fig. 2. An example program illustrating WAR and RIO problems

To ensure that code containing WAR dependencies executes idempotently, existing checkpoint-based systems must add *potentially inconsistent* variables to the checkpoint [Lucia and Ransford 2015; Maeng and Lucia 2018] and restore those variables with the checkpoint after a power failure. In the example, w is potentially inconsistent because a write to w before the power failure may be visible to a read after restarting from the checkpoint. At line 0, a checkpoint system needs to save a *version* of w with the checkpoint. Prior work observed that not all WAR dependences lead to inconsistency [Lucia and Ransford 2015; Maeng et al. 2017; Woude and Hicks 2016]. If w had been written before being read, e.g. if line 1 was instead $w := 3$; if $a > 0$, then w need not be checkpointed. The read on line 1 would always see a consistent value.

Checkpointing WAR variables that are not write-dominated is the current state-of-art in ensuring idempotent re-execution, which is *insufficient* for correctness, due to RIOs [Surbatovich et al. 2019].

3.2 Repeated Inputs Cause Incorrect Behavior

Applications that target low-power embedded systems rely heavily on peripheral devices, such as sensors and radios. A program stores in a variable the result of an input operation. In an intermittent execution, a program may execute an input operation before a power failure, and then repeat that input operation after a failure, in both cases fetching a fresh, usable value. However, repeating the input operation can lead to incorrect behavior when a program's control- or data-flow depends on the result of that input operation. We refer to a re-executed input as a **RIO**: a **R**epeated **I**nput **O**peration that can generate a different value each execution.

Continuing with Figure 2 (b), the columns to the right show how a RIO causes incorrect behavior. The example now assumes that all variables are in non-volatile memory and that the system now checkpoints variables involved in potentially inconsistent WAR dependencies at line 0, saving and restoring w and z. The starting memory N_{0f} has $a \mapsto 0$, so the initial branch is not taken. Instruction $i = \mathsf{IN}()$; on line 6 reads an input value (e.g., from a sensor). Depending on the sensor reading, a continuous execution could correctly end with either state N_{10} or N_{13}.

An intermittent execution may produce a result different from both N_{10} and N_{13}. Such an execution may first get an input greater than 1 at line 6, causing the branch at line 7 to be taken. Power then fails and the program restarts. After the restart, the input is less than 1, and the branch at line 7 is not taken. The final state is N'_{13}, which is inconsistent with all correct, continuously-powered outcomes because the RIO is not idempotent.

An intermittent execution with inputs is correct if it corresponds to a continuously-powered execution, regardless of the inputs. Here, the RIO causes different branches to be taken and non-volatile variable y is written on only one of them. Checkpoint systems that version WAR variables do not handle y's RIO problem because y is not a WAR variable. *No existing checkpoint or task-based intermittent execution system correctly handles these non-idempotent RIOs.* A correct checkpoint system must store y's value at the checkpoint and restore it on reboot. We refer to variables that are affected by RIOs (e.g., y) as **RIO variables**. Preventing inputs from re-executing, as by placing a checkpoint immediately after the operation, is not an adequate solution as some inputs must be re-executed to be timely [Hester et al. 2017]. This paper fills the gap left by RIOs, formally and with a practical system implementation that makes intermittent systems robust to RIOs.

4 SYSTEM ASSUMPTIONS AND FORMAL MODEL

We define a language to model checkpoint-based intermittent execution with inputs, providing the syntax and semantics for both continuously-powered and intermittent executions. First, we explain the lower-level system assumptions to justify our choice of modeling language.

Target System Assumptions Our target intermittent systems use low-end microcontrollers (MCUs) such as the TI MSP430FR series [TI Inc. 2020a]. These are single-threaded, single-issue, in-order compute cores. The MCUs have embedded, on-chip volatile SRAM or DRAM and non-volatile Flash, FRAM, or STT-MRAM. These architectures often lack caches or have only a simple write-through cache to avoid repeated non-volatile memory accesses. Unlike prior work in persistent memory targeting more complex architectures [Blelloch et al. 2018; Izraelevitz et al. 2016b; Raad and Vafeiadis 2018; Raad et al. 2019b,a], we need not reason about concurrency or persist order due to write-back caches or other microarchitectural optimizations. We thus realistically assume that execution and persist order are the same and that the compiler never re-orders an instruction past a checkpoint.

Syntax Our simple language includes accesses to volatile memory, accesses to non-volatile memory, and branch statements. We include arrays but omit general pointer arithmetic. We also omit functions calls and unbounded loops. These omissions do not affect our ability to capture the

Values	v	::=	$n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{in}(\tau)$	*Configuration*	Σ	::=	(τ, κ, N, V, c)
Expressions	e	::=	$x \mid v \mid e_1\ \mathsf{bop}\ e_2 \mid a[e']$	*Cont. config.*	σ	::=	(τ, N, V, c)
Instructions	ι	::=	$x := e \mid a[e] := e' \mid x := \mathsf{IN}() \mid$	*Volatile mem.*	V	:	M
			$\mathsf{skip} \mid \mathsf{checkpoint}(\omega) \mid \mathsf{reboot}(n)$	*Non-vol. mem.*	N	:	M
Commands	c	::=	$\iota \mid \iota; c \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$	*Context*	κ	::=	(N, V, c)
Memory loc.	loc	::=	$x \mid a[n]$	*Read obs.*	r	::=	$\mathsf{rd}\ loc\ v \mid r, r$
Chckpnted loc.	ω	::=	$\omega, x, \mid \omega, a^n$	*Observation*	o	::=	$[r] \mid \mathsf{in}(\tau) \mid \mathsf{reboot}$
Mem. mapping	M	::=	$\mathsf{Loc} \rightarrow \mathsf{Val}$				$\mid \mathsf{checkpoint}$

Fig. 3. Syntax and Semantic Constructs

behavior of existing intermittent execution models. Existing systems [Lucia and Ransford 2015; Maeng et al. 2017] do not allow recursive function calls, so any code in a function body can be inlined. Including general pointer arithmetic would not change the correctness invariants we present, as the definitions consider memory locations directly, but would complicate the implementation of any checkpoint algorithm (discussed in Section 9), as the alias sets of the memory locations in the definitions would need to be tracked as well. Unbounded loops can be handled by extending our infrastructure with loop invariants, which do not introduce technical difficulties but unnecessarily complicate the presentation. Though simple, this modeling language suffices to illustrate clearly the key challenges in defining memory-consistent intermittent execution models.

We summarize the syntax in Figure 3. We write v to denote values, which can be numbers n, the boolean values true and false, and inputs $\mathsf{in}(\tau)$, representing the input gathered at time τ. Expressions, denoted e, can be variables, values, binary operations of expressions, or an array element. Array lengths are fixed and all array indices are assumed in bounds; this assumption is necessary for correctness and memory safety for real C code is orthogonal [Grossman et al. 2002] and beyond our scope. Instructions, denoted ι, consist of assignments to variables and arrays, checkpointing, rebooting, skip, and synchronous input operations $\mathsf{IN}()$. We write ω to denote the set of non-volatile variables and arrays that must be saved with a checkpoint to avoid inconsistency. We call these variables *checkpointed locations*. In the example in Figure 2, $\mathsf{checkpoint}(\{w, y, z\})$ would precede the if statement on line 1, which include both WAR and RIO variables. We write a^n to represent all the locations in the array a. That is: each a^n in ω represents the set of locations $\{a[1], \cdots, a[n]\}$. We often omit the bounds n and write a directly. We assume that checkpoint operations are manually inserted into code (e.g., like DINO [Lucia and Ransford 2015]). Section 2.2 of the TR details the algorithm to compute ω for WAR variables, as in existing systems. Section 7 describes our novel algorithm for computing ω for RIOs. A program is a command c, which is an atomic instruction, a sequence of instructions, or an if branching statement. We lift all the branches to the top-level for ease of explanation. Any program with general branching statements can be re-written to our language and bounded loops can be un-rolled to `if` statements.

Semantics for Intermittent Execution We focus on intermittent execution semantics. The rules for continuously-powered execution semantics are standard and can be found in TR Section 1.2. First, we define the necessary runtime constructs in Figure 3. Memory is a mapping from a location, which is either a variable or an array index, to a value. We distinguish between volatile and non-volatile memory, which are disjoint. The method of specifying where a variable resides varies; systems may provide abstractions [Colin et al. 2018; Maeng et al. 2017], do automatic compiler analysis [Maeng and Lucia 2018], or assume that all data is non-volatile [Woude and Hicks 2016]. A configuration Σ is a tuple consisting of a timestamp τ, a checkpoint context κ, non-volatile memory state, volatile memory state, and a command to be executed c. The checkpoint context κ consists of the non-volatile data, volatile data, and command saved at the last checkpoint. The timestamp is the logical time at the current configuration. Executing commands and evaluating expressions

$$\text{I/O-CP-CkPt}\quad \frac{}{(\tau, \kappa, N, V, \mathsf{checkpoint}(\omega); c) \overset{\mathsf{checkpoint}}{\Longrightarrow} (\tau+1, (N|_\omega, V, c), N, V, c)}$$

$$\text{I/O-CP-PowerFail}\quad \frac{\mathsf{pick}(n)}{(\tau, \kappa, N, V, c) \Longrightarrow (\tau+1, \kappa, N, \mathit{reset}(V), \mathsf{reboot}(n))}$$

$$\text{I/O-CP-Reboot}\quad \frac{\kappa = (N, V, c)}{(\tau, \kappa, N', V', \mathsf{reboot}(n)) \overset{\mathsf{reboot}}{\Longrightarrow} (\tau+n, \kappa, N' \lhd N, V, c)}$$

Fig. 4. Selected semantic rules

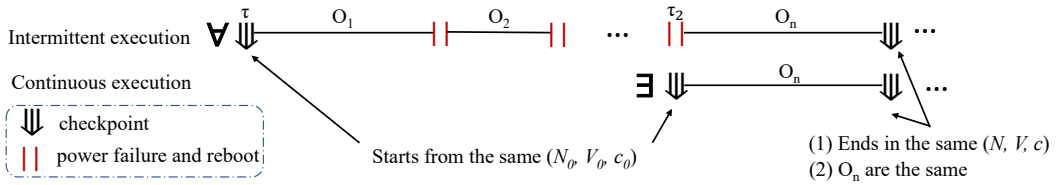

Fig. 5. Illustrating the correctness definition

generate observations, which are memory reads, input reads, or occurrences of a checkpoint or reboot instruction. These observations are used for facilitating definitions of correctness.

The semantic rules are of the form: $(\tau, \kappa, N, V, c) \overset{O}{\Longrightarrow} (\tau', \kappa', N', V', c')$, where O is a list of observations. We write $\Longrightarrow$ to denote an intermittent execution, and $\longrightarrow$ to denote a continuous execution. We show rules specific to intermittent execution in Figure 4. The rule I/O-CP-CkPt states that at a checkpoint, the context κ is updated to include the portion of the current non-volatile memory whose domain is ω, the current state of volatile memory, and the current command to be executed. We write $m|_\omega$ to denote the part of m, whose domain is ω. The system generates the checkpoint observation and proceeds to execute the command after the checkpoint. This operation is assumed to be atomic, implemented at the low-level with an atomic flag update and double buffering to ensure there is always a valid last checkpoint. The I/O-CP-PowerFail can execute at any step, as power failures can occur at any time, and is thus non-deterministic. When power fails, volatile memory is reset, the current command is lost, and a positive integer n is picked at random for the subsequent reboot instruction. The system then transitions to reboot. On reboot, I/O-CP-Reboot partially restores non-volatile memory using the checkpoint. This rule applies even before the first checkpoint is reached as κ is a piece of memory initialized with the starting V and c and empty non-volatile portion. We write $m_1 \lhd m_2$ to denote the memory resulting from updating locations in m_1 with the values of those locations in m_2. The reboot restores volatile memory and the command to the values in the checkpoint's context. A reboot is added to the observation sequence, and the timestamp increases to $\tau + n$. The increase captures the idea that a power failure can have an arbitrary duration.

We write $N, V \vdash e \Downarrow_r v$ to mean that with memories N and V, expression e evaluates to value v with observation r. For example, $N_{0f}, V \vdash w \Downarrow_{\mathsf{rd}\, w\, 4} 4$ is a sub-derivation when line 2 of the program in Figure 2 is executed. The rules are standard, so we omit them for space.

5 FORMALLY CORRECT INTERMITTENT EXECUTION

A program can be executed correctly in an intermittent model if and only if any completed intermittent execution trace of the program corresponds to a continuous execution, w.r.t. the program context and the observation sequence. More precisely, comparing an intermittent and continuous execution, the program contexts, including volatile and non-volatile memory and

the command to be executed must be the same by the end of the program. To show this, we examine execution segments between checkpoints. Each partial re-execution from a checkpoint can observe a different value produced by the same input. Consequently, observation sequences from partial executions are *not necessarily* prefixes of the same continuous execution. To be correct, the observation sequence of the *final* re-execution segment in an intermittent execution must match a continuous execution with the same input results. We illustrate this in Figure 5. Time advances from left to right. The top line is an intermittent execution trace. We detail a segment between two checkpoints, marked by down arrows. Multiple power failures and reboots are present in these segments, demarcated by red parallel bars. The observed memory reads O_i are shown on top of the line. For each such intermittent execution, the correctness property dictates the existence of a continuous execution—the second line—such that the read accesses from the latest reboot to the checkpoint (O_n) match the read accesses of that continuous execution. Furthermore, the ending configuration of both executions at the checkpoint are the same (excepting the extra context κ in the intermittent configuration). The above holds for all execution segments, including the last.

To formalize this definition, we introduce additional notation and constructs to relate intermittent and continuous program contexts and observation sequences. We define the erasure of the configuration: $(\tau, \kappa, N, V, c)^- = (\tau, N, V, c)$ to relate the configurations at the checkpoints. We next formally relate the observation sequences.

$$\textsc{I-Rb-Base}\ \frac{}{O \leqslant^m O} \qquad \textsc{I-Rb-Ind}\ \frac{O_1' \leqslant^m O_2}{O_1, \mathsf{reboot}, O_1' \leqslant^m O_2} \qquad \textsc{Cp-Base}\ \frac{O_1 \leqslant^m O_2}{O_1 \leqslant_c^m O_2} \qquad \textsc{Cp-Ind}\ \frac{O_1 \leqslant^m O_2 \qquad O_1' \leqslant_c^m O_2'}{O_1, \mathsf{checkpoint}, O_1' \leqslant_c^m O_2, O_2'}$$

The rules use $\leqslant^m$ and $\leqslant_c^m$ to express the prefix requirements of the observation sequence of the intermittent execution (O_1) to the observation of the continuous execution (O_2). $\leqslant^m$ expresses a relation between an O_1 that may include reboots to O_2, and $\leqslant_c^m$ expresses a relation between an O_1 that may include both reboots and checkpoints to O_2. The crucial aspect of the observation relation is in rule I-Rb-Ind. An intermittent observation consisting of two observation prefixes O_1 and O_1' separated by a reboot relates to the continuous observation O_2 if the latter prefix relates to the continuous observation. The intuition of this rule is that the observation prefix of an intermittent execution before a reboot may read values produced by input operations. After the reboot, the input operations may return different values, so the old prefix should be discarded. The intermittent and continuous executions need only agree on observations after the most recent reboot.

The correctness of an intermittent execution model is defined as follows:

Definition 1 (Correctness of Intermittent Execution)**.**
A program c can be correctly intermittently executed if for all τ, N, V, O_1 s.t. $(\tau, \emptyset, N, V, c) \overset{O_1}{\Longrightarrow}{}^* \Sigma$, where the program in Σ is skip *(i.e., the program terminated), then $\exists O_2, \tau_2, \sigma$ s.t. $(\tau_2, N, V, c) \overset{O_2}{\longrightarrow}{}^* \sigma$, $\tau_2 \geq \tau$, $O_1 \leqslant_c^m O_2$, and $\sigma = (\Sigma)^-$.*

Figure 6 illustrates the relations in the correctness definition by revisiting the code from Figure 2. Assume there is a checkpoint immediately preceding the branch on a, which saves the state of $\{w, y, z\}$. Consider a power failure after the assignment to y on line 9, which lasts for 4 timestamps. The column on the left shows the intermittent execution state and the right shows the continuous execution state, starting at a later time, time 8. The final state of the executions (N_5' and N_5) are equal, despite differences in their execution paths and intermediate states. Further, the observation sequences relate: $\mathsf{checkpoint}, \mathsf{in}(1), \mathsf{reboot}, \mathsf{in}(9), \mathsf{rd}\, z\, 3 \leqslant_c^m \mathsf{in}(9), \mathsf{rd}\, z\, 3$. There happens to be no reads before the reboot. If there were, they would not need to match the reads on the right, as they

are not in the final (successful) re-execution. To ensure this correctness property, a checkpointed set must include *both* WAR and RIO variables, which we will explain in the next section.

6 PROVING MEMORY CONSISTENCY

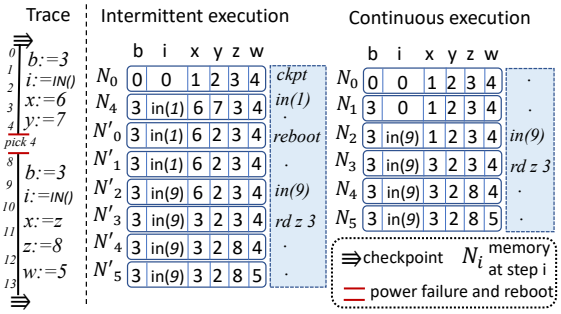

Fig. 6. Illustrating the correctness definition by example

Non-volatile memory updates of an intermittent execution can diverge from non-volatile memory updates of a continuous execution in two key ways. A continuous execution updates memory through the writes of a single execution of the program. An intermittent execution updates memory through multiple executions of a prefix of the program and by overwriting some set of non-volatile memory locations after a reboot. To be correct, an intermittent execution model must ensure that any inconsistencies caused by these different update traces have resolved by the next checkpoint. In this section, we define for the first time invariants on the checkpointed set and non-volatile memory updates that allow us to prove an intermittent execution model correct. A runtime system that upholds these invariants will provably update memory consistently, no matter the algorithm used in implementation.

6.1 Locations to Checkpoint

The correctness of an intermittent execution model requires restoring at reboot a set of non-volatile memory locations ω (e.g., $\{y, z, w\}$ in Figure 6). We refer to the minimal set of potentially inconsistent non-volatile memory locations as $\omega^{\dagger}$ and observe that precisely computing this set is difficult in general. It is safe to over-approximate $\omega^{\dagger}$. One safe over-approximation is all of non-volatile memory, $\omega^{all} = \mathsf{dom}(N)$, but ω^{all} is inefficient because it requires unnecessarily checkpointing many variables. Existing systems use ostensibly less conservative over-approximations, such as a subset of the variables involved in WAR dependence, ω^{WAR} [Hicks 2017; Lucia and Ransford 2015; Maeng et al. 2017; Maeng and Lucia 2018; Woude and Hicks 2016], but as we have shown, this set misses any variables that are inconsistent due to RIOs. Next, we describe how to to check that all potentially inconsistently written locations are checkpointed. We then define an algorithm to statically analyze the program and add those locations to the checkpoint in Section 7.

To express that a checkpoint includes the subset of potentially inconsistent variables, we introduce two judgments: $\Vdash_{WAR} c : \mathsf{ok}$, which checks variables inconsistent due to WARs and $\Vdash_{RIO} c : \mathsf{ok}$ which checks variables inconsistent due to RIOs. We formally define rules for this judgment here. Rules for the WAR checking judgment are in TR Section 2.1.

Our checking algorithm leverages taint-tracking to find branch operations that depend on an input, then ensures that if a non-volatile location is written on any path of such a branch, that location is either written on all paths of the branch or is in the checkpointed set. We call variables that are written on all paths regardless of inputs *must-write variables*.

There are two top-level judgments for commands, depending on whether control is currently input-dependent — i.e., *tainted* — or not: $N; M \Vdash_{taint} c : \mathsf{ok}$ and $N; I; M \Vdash_{RIO} c : \mathsf{ok}$ respectively. Likewise there are two judgments for instructions: $N; M' \Vdash_{taint} \iota : \mathsf{ok}$ or $N; I; M \Vdash_{RIO} \iota : \mathsf{ok}$. N is the set of checkpointed variables, I is the set of variables that depend on inputs (control and data dependence), and M is the set of variables written prior to executing c. The $\Vdash_{taint}$ judgment does not use I as the judgment itself carries the information that control is tainted. The $\Vdash_{taint}$ check for instructions ensures that all writes access checkpointed variables in N or are must-write variables,

which captures how RIOs' effects may transitively taint variables through dependences. We show rule RIO-Assign-tainted as an example.

$$\text{RIO-Cp}\ \frac{\omega; \emptyset; \emptyset \Vdash_{RIO} c : \mathsf{ok}}{N; I; M \Vdash_{RIO} \mathsf{checkpoint}(\omega); c : \mathsf{ok}} \qquad \text{RIO-If-NDep}\ \frac{I \cap rd(e) = \emptyset \quad N; I; M \Vdash_{RIO} c_i : \mathsf{ok} \quad i \in [1,2]}{N; I; M \Vdash_{RIO} \mathsf{if}\ e\ \mathsf{then}\ c_1 \mathsf{else}\ c_2 : \mathsf{ok}}$$

$$\text{RIO-If-Dep}\ \frac{I \cap rd(e) \neq \emptyset \quad M \Vdash^{mstWt} \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : M' \quad N; M' \Vdash_{taint} c_i : \mathsf{ok}\ i \in [1,2]}{N; I; M \Vdash_{RIO} \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : \mathsf{ok}} \qquad \text{RIO-Assign-tainted}\ \frac{x \in (M \cup N)}{N; M \Vdash_{taint} x := e : \mathsf{ok}}$$

We explain selected rules for commands. Rule RIO-Cp applies to a command starting with a checkpoint and checks the remaining command c using the checkpoint's checkpointed set ω and an empty I and M. Rule RIO-If-NDep checks a branch that is input-independent. RIO-If-Dep checks an input-dependent branch , identifying its must-write variables up to the next checkpoint using auxiliary judgment $M \Vdash^{mstWt} c : M'$ (not shown). The resulting M' includes variables written on the path up to the branch ({i, b} in the example) and any variables that must be written on all paths from the branch ($\{x\}$ in the example). The key difference between these two rules is that RIO-If-NDep checks its sub-commands with $\Vdash_{RIO}$ and the must-write set M, whereas RIO-If-Dep checks its sub-commands with $\Vdash_{taint}$ and M'. The intuition for this is that an input-independent branch will always evaluate the same way. Any variable written will be written on any re-execution, even if it is not written on all paths. An input-dependent branch may not take the same path, however, so the sub-command must be checked with the must-write set of all paths from the branch.

Returning to Figure 2, the rules check the if statement at line 7 using judgment $\Vdash_{taint}$. The M' is $\{i, b, x\}$. To satisfy the check, N must include y, z, w.

6.2 Defining the Effect of Input on Execution Prefixes

Input can cause an intermittent execution's memory state to vary across re-executions, behaviour impossible on a continuous execution. We introduce notation showing how input interacts with a program execution. $O|_{\mathsf{in}}$ denotes the sequence of input values in observation sequence O. Let "trace" refer to the sequence of execution states with observations annotated on top of each transition generated by an intermittent execution. We define $Run(\sigma, \mathcal{I}, c)$ to be a trace starting at σ, ending in command c with the input sequence $\mathcal{I}$ and without checkpoints. Formally:

$$Run(\sigma, \mathcal{I}, c) = \{T \mid T = \sigma \xrightarrow{O}{}^{*} (\tau, N, V, c) \wedge O \text{ contains no checkpoints} \wedge \mathcal{I} = O|_{\mathsf{in}}\}$$

We write $Run(\sigma, \mathcal{I}, CP)$ to denote the trace ending in the nearest checkpoint. Note that given a set of inputs, $Run(\sigma, \mathcal{I}, c)$ is always a singleton set with a uniquely determined trace: inputs already executed from the *previous* checkpoint to the current execution point are fixed, and any execution to the current point with those same inputs will yield the same trace. There are, however, multiple possible traces from an arbitrary point c to the *next checkpoint* due to inputs yet to execute.

We write $Wt(T)$ to be the write set of a trace and $FstWt(T)$ to be the set of variables written before they are read in a trace. We define the set of locations that must be written on any input:

$$MstWt(N, V, c) = \{loc \mid \forall \tau, N, V, c, \forall \mathcal{I}, \forall T \in Run((\tau, N, V, c), \mathcal{I}, CP), loc \in Wt(T)\}$$

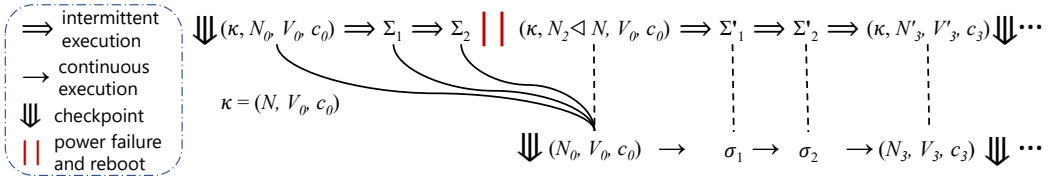

Fig. 7. Illustrating the correctness proofs and invariants

The must-write set of the example program is $\{b, x\}$. We define the set of locations that must be first written before being read, no matter the input value, below:

$$MFstWt(N, V, c) = \{loc \mid \forall \tau, N, V, c, \forall \mathcal{I}, \forall T \in Run((\tau, N, V, c), \mathcal{I}, CP), loc \in FstWt(T)\}$$

This set contains locations in non-volatile memories that are first written (not read) on all possible runs through the checkpointed region, starting from c. We call the set the *must-first-write* set, or $MFstWt$. In the example program, the $MFstWt = MstWt$, but the distinction is important as the must-first-write set will not include variables that have a (non-write-dominated) WAR dependence.

6.3 Invariants Relating Memories

As shown in Figure 6, the non-volatile memories are not always the same. To prove the intermittent execution model correct, we need to identify relations between the intermittent execution and the continuously-powered execution configurations. Therefore, we define relations between memories in intermittent and continuous executions, as illustrated in Figure 7. We relate the memory states at the same execution point with dashed lines and relate an intermittent execution's memory state at any point to a continuous execution's memory at its initial point with solid lines. These relations describe how memory locations are allowed to differ, while still converging to the same memory and observation by the next checkpoint, the key invariant for correctness.

Arbitrary-point Memory Relation (Solid Line) We define the relation of the memory state of an intermittent execution N_{int} at an arbitrary point to the memory state of a continuous execution N_{cont} at its initial point:

Definition 2 (Related memories between current and initial execution point). $N_{ckpt}, N, V, c \vdash N_{int} \sim N_{cont}$ *iff* $\mathsf{dom}(N_{int}) = \mathsf{dom}(N_{cont})$, $\forall loc \in N_{int}$ *s.t.* $N_{int}(loc) \neq N_{cont}(loc)$, $loc \in N_{ckpt} \cup MFstWt(N, V, c)$

For correctness, all locations that differ between N_{int} and N_{cont} must be in the checkpointed set N_{ckpt} or in the must-first-write set of the initial command c. The checkpoint reverts writes to data in the checkpointed set on each reboot. Re-execution after a reboot over-writes each variable in the must-first-write set because *every* path to the next checkpoint writes to variables in that set, regardless of input. In Figure 6, variables b, i, x, y differ between N_0 and N_4. Given the initial memory state, execution always takes the false path from the first branch and always writes b and i. All paths write x and it is always first written to. Finally, y is checkpointed. Together these actions reconcile all differences between N_0 and N_4.

Same-point Memory Relation (Dashed Line) We next define the relation of the memory state of an intermittent execution N_{int} and a continuous execution N_{cont} at the same point :

Definition 3 (Related memories at the same execution point). $\tau, N, V, c, c', \mathcal{I} \vdash N_{int} \sim N_{cont}$ *iff* $\mathsf{dom}(N_{int}) = \mathsf{dom}(N_{cont})$ *and* $\forall loc \in N_{int}$ *s.t.* $N_{int}(loc) \neq N_{cont}(loc)$,

- $loc \in MFstWt(N, V, c)$
- *let* $\{T\} = Run((\tau, N, V, c), \mathcal{I}, c')$ *and the last state of* T *is* (τ', N_{int}, V, c'), $loc \in MstWt(N_{int}, V, c')$ *and* $loc \notin Wt(T)$

The relation is parameterized with a timestamp τ and an input sequence $\mathcal{I}$. The relation uses τ and $\mathcal{I}$ to define the singleton set $Run(\sigma, \mathcal{I}, c')$ from the initial point to the current point.The relation has access to all parameters because at each point in the execution, all prior input values are already concrete and timestamps are given. Beyond sharing the same domain, the definition restricts locations that differ between the memories. If such a location is not written from the initial execution point to the current one, then the location must be in the must-first-write set of the entire trace *and* must be written between the current execution point and the next checkpoint. The intuition is that differing locations must be written to on all possible paths through the remainder of the trace for the intermittent and continuous traces to converge to the same state. Moreover, re-execution following any path dictated by fresh input values should not read locations that differ, which would cause non-idempotent re-execution (i.e., first written to in every execution).

In Figure 6, the continuous and intermittent states differ at corresponding points N_0 and N'_0, N_1 and N'_1, and N_2 and N'_2. Starting with N_0, the execution writes b, i, x regardless of input. After stepping to states N_1, N'_1, b cannot differ because the execution from its initial point to the current point wrote to b. After stepping to states N_2, N'_2, i cannot differ because the execution from its initial point to the current point wrote to i. i's written value is the same in both executions because we choose a continuous execution that reads the input at time 9, which the intermittent execution also reads. x is not yet written, but will be on all paths. At states N_3 and N'_3, all locations must be the same. x, b, i have been written between the initial and current execution point and none are written between the current execution point and the next checkpoint. z is written to on the current path, but is not in the must-first-write set of the entire trace, nor are w and y.

6.4 Proving Correctness

We prove the following theorem:

Theorem 4 (Correctness). *If* $\Vdash_{WAR} c : \mathsf{ok}, \Vdash_{RIO} c : \mathsf{ok}$ *then c can be correctly intermittently executed.*

We actually prove a stronger theorem that relates an intermittent execution up to checkpoints to a corresponding continuous execution. Only at each checkpoint, are the memories guaranteed to sync up between the two executions.

The proof requires augmenting the semantics with variable taint tracking, dynamically marking all input-dependent locations. We leverage standard taint tracking semantics rules and omit them here. The proof follows the structure in Figure 7 and requires the following properties: (1) arbitrary intermittent configurations relate to the continuous initial configuration, (2) each intermittent configuration relates to a continuous configuration at the same execution point, and (3), after reboot, we can switch from the relation illustrated by the solid line to that of the dashed line and after checkpoint, we can switch from dashed line to solid line.

With properties (1) – (3) established, the proof of Theorem 4 is by induction over the structure of the intermittent execution trace. First over the number of checkpoints to show that, from checkpoint to checkpoint, the resulting memories are the same and memory reads are idempotent. For each segment between checkpoints, we induct over the number of reboots and use the relations in the previous section to relate memory at each execution step. Note that no existing intermittent execution model checks $\Vdash_{RIO} c : \mathsf{ok}$; none meets a reasonable correctness definition in the presence of I/O, which is one of the key results of this work. We show the full proofs in TR Section 6.6.

7 COLLECTING EXCLUSIVE MAY-WRITES

Given our correctness definition, an intermittent execution model must collect and checkpoint not only WAR variables, but also RIO variables. Our algorithm identifies a (safe) conservative

over-approximation of this set. If a variable might be written on one side of a branch and not the other, and the branch condition might change from one re-execution to the next due to a RIO, then the variable should be checkpointed. In other words, RIO variables are in the *exclusive may-write* set for some command c – i.e., the set of variables that may be written on exclusively one side of some future branch, but that will not be written unconditionally. Using the exclusive may-write set, a simple rewriting algorithm can transform a program with empty checkpoint sets into a program that correctly checkpoints RIO variables.

Our algorithm computes exclusive may-write sets, and identifies input-dependent (or *tainted*) branches. Given a branch if e then c_1 else c_2, if e is not (transitively) input-dependent, the branch's outcome is the same on every re-execution. The first branch in Figure 2 is never taken under N_{0f}, and the write to b happens regardless of line 6's input; the write is not in the exclusive may-write set. If e is input-dependent, the branch outcome depends on input and later writes are candidates for exclusive may-write. In Figure 2, the branch at line 7 is input-dependent and its exclusive may-write set is $\{w, y, z\}$, each of which are written on one, but not both sides of the branch. Our algorithm uses taint analysis to identify input-dependent branches, and adds to ω exclusive may-write variables for input-dependent branches.

Collection per instruction. Two sets of rules collect the exclusive may-write set X, must-write set M, and input-dependent variable set I for instructions: $X; M; I \Vdash_{RIO} \iota : X'; M'; I'$ and $X; M \Vdash_{taint} \iota : X'; M'$. The $\Vdash_{RIO}$ rules apply to ι in commands from an input-independent branch and $\Vdash_{taint}$ rules apply to ι in commands from an input-dependent branch. The primary distinction for instruction level rules is that I does not need to be collected in the $\Vdash_{taint}$ rules as our branches never merge. We explain selected $\Vdash_{RIO}$ rules; rules for $X; M \Vdash_{taint} \iota : X'; M'$ are similar with taint tracking removed.

$$\frac{}{X; M; I \Vdash_{RIO} x := \mathsf{IN}() : X; M \cup x; I \cup x}\ \text{I/O-Get}$$

$$\frac{I \cap rd(e) \neq \emptyset}{X; M; I \Vdash_{RIO} x := e : X; M \cup x; I \cup x}\ \text{I/O-Assign-dep}$$

$$\frac{I \cap rd(e) = \emptyset \qquad x \in I}{X; M; I \Vdash_{RIO} x := e : X; M \cup x; I \setminus x}\ \text{I/O-dep-clear}$$

$$\frac{I \cap rd(e) \neq \emptyset}{X; M; I \Vdash_{RIO} a[e] := e' : X \cup a; M; I \cup a}\ \text{I/O-Arr-loc}$$

All assignments add the variable x to M, since x must be written on the current command. Rule I/O-Get adds x to I. If any assignment has an expression that reads a value in I, the assigned location is also added to I, as taint propagates to x (rule I/O-Assign-dep). Conversely, assigning a location in I to an input-independent expression removes that location from I, effectively clearing its taint (rule I/O-dep-clear). Propagating taint to an array element would cause the entire array a to be conservatively tainted. If an array index is tainted, then a is added to X because the written array element may differ in each re-execution (rule I/O-Arr-loc).

Collection for commands Two sets of rules define collection and rewriting for commands: $X; M; I \Vdash_{RIO} c \longrightarrow c' : X'$ and $X; M \Vdash_{taint} c \longrightarrow c' : X'; M'$, with the same distinction as the ι rules between $\Vdash_{RIO}$ and $\Vdash_{taint}$. These rules compute the exclusive may-write set X' and must-write set M' up to a checkpoint in c and rewrite the checkpoint command to use the collected X' as ω. Rewriting an instruction ι directly uses the rules we introduced in the previous paragraph to collect relevant variants (e.g, X, M) and rewrites of itself. Much of the complexity for commands

is collecting exclusive may-write and must-write sets from (nested) branches. We show key rules:

$$\frac{I \cap rd(e) \neq \emptyset \quad X; M \Vdash_{taint} c_i \longrightarrow c_i' : X_i; M_i \; i \in [1,2]}{X; M; I \Vdash_{RIO} \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \longrightarrow \mathsf{if}\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2' : (X_1 \cup X_2 \cup M_1 \cup M_2) \setminus (M_1 \cap M_2)}\ \text{I/O-If-Dep}$$

$$\frac{\emptyset; M \Vdash_{taint} c_i \longrightarrow c_i' : X_i; M_i \qquad i \in [1,2]}{X; M \Vdash_{taint} \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \longrightarrow \mathsf{if}\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2' : (X_1 \cup X_2 \cup M_1 \cup M_2) \setminus (M_1 \cap M_2); (M_1 \cap M_2)}\ \text{I/O-If-Tainted}$$

$$\frac{\emptyset; \emptyset; \emptyset \Vdash_{RIO} c \longrightarrow c' : X'}{X; M \Vdash_{taint} \mathsf{checkpoint}(); c \longrightarrow \mathsf{checkpoint}(\mathsf{X}'); c' : X; M}\ \text{CP-Tainted}$$

I/O-If-Dep applies to an input-dependent branch encountered when control is not yet tainted. The rule switches from $\Vdash_{RIO}$ to $\Vdash_{taint}$ in the premises because the condition expression is input dependent. The exclusive may-write set is the union of exclusive may-write and must-write sets from each side of the branch, minus the intersection of the must-write sets. I/O-If-Tainted applies to branches encountered while in $\Vdash_{taint}$ This rule must also collect the branch's must-write set, which is the intersection of the must-write sets from both sides of the branch, unioned with the must-write set from before the if statement.

Exclusive-May-Write Collection Example Figure 8 illustrates exclusive may-write collection, using code from Figure 2. We letter each branch outcome path and number each branch instruction. Collection begins in the $\Vdash_{RIO}$ rule, as the branch is not tainted. For path A, X_A is $\emptyset$. For path B, the must-write set is $\{b, i\}$ before branching on i. We apply $\Vdash_{taint}$ rules to paths C and D because i is input-dependent. X_C and X_D are empty, $M_C = \{b, i, x, y\}$, and $M_D = \{b, i, x, z, w\}$. Then the must-write set for branch 2 is the intersection' M_C and M_D, which is $\{b, i, x\}$. The may-write set of paths C and D is the union of the segments' X and M sets: $\{b, i, w, x, y, z\}$. The exclusive may-write set removes variables that must be written on both paths (i.e., $\{b, i, x\}$); so $X_2 = \{w, y, z\}$. Collection propagates $X_B = X_2$ and the final exclusive may-write set for branch 1 is the union of those for paths A and B: $X_1 = \{w, y, z\}$. For this code snippet, $\{w, y, z\}$ have to be checkpointed.

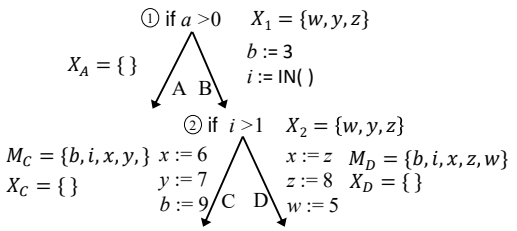

Fig. 8. Example of exclusive may-write set collection.

Inserting ω At a checkpoint, collection inserts the exclusive may-write set returned from rewriting c as ω (rule CP-Tainted). The rules re-write the command after a checkpoint using the judgment $\Vdash_{RIO}$, *even if control is tainted.* This re-writing is correct because after checkpointing, the input operation will never be re-executed and execution is deterministic until the next input operation.

Correctness of the Algorithm We prove that the collection rules are safe with regard to the checking rules, which is a condition in Theorem 4.

Lemma 5. *If* $\Vdash_{RIO} c \longrightarrow c' : X$ *then* $\Vdash_{RIO} c' : \mathsf{ok}$.

The lemma states that if a command has been rewritten using the collection algorithm, the rewritten command is safe with respect to RIOs. Note that separating the collection algorithm from the checking $\Vdash_{RIO} c' : \mathsf{ok}$ enables modular proofs. A different collection algorithm does not change the overall correctness proof as long as it can be shown to be safe w.r.t. the checking rules.

8 EQUIVALENCES BETWEEN SYSTEMS

The checkpoint system presented is based on DINO [Lucia and Ransford 2015], but many others exist. We additionally formalize four alternative implementations—undo logging [Maeng and Lucia 2018], redo logging, idempotent regions [Woude and Hicks 2016], and a task-based execution model Alpaca [Maeng et al. 2017]. Instead of reproving the correctness theorems for each system, we define and prove a bi-simulation relation between the basic system and each alternative showing that they are equivalent. While these systems differ in mechanism and performance, the equivalence result shows that their correctness criteria are the same, allowing us to implement the algorithm in Section 7 for the more performant Alpaca. We relegate bi-simulation for undo logging and idempotent regions to TR Section 4, as DINO implements a conservative form of undo-logging, and idempotent regions differ from the basic model by constraints on checkpoint placement.

8.1 Redo Logging

State restoration can be implemented with redo-logging, which works by logging memory updates during execution and committing the log to memory upon reaching a checkpoint. A redo logging context κ_{RL} is of the form $(\mathcal{L}, V, c, \omega)$, where $\mathcal{L}$ is a log and ω has the same meaning as before.

We illustrate the key behaviour of redo-logging and its relation to the basic model in Figure 9. `Ckpt Code` is a simple program that initializes the variables x, y, z, checkpoints $\{x, y\}$, swaps their values using a volatile variable a, and takes another checkpoint before continuing with the rest of the program. Columns (a-c) on the right show the program's state at each point in the execution. Column (a) shows the execution of the basic model starting from the checkpoint at line 4. Column (b) shows a redo-logging execution. Volatile memory V and command c are equivalent at each step and omitted. DINO starts with N_c containing the values of x, y at the checkpoint. Redo-logging starts with an empty log and $\omega = \{x, y\}$. In any checkpoint region, the domain of the log will be ω. If the program contains an assignment to a non-volatile location in ω (lines 6,7), the update is placed directly into the log, leaving non-volatile memory untouched. Otherwise the variable is updated directly in non-volatile memory (line 8). On reboot, the redo log clears but leaves non-volatile memory untouched, as all updates to locations in ω reside in the log only. The basic model updates non-volatile locations in ω to the values from the checkpoint. As the program re-executes, updates to locations in ω are redone, either directly to non-volatile memory (a) or to the log (b). When the program reaches the next checkpoint on line 9, the Redo model applies the log to non-volatile memory, committing the changes.

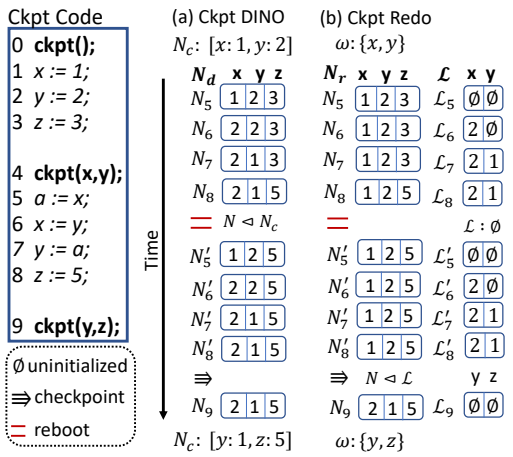

Fig. 9. Illustrating the relation of DINO to redo-logging

A key part of the bi-simulation relation between DINO and redo-logging is that if the value of a location in redo-log non-volatile memory N_r is not equal to the same location in DINO's non-volatile memory N_d, then that location is in the domain of ω and therefore in the log. The value in the log is equal to the value in N_d, as both reflect its latest update. Consequently, $N_d = N_r \triangleleft \mathcal{L}$. At reboots and checkpoints $\mathcal{L}$ is empty, and the non-volatile memory, volatile memory, and command of both models are equal. We formalize this relation and prove bi-simulation in TR Section 4.4.

8.2 Task-Based Systems

Task-based systems [Colin and Lucia 2016; Colin et al. 2018; Hester et al. 2017; Maeng et al. 2017; Ruppel and Lucia 2019] require the programmer to structure an intermittent program as a series of transaction-like tasks. Updates within a task are not be visible to other tasks (including re-executions of the same task) until the task commits. Task semantics rely on either undo or redo logging. The key differences between checkpoint- and task-based systems are in their memory abstraction and control structure.

$$\begin{array}{llll} \textit{Task Map} & T & ::= & \cdot \mid T, i \mapsto (\omega, c) \\ \textit{Instr.} & \iota & ::= & \cdots \mid \mathsf{toTask}(i) \end{array}$$

A task-based program has no checkpoints, and is instead a series of tasks. The context is augmented with a task map T, from task IDs to a checkpoint set ω and command for the corresponding task. The context κ_{TSK} for intermittent execution is a pair (T, i) consisting of the task map and the ID of the current task. A special instruction $\mathsf{toTask}(j)$ ends the current task, transitioning to task j (rule TSK-Trans).

$$\frac{\kappa_{TSK} = (T, i) \qquad T(j) = (\omega, c)}{(\kappa_{TSK}, \mathcal{T}s, \mathcal{T}p, \mathcal{T}\iota, \mathsf{toTask}(j)) \overset{\text{transition}}{\Longrightarrow}_{TSK} ((T, j), \mathcal{T}s \triangleleft \mathcal{T}p, \emptyset, \mathcal{T}\iota, c)} \text{ TSK-Trans}$$

The task-system has a memory abstraction of task-shared memory $\mathcal{T}s$, task-local memory $\mathcal{T}\iota$, and task-private memory $\mathcal{T}p$. A programmer assigns variables accessed in multiple tasks to task-shared memory and variables used only in a single task to task-local. Task shared memory must be non-volatile, and task-local can be split into volatile and non-volatile sections, $\mathcal{T}\iota_V$ and $\mathcal{T}\iota_N$ respectively. Task-private variables are hidden from the programmer and used to implement logging. A task must initialize task-local variables by writing them before reading them. Like basic checkpoints, the command in each task is well-formed given the checkpoint set: $\omega \Vdash_{WAR} c : \mathsf{ok}$ and $\omega \Vdash_{RIO} c : \mathsf{ok}$.

$$\begin{array}{llll} \textit{Commands} & c & ::= & \cdots \mid \mathsf{goto}\ \ell \\ \textit{Code context} & \Psi & ::= & \Psi \cdot \mid \ell_i : \mathsf{checkpoint}(\omega); \mathsf{c} \end{array}$$

To prove equivalence between a task-based and a checkpoint-based system, we first translate task transitions to checkpoint commands. We augment the basic language with a goto command and a code context Ψ that includes a set of labeled program points, each beginning with a checkpoint. Correctness still holds; extending the main proofs is trivial as goto does not change memory.

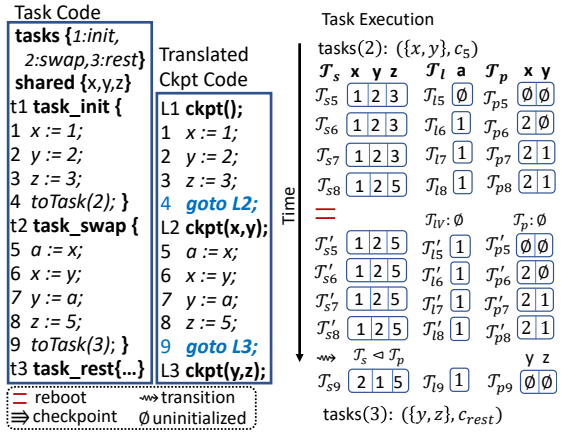

Fig. 10. Relating checkpoint redo-logging to tasks

We specify a translation relation from tasks to a redo-logging code context, written $T \rightsquigarrow \Psi$. The key idea is that each task can be translated to a checkpoint followed by the translated task command: $i \mapsto (\omega, c_t) \rightsquigarrow \ell_i : \mathsf{checkpoint}(\omega); c_r$. Here $c_r = [\![c_t]\!]$ and task transitions are translated to gotos: $[\![\mathsf{toTask}(i)]\!] \rightsquigarrow \mathsf{goto}\ \ell_i$. The translation of the rest of the constructs recursively translates the sub-terms and returns the same construct when an instruction is reached. We show a task-based version of the program to swap x and y and

its translation to a checkpoint program in Figure 10. The program is a series of three tasks: `init`, `swap`, and `rest`. The variables x, y, z are shared between the tasks. Variable a is local to task `swap`. In the translation, each `toTask` is replaced by a `goto` whose label points to a checkpoint followed by the translated task command.

Using these constructs and the translated program, we relate a configuration of a task-based system to that of redo-logging. We show the formal relation below and an execution of the task program on the right side of Figure 10. The redo-log execution of the translated program is the same as in column (b) of Figure 9 as the only difference to the original `Ckpt` code is the addition of the blue `goto` instructions.

$$\frac{\begin{array}{c}\Sigma_{RL} = (\kappa_{RL}, N_r, V_r, c_r) \qquad \Sigma_{TSK} = (\kappa_{TSK}, \mathcal{T}s, \mathcal{T}p, \mathcal{T}\iota, c_t) \\ \kappa_{RL} = (\mathcal{L}, V_c, c_c, \omega_r) \qquad \kappa_{TSK} = (T, i) \qquad T(i) = (\omega_t, c_{tt}) \qquad T \rightsquigarrow \Psi \qquad [\![c_t]\!] = c_r \qquad [\![c_{tt}]\!] = c_c \\ \omega_t = \omega_r \qquad \mathcal{T}p = \mathcal{L} \qquad \mathcal{T}\iota = \mathcal{T}\iota_V, \mathcal{T}\iota_N \qquad N_r = \mathcal{T}s \cup \mathcal{T}\iota_N \qquad V_r \approx \mathcal{T}\iota_V \qquad \mathsf{dom}(V_r) \subseteq \mathsf{dom}(\mathcal{T}\iota_V)\end{array}}{\Sigma_{TSK} \nrightarrow \Psi, \Sigma_{RL}}$$

The redo-log $\mathcal{L}$ and $\mathcal{T}p$ are equivalent. Any updates to locations in ω will be placed into $\mathcal{T}p$, not $\mathcal{T}s$. If $\mathcal{T}\iota$ is entirely volatile, then $\mathcal{T}s$ and N_r will also be equivalent. Otherwise, if a is stored in a non-volatile location, then N_r will be equal to the union of $\mathcal{T}s$ and $\mathcal{T}\iota_N$. If a in a volatile location, $\mathcal{T}\iota_V$ will be equal to the redo log volatile memory, once a has been initialized ($V_r \approx \mathcal{T}\iota_V$). This qualification is necessary as $\mathcal{T}\iota_V$ is cleared on reboot (after line 8), whereas redo logging restores the checkpointed volatile memory. Translated commands will always be well-formed w.r.t. task-local memory, however, so there can be no *read* to a volatile memory location before it is initialized. Thus any memory accesses on the two systems will be equivalent. This property does not hold for any arbitrary redo-log program; consider a redo-log program where the assignment to a occurred before the checkpoint — the first access to a after the checkpoint would be a read. At line 9, task `swap` transitions to task `rest`. A task transition commits $\mathcal{T}p$ to $\mathcal{T}s$, resets $\mathcal{T}p$, and transfers control to the specified task, switching the task reference in the context to the new task. The translated program jumps to the label $L3$, corresponding to the command $\mathsf{checkpoint(y, z)}; [\![\mathsf{c_{rest}}]\!]$. When it executes this checkpoint, it updates N_r with the log and the clears the log. As $\mathcal{T}p$ and $\mathcal{L}$ are equivalent, the updated non-volatile memory is still equivalent to the union of the task-shared and non-volatile task-local memories. Furthermore, as no writes are left to occur, $V_r = \mathcal{T}\iota_V$. We prove equivalence in TR Section 5.

9 IMPLEMENTATION

We implemented the exclusive may-write (EMW) collection algorithm in Section 7, which consists of both write-set collection and taint-tracking, and combined its output with Alpaca's runtime system, yielding an intermittent execution runtime with safe access to I/O. We built two variants: *EMW*, which backs up EMW sets for all branches (correct, but conservative), and *taint-optimized EMW*, which calculates the EMW set for only input-dependent branches. EMW requires no code changes, but backs up some unnecessary variables. Taint-optimized EMW requires very minor code changes to annotate input operations, but backs up a smaller, far less conservative variable set.

9.1 System

We implemented EMW collection in LLVM [Lattner and Adve 2004] and to Alpaca, we added support to back up EMW variables. Alpaca is a task-based system and can use either undo or redo logging [Maeng et al. 2019]. As Section 8 shows, a task-based program translates into an equivalent checkpoint-based program with either style of logging. We use undo-log Alpaca since it is the most efficient Alpaca variant.

9.2 Limitations Due to C Features

The algorithm in Section 7 is sound for our simple modeling language. However, Alpaca extends C, which has several features not present in the modeling language, such as merging branches, arbitrary pointers, and non-recursive functions (which do not checkpoint state), leading to a few differences between the formal statement of the algorithm and the implementation. Merging branches and functions do not require changes to the algorithm. All paths can be explored even if a branch merges A write in an unconditionally executed block will execute on all paths and be in the must-write set. Functions are treated as inlined, leveraging the lack of recursion.

Taint tracking Pointers and functions complicate taint tracking. Taint-optimized EMW collection is sound only if taint does not propagate indirectly, as through pointer arithmetic (e.g., y points to N, the address of a tainted location, x points to $N-1$, $x++$. The algorithm would miss that x points to a tainted location). Our implementation propagates taint through function parameters and return values. A call with a tainted parameter taints the corresponding argument. A tainted return value taints the store of the return value in the function's caller. A function may taint a reference parameter, and our implemented algorithm taints the corresponding parameter in the function's caller (similar to [Surbatovich et al. 2019]). These aliasing limitations of taint tracking do not affect the soundness of taint-agnostic EMW collection, and none of our test programs had indirect taint propagation, which would compromise soundness.

Write set collection To compute write sets, we assume that task-shared variables must be stored to directly, and cannot be aliased through a task local pointer. All Alpaca applications followed this behaviour. This direct access of task-shared variables allows the algorithm to compute may and must write sets precisely, apart from arrays. This limitation is due to our prototype implementation and is not inherent to the formal algorithm. To extend the prototype to compute safe EMW sets with complex aliasing, must-write sets should include must-alias only, and may-write sets should include may-alias locations. As in the formal algorithm, any array written to on a tainted branch is conservatively (safely) put into the EMW set.

9.3 Algorithm Implementation

We implement taint tracking as a fixed-point dataflow analysis, propagating data taint in a traversal. At the end of each traversal, the algorithm examines any instructions that introduced inter-procedural dataflow, and adds any new sinks to a worklist from which to start future traversals. When no new inter-procedural flows are identified, the algorithm is at a fixed point and stops. The analysis returns a list of tainted instructions. As we mentioned earlier, we could under-taint, though we did not observe any under-tainting.

EMW collection is a separate analysis directly implemented from our formal description. Taint-enabled EMW collection narrows the scope of the analysis, using the taint tracking analysis result and calculating the EMW sets for conditionals that are tainted only. EMW collection returns a map from a function to its computed EMW sets.

We modify Alpaca's undo-logging compiler analysis to use the EMW set information. Alpaca maintains a per-function set of WAR variables to undo-log in their called task. We modify Alpaca to include a function's EMW set with the function's WAR variables, which are then passed together to Alpaca's existing undo-logging instrumentation pass, which allocates undo log storage, creates checkpoint metadata, and adds undo-logging instrumentation.

Comparison to IBIS' algorithm The specification of the algorithm in IBIS is unsound. Even assuming perfect pointer aliasing and taint propagation, IBIS could still miss bugs. IBIS detects RIO bugs by calculating the may-write sets of paths off tainted branches and comparing them. If the

may-write sets are equal it reports no bug. Consider if($tainted\ e$) $x := 1$; if($e2$) $y := 1$ else $z := 1$; else $x := 1$; $y := 1$; $z := 1$. IBIS would report no bugs as the may-write sets are the same, but y and z are in the EMW set and thus potentially inconsistent. Additionally, conservatism, whether due to implementation decisions such as aliasing or inherent in static analyses (such as opaque path conditions) hampers the usability of IBIS. Any variable falsely identified as potentially inconsistent generates a confusing false-positive bug report that the programmer must reason through, whereas the EMW runtime safely adds it to the checkpoint at little runtime cost.

10 EVALUATION

The goal of the evaluation is to show that modifying Alpaca to correctly support input operations is practically efficient. We evaluate Alpaca's baseline system, a variant that checkpoints all data identified by our exclusive may-write (EMW) analysis without taint tracking, and a variant that checkpoints all data identified by our EMW analysis refined with taint-tracking support. Our data show that our analysis provides correctness, through checkpointing both WAR and RIO variables, with low run time and memory overheads. EMW alone has very low overheads with *no programming effort* and EMW plus taint analysis has *virtually no time overhead* and very low memory overhead, but asks the programmer to annotate input operations. To demonstrate the programmability benefit of our analysis, we perform case studies, showing that it is non-trivial (sometimes complicated) to fix RIO bugs manually, even using a state-of-the-art bug detection tool (IBIS [Surbatovich et al. 2019]), but trivial using our analysis.

10.1 Benchmarks

We use benchmarks from the IBIS paper [Surbatovich et al. 2019], obtained from the authors, as they run on Alpaca and have input bugs. There are 11 programs: 7 drivers and low-level applications from TI-RTOS [TI Inc. 2020b] and 4 from Alpaca [Maeng et al. 2017]. The TI-RTOS programs are `bmp` a pressure sensor driver, `hdc` a humidity sensor driver, `elink`, a radio implementation, `mpu`, a magnetometer driver, `opt`, an optical sensor driver, `temp`, a temperature sensor driver, and `wsn`, a sensor data aggregator. The Alpaca programs are `ar`, activity recognition, `bc`, bit counting, `cem`, a compressive logger, and `cuckoo`, a cuckoo filter. IBIS found RIO bugs in `mpu`, `opt`, `temp`, and `wsn`.

10.2 Performance Overhead of EMW Tracking

Checkpointing data added to ω by EMW analysis guarantees correctness and causes only low run time and memory overheads. Figure 11 shows run time normalized to Alpaca for plain Alpaca (blue), Alpaca with checkpointing for EMW sets (yellow), and Alpaca with checkpointing for I/O-tainted EMW data only (green). Each bar averages 100 run times on continuous power with fixed inputs, and error bars are a 95% confidence interval. Plain Alpaca is fastest, *but incorrect* because it does not back up variables made inconsistent by RIOs. EMW with taint-tracking has virtually no time overhead (0 - 0.7%) because the analysis checkpoints the few variables from the EMW set required for correctness. Checkpointing full EMW sets has higher overhead, ranging from negligible to nearly 15% for `cuckoo`. The higher overheads of EMW alone demonstrate the need for taint tracking, which eliminates overheads, requiring only that the programmer annotate input operations.

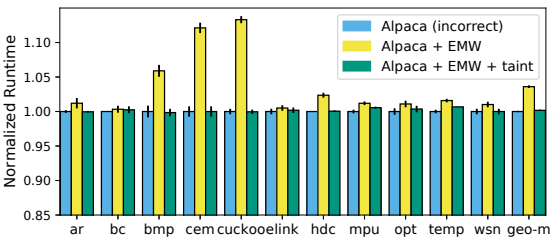

Fig. 11. Normalized runtimes of Alpaca, Alpaca with EMW sets, and Alpaca with taint-optimized EMW sets

Checkpoint and Memory Overheads Taint-optimized EMW analysis checkpoints only the few variables necessary to avoid RIOs and WARs, while using EMW analysis conservatively requires checkpointing many more variables, at a higher memory overhead. Figure 12 shows the bytes needed to back up variables identified by EMW and taint-optimized EMW. The left bar in each pair is for EMW alone, and the right is for taint-optimized EMW. Each bar is broken up into bytes due to variables in a WAR dependence (blue), untainted variables conservatively in the exclusive-may-write set (yellow), and tainted variables in the exclusive may-write set (green). Any array is double-buffered, potentially causing a large difference in log size (bmp) Together, the yellow and green segments in an EMW bar include all tainted and untainted EMW variables. The taint-optimized EMW bar eliminates the yellow segment, identifying input-tainted EMW variables only and illustrating the conservatism in EMW alone that requires checkpointing more variables.

Figure 13 quantifies the normalized memory overhead caused by the increase in logged variables, accounting for all checkpoint storage and metadata (including array-size dependent metadata [Maeng et al. 2017]). Taint-optimized EMW reduces memory overheads significantly compared to EMW alone.

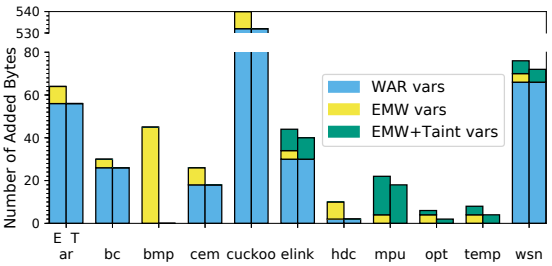

Fig. 12. Space needed to back up variables identified by EMW and taint-optimized EMW analysis, by category

Programmability Benefits of EMW Using taint-optimized EMW analysis is a simpler solution for repeated I/O than manually changing code. Prior work detects RIO bugs using an ad hoc approximation of our taint-optimized EMW analysis [Surbatovich et al. 2019], suggesting that the programmer fix bugs. Taint-optimized EMW has low overheads and requires the programmer to annotate input operations only, which is simple. Manually finding and fixing bugs is relatively more complex. IBIS [Surbatovich et al. 2019] advises re-initializing I/O-tainted variables at the start of the task that taints them. This strategy moves each variable into the *must-first-write* set, causing it to be written on every task execution and eliminating the need to include it in ω. However, unconditional initialization in a task may change a program's meaning if the value overwritten by the initialization is important. Instead, a programmer could create their own backup copy of the variable and save its value on the first write in the task; doing so amounts to manually applying undo-logging, guided by IBIS's bug report.

Manual undo-logging requires modifying each use, and adding an initialization and a backup operation, requiring changing at least $\sum(\forall loc \in \mathsf{EMW}, 2 + \mathsf{uses}(loc))$ lines of code. Concretely, mpu required 22 changes, opt required four, temp required seven, and wsn required 17. As reported by IBIS, only these benchmarks had RIO bugs that required code changes to fix.

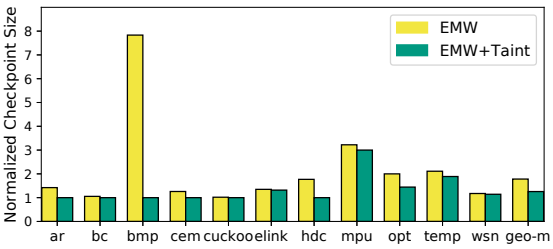

Fig. 13. Normalized checkpoint size overhead caused by EMW tracking and taint-optimized EMW tracking.

Manual fixing is not only onerous, but ultimately duplicative: manual backup introduces a WAR dependence on a task-shared variable and any non-idempotent EMW variable will be added to the checkpoint anyway by the WAR analysis after fixing manually. IBIS may also miss RIO bugs. Fixing only IBIS' reported bugs may result in still incorrect code. Our taint-optimized

EMW analysis eliminates the risk of manual code fixes and directly backs up the necessary data, ensuring correctness with reasonable overhead and essentially no programming burden.

11 EXTENDING THE FRAMEWORK

We use the presented framework to define and prove correctness with respect to memory consistency and input operations. There are more properties that must be reasoned about to truly develop provably correct, reliable intermittent systems. This framework serves as a foundation that can be built upon to reach that ultimate goal. We discuss the strategies to extend the framework to include other programming models or correctness properties. Some programming models can be modeled by straightforward extensions, such as changes to the execution state or logging mechanisms (Section 8). Others require more significant changes to the current framework, such as guaranteeing forward progress.

11.1 Extending Memory Consistency to Other Programming Models or Architectures

Extending the memory consistency theorem to cover systems with different checkpointing algorithms or architectural state is straightforward. The proof of the memory consistency theorem is built around two memory relations 1) current-to-initial execution point and 2) same execution point (Section 6.3). If the intermittent and continuous states satisfy relation 1, the states remain related after each reboot. If they satisfy 2, the states will have the same memory by the next checkpoint. To extend the framework to cover a new execution model, one can show that these relations hold directly, or one can show equivalence to the basic model as in Section 8. A model with differing checkpoint placement or additional language instructions may satisfy these relations directly, such as *just-in-time* checkpointing [Balsamo et al. 2015; Maeng and Lucia 2019] or idempotent regions [Woude and Hicks 2016]. If a model changes the state tuple or runtime constructs—e.g., the context or memory layout, additional architectural components—it is more practical to show correctness via bi-simulation, rather than tweaking the parameters of relations 1 and 2. In Section 8, we showed equivalence to systems with differing program models. Below we sketch an extension to a different architecture, one that uses a write-back cache.

A write-back cache commits an update to memory only when the update is evicted from the cache, either due to an eviction policy or through explicit flushing. Thus, the order in which updates execute may differ from the order they are evicted and committed. Our target hardware has a write-through cache, allowing us to assume identical persist and execution order in the presentation of the basic model (Section 4), but devices with write-back caches are reasonable future targets. We show that the framework can be simply extended to handle this change in architecture. The state tuple of the basic model should be extended with a cache C. An update to a location l may be made to the cache instead of N, and a future eviction from the cache updates N with the value, i.e., $N \triangleleft C[l]$. Note that this is almost exactly the behaviour of Redo logging (Section 8.1); updates to checkpointed locations are logged, and the log is committed at a checkpoint. Updates to N occur out of execution order. The key change to the semantics is that the commit must flush the newly added cache as well as the log, acting as a serialization point. Additionally, items in C can be flushed before reaching a checkpoint. This behaviour does not introduce new inconsistencies, however, as region re-execution is idempotent. Consider an execution that caches updates x and y, persisting only y before failing. y is after x in execution order but before x in persist order. If the first access to y after reboot is a write, the previously persisted value is never accessed. If the access is a read, then y had a WAR dependence. y is thus in ω and the update would have been made to the redo log. Adding a write-back cache to a sequential redo-log model thus requires only minor changes to the semantics and bi-simulation relation, as the model already commits at checkpoints and safely redoes partial flushes.

Undo-logging with a write-back cache is more complex. In the basic model presented, the back-up copies of variables are created at the checkpoint, so flushing after a checkpoint persists both cached updates from the previous region and all the backup copies. A more performant undo log that creates backup copies on demand could become inconsistent if the update to general non-volatile memory persists before the update to the log, and power then fails. A simple way to make caching correct is that any update to the log must be flushed immediately, so the log update always is persisted before the general update, but this destroys much of the benefit of having a cache for any variable in ω. This issue of persisting log data before program data is a known and well studied problem with logging on persistent memory [Chakrabarti et al. 2014; Genç et al. 2020; Raad et al. 2019a].

Updating the checkpoint semantics to flush (and fence) the cache makes a checkpoint a serialization point between checkpointed regions. The WAR:ok and RIO:ok checks are static, and thus put any variable that could potentially be inconsistent (including those visible to a re-execution of the current checkpoint region) into the checkpoint set. Any out-of-order persists or variables from partial flushes are thus either over-written during the course of re-execution, were made to a log (redo model) or are undone when applying the checkpoint set (basic, undo-log model).

As energy-harvesting devices develop to target more complex architectures, the modular next step is to add a hardware layer to the model to abstract away ordering details from the higher-level theorem. The instructions to fence and flush updates are ISA specific [Raad et al. 2019b,a]. As shown above, changes to the architecture need not dramatically effect the memory consistency theorem. The interface with the hardware layer should provide certain assumptions, e.g., checkpoints are a serialization point, updates are linearizable. The proofs of these assumptions would not change the main correctness theorem, but can instead draw from formalizations in prior work [Chakrabarti et al. 2014; Raad and Vafeiadis 2018; Raad et al. 2019b,a].

11.2 Towards Correctness Properties beyond Memory Consistency

Ensuring that programs are correct with respect to memory consistency is a crucial first step towards reliable intermittent computation, but there remain others, such as ensuring progress, correct timing, and concurrency (Section 2.1). The presented framework can be extended modularly to reason about these properties.

Forward Progress To be able to guarantee forward progress, any possible trace between checkpoints must not consume more energy than can fit in the energy buffer. Reasoning about the energy consumption of a trace requires creating an energy model. This energy model must model the full system, including peripherals, as energy consumption depends on all components on the board, not just the CPU. The prior work CleanCut [Colin and Lucia 2018] develops an energy model to guide programmers in creating appropriately sized tasks, but it is probabilistic, and furthermore does not consider the full system. Developing such a full-system, non-probabilistic energy model is a complex problem. While this energy model is necessary to reason about the forward progress property, it does not change the memory consistency correctness property presented in this work. Rather, an additional energy layer should be added to the framework. To be correct, any intermittent execution must correspond w.r.t. memory to some continuous execution, and additionally a trace between checkpoints must always take less energy than can fit in the energy buffer. Thus, we anticipate that energy modelling can be added to the current framework modularly.

Concurrency through Interrupts While there are not yet multi-core intermittently-powered devices, some research [Ruppel and Lucia 2019; Yildirim et al. 2018] addresses interrupt driven computation. In such execution models, inputs can be asynchronous and ephemeral —after reboot,

interrupts may not occur as they did before power-failure. Intermittent executions must be consistent, as in defined in the theorem presented here, but they must also correctly deal with concurrency, The updates to memory of any interrupt handlers—including those partially executed–and the main thread of program execution must be linearizable. Linearizability for persistent memory is a well-studied problem [Izraelevitz et al. 2016a,b; Liu et al. 2018; Raad et al. 2019a]. Extending the framework requires adding interrupts to the semantics and adding linearizability to the proof.

Time-sensitivity In this work, as in [Koskinen and Yang 2016], the continuous, non-crashy execution to which the intermittent, crashy execution corresponds can pause for arbitrary amounts of time, while the system recovers to a consistent state. Allowing these arbitrary pauses at any location can make the correctness definition too weak for programs whose behaviour depends on highly timing-sensitive input processing. The value an input operation returns depends on the time that it was gathered — arbitrary pauses within a sequence of input operations can produce program behaviour not possible on a continuous execution without pauses. To be correct w.r.t timing of inputs, an intermittent execution must not only correspond to some continuous execution, but that continuous execution must be one without pauses in time-sensitive regions. Which regions are time-sensitive is frequently application dependent. Robust reasoning about time-sensitivity requires adding language constructs to describe the time-constraints on data, as well as mechanisms to preserve the constraints. As with the properties above, adding time-sensitivity to the framework does not change the underlying memory-consistency theorem, but adds another constraint to correct intermittent execution.

12 CONCLUSION

We provide the first formal framework for examining the correctness of intermittent systems, w.r.t memory consistency. We show the framework's usefulness by using it to formalize intermittent systems with input operations, showing that many existing systems do not meet reasonable correctness criteria, and using the correctness invariants to implement a correct runtime system. We further extend the framework to show that a variety of existing systems are equivalent, indicating that the same correctness properties hold for all the modeled systems. This framework lays the foundation for formally defining intermittent system correctness, a crucial step towards the development of provably correct, reliable applications for intermittent systems. Future work should extend the framework to define properties beyond memory consistency, such as timeliness, forward progress, or concurrency.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their feedback, and members of the Abstract Research Lab for their insightful comments on initial drafts. This work was generously funded through National Science Foundation Award 2007998 and National Science Foundation CAREER Award 1751029.

REFERENCES

Alberto Arreola, Domenico Balsamo, Geoff Merrett, and Alex Weddell. 2018. RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems. *Sensors* 18 (01 2018). https://doi.org/10.3390/s18010172

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). 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). https://doi.org/10.1109/LES.2014.2371494

Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. 2019. Delay-Free Concurrency on Faulty Persistent Memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19)*. ACM, New York, NY,

USA. https://doi.org/10.1145/3323165.3323187

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

Guy E. Blelloch, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2018. The Parallel Persistent Memory Model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*. https://doi.org/10.1145/3210377.3210381

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

Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2660193.2660224

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

Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. https://doi.org/10.1145/1950365.1950380

Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. 2016. An Energy-interference-free Hardware-Software Debugger for Intermittent Energy-harvesting Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. https://doi.org/10.1145/2872362.2872409

Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2983990.2983995

Alexei Colin and Brandon Lucia. 2018. Termination Checking and Task Decomposition for Task-Based Intermittent Programs. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3178372.3179525

Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. A Reconfigurable Energy Storage Architecture for Energy-Harvesting Devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3173162.3173210

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

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 (PLDI '12)*. https://doi.org/10.1145/2254064.2254120

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

Bradley Denby and Brandon Lucia. 2020. Orbital Edge Computing: Nanosatellite Constellations as a New Class of Computer System. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3373376.3378473

Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, and Wolfgang Reif. 2016. Inside a Verified Flash File System: Transactions and Garbage Collection. In *Verified Software: Theories, Tools, and Experiments*, Arie Gurfinkel and Sanjit A. Seshia (Eds.). Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-29613-5_5

Francesco Fraternali, Bharathan Balaji, Yuvraj Agarwal, Luca Benini, and Rajesh Gupta. 2018. Pible: battery-free mote for perpetual indoor BLE applications. In *Proceedings of the 5th Conference on Systems for Built Environments*. ACM. https://doi.org/10.1145/3276774.3282823

Karthik Ganesan, Joshua San Miguel, and Natalie Enright Jerger. 2019. The What's Next Intermittent Computing Architecture. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. https://doi.org/10.1109/HPCA.2019.00039

Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. 2020. Crafty: Efficient, HTM-Compatible Persistent Transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3385412.3385991

Graham Gobieski, Brandon Lucia, and Nathan Beckmann. 2019. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3297858.3304011

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA. https://doi.org/10.1145/512529.512563

Josiah Hester and Jacob Sorber. 2017. Flicker: Rapid Prototyping for the Batteryless Internet of Things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17)*. https://doi.org/10.1145/3131672.3131674

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

Josiah Hester, Nicole Tobias, Amir Rahmati, Lanny Sitanayah, Daniel Holcomb, Kevin Fu, Wayne P. Burleson, and Jacob Sorber. 2016. Persistent Clocks for Batteryless Sensing Devices. *ACM Trans. Embed. Comput. Syst.* 15, 4, Article 77 (Aug. 2016). https://doi.org/10.1145/2903140

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

Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016a. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA. https://doi.org/10.1145/2872362.2872410

Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016b. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing*, Cyril Gavoille and David Ilcinkas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-53426-7_23

Neal Jackson, Joshua Adkins, and Prabal Dutta. 2019. Capacity over Capacitance for Reliable Energy Harvesting Sensors. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks (IPSN '19)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3302506.3310400

Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. 2014. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*. https://doi.org/10.1109/VLSID.2014.63

Rajeev Joshi and Gerard Holzmann. 2007. A Mini Challenge: Build a Verifiable Filesystem. *Formal Asp. Comput.* 19 (06 2007). https://doi.org/10.1007/s00165-006-0022-3

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

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

Eric Koskinen and Junfeng Yang. 2016. Reducing Crash Recoverability to Reachability. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA. https://doi.org/10.1145/2837614.2837648

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

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)*. https://doi.org/10.1109/MICRO.2018.00029

Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. https://doi.org/10.1145/2737924.2737978

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 (MICRO-50 '17)*. ACM, New York, NY, USA. https://doi.org/10.1145/3123939.3124533

Kaisheng Ma, Xueqing Li, Shuangchen Li, Yongpan Liu, John Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015a. Nonvolatile processor architecture exploration for energy-harvesting applications. *IEEE Micro* 35, 5 (2015). https://doi.org/10.1109/MM.2015.88

Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015b. Architecture exploration for ambient energy harvesting nonvolatile processors. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. https://doi.org/10.1109/HPCA.2015.7056060

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

Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2019. Alpaca: Intermittent Execution without Checkpoints. (2019). arXiv:cs.DC/1909.06951

Kiwan Maeng and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA. 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 '19)*. https://doi.org/10.1145/3314221.3314613

Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. 2019. On intermittence bugs in the battery-less internet of things (WIP paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM. https://doi.org/10.1145/3316482.3326346

J. San Miguel, K. Ganesan, M. Badr, and N. E. Jerger. 2018. The EH Model: Analytical Exploration of Energy-Harvesting Architectures. *IEEE Computer Architecture Letters* 17, 1 (Jan 2018). https://doi.org/10.1109/LCA.2017.2777834

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

Dushyanth Narayanan and Orion Hodson. 2012. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. https://doi.org/10.1145/2150976.2151018

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

Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. 2015. Fault-Tolerant Resource Reasoning. In *Programming Languages and Systems*, Xinyu Feng and Sungwoo Park (Eds.). Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-26529-2_10

Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. 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). https://doi.org/10.1109/MM.2015.46

Proteus Digital Health. 2015. Proteus Digital Health. http://www.proteus.com/. (2015).

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). https://doi.org/10.1145/3276507

Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019b. Persistency Semantics of the Intel-X86 Architecture. *Proc. ACM Program. Lang.* 4, POPL, Article 11 (Dec. 2019). https://doi.org/10.1145/3371079

Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019a. 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). https://doi.org/10.1145/3360561

Ganesan Ramalingam and Kapil Vaswani. 2013. Fault Tolerance via Idempotence. (2013). https://doi.org/10.1145/2429069.2429100

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

Emily Ruppel and Brandon Lucia. 2019. Transactional Concurrency for Intermittent Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. https://doi.org/10.1145/

3314221.3314583

Gerhard Schellhorn, Gidon Ernst, Jörg Pfähler, Dominik Haneberg, and Wolfgang Reif. 2014. Development of a Verified Flash File System. In *Proceedings of the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - Volume 8477 (ABZ 2014)*. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-43652-3_2

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

Milijana Surbatovich, Limin Jia, and Brandon Lucia. 2019. I/O Dependent Idempotence Bugs in Intermittent Systems. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 183 (Oct. 2019). https://doi.org/10.1145/3360609

TI Inc. 2020a. Overview for MSP430FRxx FRAM. https://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview.html. (2020). Visited October 14th, 2020.

TI Inc. 2020b. TI-RTOS: Real-Time Operating System (RTOS) for Microcontrollers (MCU). (2020). https://www.ti.com/tool/TI-RTOS-MCU Visited October 14th, 2020.

Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. https://doi.org/10.1145/1950365.1950379

Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation without Hardware Support or Programmer Intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, Savannah, GA. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/vanderwoude

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

Zac Manchester. 2015. KickSat. http://zacinaction.github.io/kicksat/. (2015).

Hong Zhang, Mastooreh Salajegheh, Kevin Fu, and Jacob Sorber. 2011. Ekho: Bridging the Gap Between Simulation and Reality in Tiny Energy-harvesting Sensors. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems (HotPower '11)*. Article 9. https://doi.org/10.1145/2039252.2039261