Intermittent Computing with Peripherals, Formally Verified

Gautier Berthou
Univ Lyon, INSA-Lyon, Inria, CITI

Pierre-Évariste Dagand
Sorbonne Univ, CNRS, Inria, LIP6

Delphine Demange
Univ Rennes, Inria, CNRS, IRISA

Rémi Oudin
Sorbonne Univ, CNRS, Inria, LIP6

Tanguy Risset
Univ Lyon, INSA-Lyon, Inria, CITI

Abstract

Transiently-powered systems featuring non-volatile memory as well as external peripherals enable the development of new low-power sensor applications. However, as programmers, we are ill-equipped to reason about systems where power failures are the norm rather than the exception. A first challenge consists in being able to capture *all* the volatile state of the application – external peripherals included – to ensure progress. A second, more fundamental, challenge consists in specifying how power failures may interact with peripheral operations. In this paper, we propose a formal specification of intermittent computing with peripherals, an axiomatic model of interrupt-based checkpointing as well as its proof of correctness, machine-checked in the Coq proof assistant. We also illustrate our model with several systems proposed in the literature.

CCS Concepts: • **Computer systems organization** → *Embedded systems*; • **General and reference** → *Reliability*; • **Software and its engineering** → *Checkpoint / restart*; • **Theory of computation** → *Program specifications*; *Program semantics.*

Keywords: Intermittent computing; Formal specification; Semantic refinement

ACM Reference Format:
Gautier Berthou, Pierre-Évariste Dagand, Delphine Demange, Rémi Oudin, and Tanguy Risset. 2020. Intermittent Computing with Peripherals, Formally Verified. In *Proceedings of the 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '20), June 16, 2020, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3372799.3394365

1 Introduction

Transiently-powered systems are tiny, battery-less devices that harvest energy from their environment. The energy thus retrieved flows into short-term storage facilities, such as capacitors, leading to computation times on the order of thousands of cycles per run. A *run* denotes a continuous period of time without power failure. To ensure progress of computation across runs, system integrators often pair such devices with non-volatile memory (NVM), such as non-volatile RAM technology (FRAM, MRAM, *etc.*). This combination of features gave birth to *intermittent computing*. However, the interaction of volatile (registers, caches) and non-volatile states together with unpredictable power failures is itself a poisonous mix, dubbed the "broken time machine" [30].

One way to circumvent this issue consists in adding a sensor monitoring the remaining energy level [15]. Before the power runs out, a hardware interrupt is triggered, which is then handled in software, leading the system to checkpoint its volatile state to NVM. Checkpoints are necessarily consistent as they result from a snapshot of the application taken at a single point in time. This offers the simplicity of *static checkpointing* [25] – where explicit checkpointing instructions are spread throughout the code – without much overhead – snapshots are acquired only when necessary.

This solution leaves a critical blind spot: external peripheral devices also contain volatile state, inaccessible from the CPU – efficiently or at all. Besides, interacting with an external device changes our expectations about the system. Consider for instance a radio transceiver peripheral. The radio device requires calibration before packet emission or reception. This takes about 100μs on the device, during which the driver is busy-waiting. If a power-loss were to occur after, say, 75μs, it would be functionally incorrect to resume the transceiver in calibration mode and only wait for the remaining 25μs. To be correct, the calibration code sequence must execute within a single run.

Following earlier work on supporting peripherals in intermittent computation [2, 6, 9, 17, 29], we identify two key challenges:

(C1) Peripherals add volatile *and* opaque state to the overall system;
(C2) Peripherals have a concrete, observable impact on the environment of the system.

The present work aims at providing a conceptual framework for (1) formally expressing these two requirements; and (2) proving that a general interrupt-based checkpointing

LCTES '20, June 16, 2020, London, United Kingdom

scheme meets its specification. To this end, we make the following assumptions throughout the paper:

(A1) NVM is *solely* used to store snapshots of the application. Conversely, application code cannot access the NVM;

(A2) Checkpointing volatile state (registers, RAM, *etc.*) from the micro-controller (MCU) is a solved problem (*e.g.,* Ahmed et al. [1]) whereas the peripheral internal state is completely opaque to the application;

(A3) Peripherals act upon an environment that is idempotent. A transiently-powered system may, for example, send a network packet multiple times: we expect the network protocol to gracefully handle such situations. This touches upon a fundamental assumption of transiently-powered systems in general [35];

(A4) Liveness of the application is secured by a suitably calibrated power sensor. Checkpointing needs not *always* succeed but it is assumed to *eventually* succeed.

Assumption (A1) excludes some existing systems [18, 20, 25, 27, 29, 37] from the scope of this work. Since Challenges (C1) and (C2) are orthogonal to an application's ability to access NVM, we chose to exclude it for pedagogical clarity.

We propose a general model of interrupt-based checkpointing based on these assumptions. Dealing with an external communication bus (*e.g.,* I^2C or SPI), we may for example suffer from a power failure right after having configured the bus to address a specific component but before actually interacting with the component. In the next run, the bus will be resumed in its default state: if we resume the application where it lost power, it will fail to proceed as desired. Instead, one resolves to *log the interaction* with the peripherals and replay the log upon reboot [2, 6, 9]. Some operations must execute under continuous power to produce a meaningful outcome, as witnessed by our earlier radio device example. Applications must be able to specify *power-continuous sections*[1], asserting that a given sequence of instructions must be executed within a single run. Fig. 1 (PLF) illustrates our proposal. Peripheral devices (DEV) are accessed through a specific API. NVM (CHKPT) enables double-buffering [31], ensuring progress. A logging mechanism (LOG) is key to restore peripherals in a consistent state.

Now, this begs the question: what does it mean for our scheme to be "correct"? The application is specified (SPEC in Fig. 1) as if it was run in a continuously-powered environment. Our correctness result states that the checkpointed application behaves as prescribed by SPEC: the trace of operations emitted by peripherals is also observable in the continuous-power specification (modulo re-executions), and power-continuous sections are executed within a single run.

Our contributions are the following:

- We specify intermittent computing with peripherals (Section 2) with a labeled transition system. We strive for generality, making no assumption about the actual behavior of peripherals and allowing non-determinism, including preemptive and concurrent systems.
- We give an axiomatic model of interrupt-based checkpointing (Section 3). This includes an equational specification of a logging mechanism and a persistent storage interface, thus simplifying the task of checking the validity of one's implementation to a handful of conditions. The model consists of a state machine with five states that captures the essence of checkpointing.
- We state the correctness of our model with respect to its specification (Section 4). We show that peripheral operations are preserved despite power failures and that power-continuous sections are complied with.
- We relate our model to existing systems (Section 5).

[1]Our notion of "power-continuous section" corresponds to "atomicity" from Maeng and Lucia [29], an overloaded terminology from concurrent systems.

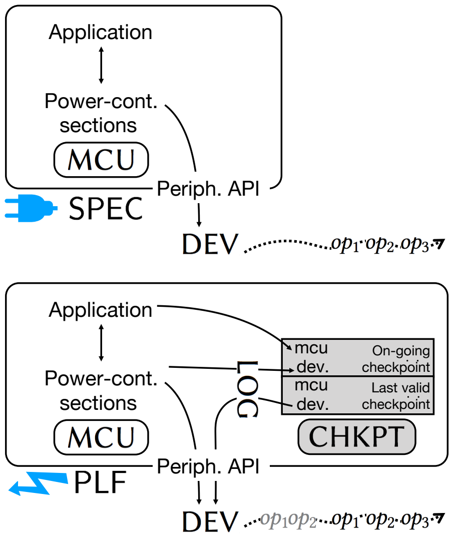

Figure 1. The checkpointing model (PLF) implements its continuous-power specification (SPEC)

Overall, the present work aims at consolidating our formal understanding of transiently-powered systems and their interaction with external peripherals. This is first and foremost a conceptual work. In particular, this paper does not provide a verification tool nor does it prove the correctness of a particular implementation: our objective is to provide system designers with a solid, actionable mental model.

All the formal definitions and results presented in this paper have been machine-checked [4] in the Coq proof assistant [36]. For readability, we have typeset our Coq definitions using set-theoretic notations. We nonetheless keep a distinct namespace per conceptual object, which follows the naming scheme NAMESPACE.object. We write NAMESPACE.t to denote abstract components whose implementation is left unspecified.

2 Intermittent Computing & Peripherals

In the following, we aim to distill the essence of intermittent computing with peripherals. We focus our attention solely on the challenges raised by the combination of power failure and peripheral devices. To this end, we introduce an axiomatic programming model. To the practitioner, it may seem far removed from the assembly code that actually drives intermittent computations. This is in fact a virtue of this work: we aim at providing a conceptual framework with which to reason about intermittent computations and their interaction with peripherals. By freeing ourselves from a particular implementation, we remain non-prescriptive about orthogonal design choices, such as the treatment of concurrency, interrupts, *etc.* We thus offer a very liberal specification that can be readily and effectively used to check the design of a concrete implementation.

In this section, we layout our *specification* of intermittent computations, which ought to be met by our checkpointing model. We encourage readers to check that the behaviors they care about in their applications can be captured by our specification.

Modeling the MCU. We specify the MCU as an overarching abstraction of the CPU registers and relevant fragments of *volatile* memory (RAM). It encompasses all the volatile states that can efficiently be checkpointed to NVM through standard techniques [3]. It does not include peripheral devices – whose treatment comes next – and non-volatile memory – which is outside the scope of our specifications, as per Assumption (A1). In the following, we let $\mathsf{MCU.t}$ be the set of possible MCU states. We use the variable $mcu \in \mathsf{MCU.t}$ to denote an arbitrary MCU state. We call $\mathsf{MCU.init} \in \mathsf{MCU.t}$ the initial MCU state, just before executing the first instruction of a given application.

Modeling peripheral devices. Handling Challenge (C1) calls for a careful distinction between the physical peripherals – whose internal state cannot be accessed by the program – and the interface it exposes to the program. We therefore introduce both an abstract description of the peripheral, representing the state of the physical device, and an interface driving the evolution of the abstract device state.

We let $\mathsf{DEV.t}$ be the set of possible physical peripheral states, and use the variable $dev \in \mathsf{DEV.t}$ to denote an arbitrary peripheral state. We call $\mathsf{DEV.init} \in \mathsf{DEV.t}$ the initial peripheral state, on power-up.

The idea that peripherals are manipulated through a well-defined interface is a natural one [9]. It may involve the low-level application binary interface (ABI) documented by a datasheet or the application program interface (API) provided by a high-level library. We accommodate either style by assuming that there exists a set $\mathsf{DEV.ops}$ of operations supported by the peripheral. Conventionally, we write $op \in \mathsf{DEV.ops}$ to denote an arbitrary operation. Given a particular device state, performing an operation has the effect of producing a new device state. The behavior of a peripheral can thus be specified through a relation

$$- \underset{\mathsf{DEV}}{\overset{-}{\rightsquigarrow}} - \subseteq \mathsf{DEV.t} \times \mathsf{DEV.ops} \times \mathsf{DEV.t}$$

where $dev \underset{\mathsf{DEV}}{\overset{op}{\rightsquigarrow}} dev'$ denotes the execution of operation op in device state dev and resulting in device state dev'.

This relation plays an essential role in our formalization. It turns $\mathsf{DEV.ops}$, over which the software has control, into operations over the physical device $\mathsf{DEV.t}$, over which the software has no control. Our simple modeling of peripheral devices accounts for systems featuring multiple physically-decoupled peripherals. We can simply consider the set of all available devices as a single one whose interface is the disjoint sum of their respective interfaces.

Specification. We now introduce, under the SPEC namespace, our axiomatic specification of an intermittent computation. We ask for just enough structure to address Challenge (C2). In doing so, we expose only the properties we care about, namely the expected observable behavior of programs under continuous power execution. The remaining implementation details, which are orthogonal to the correctness statement, are abstracted away.

To account for power-continuous code sections, we distinguish two execution modes in an intermittent computation: a program can either run in "user mode" ($\mathcal{U} \in \mathsf{SPEC.mode}$) or in "driver mode" ($\mathcal{D} \in \mathsf{SPEC.mode}$). A computation may be resumed at any point in user mode while it can only be resumed to the very first instruction of a sequence of instructions in driver mode. This means that a sequence of instructions in driver mode should either be executed entirely without being interrupted by a power failure, or the entire code block will be re-executed in the next run.

Example 2.1. The calibration code of the radio frequency synthesizer (discussed in the introduction) should therefore be specified as a driver mode code sequence. This ensures that the busy-wait is always executed as a whole before calibration is deemed completed. Other examples include non-immediate transactions, frequent with SPI or I^{2}C buses.

Modes are thus a key ingredient to specify power-continuous sections. We write $m \in \mathsf{SPEC.mode}$ to denote an arbitrary execution mode.

Our specification should be able to describe a set of desired behaviors. Since peripherals are meant to interact with their environment, a natural notion of "behavior" is the sequence of operations performed by the device. We write $\mathsf{SPEC.trace} \triangleq \mathsf{DEV.ops}^*$ the set of sequences of operations. We write $t \in \mathsf{SPEC.trace}$ to denote an arbitrary trace, ϵ to denote the empty sequence and $t\ ;\ t'$ to denote a concatenation of traces. We define $dev \underset{\mathsf{DEV}}{\overset{t}{\rightsquigarrow}}{}^{*} dev' \triangleq dev \underset{\mathsf{DEV}}{\overset{op_0}{\rightsquigarrow}} dev_0 \ldots \underset{\mathsf{DEV}}{\overset{op_n}{\rightsquigarrow}} dev'$,

$$\begin{tikzcd}
(\mathcal{U}, mcu, dev) \arrow[loop left, "\textsc{Usr}"] \arrow[r, bend left, "\textsc{Enter}"] & (\mathcal{D}, mcu, dev) \arrow[loop right, "\textsc{Drv}"] \arrow[l, bend left, "\textsc{Leave}"]
\end{tikzcd}$$

Figure 2. Specification state machine

the sequential execution of the trace $t = op_0 \ ; \ \ldots \ ; \ op_n$ starting from device state *dev*, resulting in device state *dev′*.

Intermittent computations are described axiomatically. The state of the computation

$$\mathsf{SPEC.state} \triangleq \mathsf{SPEC.mode} \times \mathsf{MCU.t} \times \mathsf{DEV.t}$$

consists of a mode, an MCU state and a device state. We write $s \in \mathsf{SPEC.state}$ to denote an arbitrary state. The execution of a program is specified through a single-step transition relation $- \underset{\mathsf{SPEC}}{\overset{-}{\rightsquigarrow}} - \subseteq \mathsf{SPEC.state} \times \mathsf{SPEC.trace} \times \mathsf{SPEC.state}$ that takes an input state to produce a (possibly empty) trace of observable events and a resulting state.

This relation is subject to the following invariants.

(Axiom-Usr): In user mode, computations do not interact with peripherals, *i.e.,* if

$$(\mathcal{U}, mcu, dev) \underset{\mathsf{SPEC}}{\overset{t}{\rightsquigarrow}} (\mathcal{U}, mcu', dev'),$$

then $dev = dev'$ and $t = \epsilon$;

(Axiom-Drv): In driver mode, the emitted trace faithfully describes the physical evolution of the device, *i.e.,* if

$$(\mathcal{D}, mcu, dev) \underset{\mathsf{SPEC}}{\overset{t}{\rightsquigarrow}} (\mathcal{D}, mcu', dev'),$$

then $dev \underset{\mathsf{DEV}}{\overset{t}{\rightsquigarrow}}{}^{*} dev'$;

(Axiom-Enter): Transitions from user to driver mode are computationally transparent, *i.e.,* if

$$(\mathcal{U}, mcu, dev) \underset{\mathsf{SPEC}}{\overset{t}{\rightsquigarrow}} (\mathcal{D}, mcu', dev'),$$

then $mcu = mcu'$, $dev = dev'$ and $t = \epsilon$;

(Axiom-Leave): Transitions from driver to user mode are computationally transparent, *i.e.,* if

$$(\mathcal{D}, mcu, dev) \underset{\mathsf{SPEC}}{\overset{t}{\rightsquigarrow}} (\mathcal{U}, mcu', dev'),$$

then $mcu = mcu'$, $dev = dev'$ and $t = \epsilon$.

This axiomatization amounts to a state machine (Fig. 2) with two states – the user and driver modes – together with four possible kinds of transitions. Specifically, Enter and Leave delineate the power-continuous sections operating on external peripherals. These two transitions are purely formal, leaving the concrete state of the application unchanged (same MCU, same device and producing an empty trace). Upon reasoning about a concrete application, we are therefore free to switch mode at any point we see fit from a logical standpoint, irrespectively of its operational behavior.

The semantics of an intermittent computation follows simply by iterating the single-step transition relation above, starting from the initial state. Formally, we define the semantics SPEC.sem as the following set of traces:

$$t \in \mathsf{SPEC.sem} \Leftrightarrow \exists s, (\mathcal{U}, \mathsf{MCU.init}, \mathsf{DEV.init}) \underset{\mathsf{SPEC}}{\overset{t}{\rightsquigarrow}}{}^{*} s$$

The set SPEC.sem consists of all the admissible behaviors of the system under study. Indeed, any trace in this set corresponds to a sequence of peripheral operations performed during a continuously-powered execution.

Being a *specification*, the above definition of SPEC.sem deserves scrutiny. In particular, it must be expressive enough to account for the properties expected by a specific, real-world application. To fit within our framework, these properties need to be expressible in terms of traces specifying expected observations, and/or user-driver transitions specifying power-continuous sections. The following examples illustrate how our specification addresses Challenge (C2).

Example 2.2. Suppose we want to (1) sense a temperature with an analog component through an Analog-to-Digital Converter, then (2) convert that value to a human-readable format to (3) be displayed on a teletype. This program can be modeled either as a single sequence encompassing all three operations executing in mode $\mathcal{D}$, or as three sequences executing in, successively, $\mathcal{D} \rightarrow \mathcal{U} \rightarrow \mathcal{D}$ modes, value conversion being performed in mode $\mathcal{U}$. In both cases, the formal trace consists in a sensing operation followed by a display operation. However, the latter case describes an application that – subject to power failures – allows for the displayed temperature to be arbitrarily outdated, whereas the former case captures the timeliness requirement of the whole sequence of operations.

Example 2.3. Consider an application that regularly sends a thermal sensor's data in radio packets. The application sets a timer to periodically sense and send data, then waits for commands coming from the radio between packet emissions. If both the sensor and the radio devices are accessed through the *same* SPI bus, this SPI bus requires two different configurations (*e.g.,* bus clock frequency), both being distinct operations in DEV.ops, to communicate with both devices.

The datasheet specifies the state of all the peripherals (SPI bus, thermal sensor, radio device, timer) upon power-up: this defines DEV.init. The radio module, when set in reception mode, may fire interrupts. Retrieving the packet should be performed in a power-continuous section ($\mathcal{D}$ mode) so that either the handler runs to completion in a single run (effectively receiving the radio packet), or the packet is lost in the event of a power-loss.

Note that the interaction between, say, timer and radio interrupts requires a resource locking mechanism to properly share access to the SPI bus, independently of whether the application may reboot or not. Such a mechanism is therefore orthogonal to power-continous sections.

3 Interrupt-based checkpointing

We now give a formal description of interrupt-based checkpointing with peripherals [2, 6, 9, 29]. The present model plays two roles. First, we lay out the minimal requirements to implement an interrupt-based checkpointing system. Namely, one must be able to log peripheral actions and store an image of the MCU and of the action log to NVM. Second, we provide a conceptual framework for reasoning about the correctness of such a checkpointing system. Namely, we introduce the notion of instrumented trace.

3.1 Operation logging

To restore a physical device into a previously encountered state, we must resort to the only information accessible to the program: peripheral operations. A straightforward way to achieve this objective would consist in logging *every* such operation. Restoring the physical device would then amount to replaying the entire log. However, such an implementation would be prohibitive both in terms of time and space, especially in an embedded system. Rather than restricting ourselves to a single inefficient implementation, we consider an abstract specification of such a log. In Section 5, we show that this interface admits several efficient implementations.

We let LOG.t to be the set of such summaries of all previous peripheral operations. An operation can be added to a given log through the function $\mathsf{LOG.log} \in \mathsf{DEV.ops} \to \mathsf{LOG.t} \to \mathsf{LOG.t}$. Restoring the log, through the function $\mathsf{LOG.restore} \in \mathsf{LOG.t} \to \mathsf{DEV.t}$, allows recreating a peripheral state *ex nihilo* by replaying the log. We denote $\mathsf{LOG.init} \in \mathsf{LOG.t}$ the initial state of the log. We write $\ell \in \mathsf{LOG.t}$ to denote an arbitrary log. We require that function LOG.restore be consistent with LOG.init and LOG.log:

(Axiom-Restore-Init) Restoring the initial log yields the initial state, *i.e.,* $\mathsf{LOG.restore}\ \mathsf{LOG.init} = \mathsf{DEV.init}$;

(Axiom-Restore-Log) Given a log that faithfully represents a given device, restoring the extension of this log with a new operation gives the same peripheral state as the one obtained by running the operation on the peripheral, *i.e.,* for all $\ell \in \mathsf{LOG.t}$, $dev, dev' \in \mathsf{DEV.t}$, $op \in \mathsf{DEV.ops}$, if $\mathsf{LOG.restore}\ \ell = dev$ and $dev \underset{\mathsf{DEV}}{\overset{op}{\rightsquigarrow}} dev'$, then $\mathsf{LOG.restore}\ (\mathsf{LOG.log}\ \ op\ \ell) = dev'$.

We overload notations and write $\mathsf{LOG.log}\ \ t\ \ell$ for the sequential logging of each individual operation of a trace $t = op_0 ; \ldots ; op_n$, *i.e.,* $\mathsf{LOG.log}\ \ op_n\ (\ldots(\mathsf{LOG.log}\ \ op_0\ \ell))$.

3.2 Non-volatile checkpoint storage

Checkpointing storage is the only component in our system that is persistent across reboots. Non-volatile checkpoints are represented by an abstract set CHKPT.t. Intuitively, a checkpoint contains two snapshots of the application, implementing double-buffering [31]: a stable one (the "last" valid one) and an on-going one (the "next" valid one, if checkpointing succeeds). A snapshot consists of an MCU image and a peripheral log. We write $chkpt \in \mathsf{CHKPT.t}$ to denote an arbitrary checkpointing storage.

We access the last snapshot through the function $\mathsf{last} \in \mathsf{CHKPT.t} \to \mathsf{MCU.t} \times \mathsf{LOG.t}$, and the next snapshot through $\mathsf{next} \in \mathsf{CHKPT.t} \to \mathsf{MCU.t} \times \mathsf{LOG.t}$. For conciseness, we also define lastMcu, lastLog, nextMcu and nextLog to directly access each individual components. The initial checkpoint $\mathsf{CHKPT.init} \in \mathsf{CHKPT.t}$ is constructed from the image of the initial MCU and the initial log, *i.e.,* it satisfies the equations $\mathsf{last}\ \mathsf{CHKPT.init} = (\mathsf{MCU.init}, \mathsf{LOG.init})$ and $\mathsf{next}\ \mathsf{CHKPT.init} = (\mathsf{MCU.init}, \mathsf{LOG.init})$.

Double-buffering requires two operations. First, one must be able to overwrite atomically the last stable snapshot with the on-going one, effectively committing the on-going snapshot. This is provided by function $\mathsf{CHKPT.set} \in \mathsf{CHKPT.t} \to \mathsf{CHKPT.t}$. Second, one must be able to overwrite atomically the on-going snapshot with the last valid one, effectively resetting the on-going snapshot to the last valid one. This is provided by $\mathsf{CHKPT.reset} \in \mathsf{CHKPT.t} \to \mathsf{CHKPT.t}$. These functions are specified through a set of equations:

$$\begin{aligned}\mathsf{last}\ (\mathsf{CHKPT.set}\ chkpt) &= \mathsf{next}\ chkpt\\ \mathsf{next}\ (\mathsf{CHKPT.set}\ chkpt) &= \mathsf{next}\ chkpt\end{aligned}$$

$$\begin{aligned}\mathsf{next}\ (\mathsf{CHKPT.reset}\ chkpt) &= \mathsf{last}\ chkpt\\ \mathsf{last}\ (\mathsf{CHKPT.reset}\ chkpt) &= \mathsf{last}\ chkpt\end{aligned}$$

Finally, one must be able to update the MCU image or the log stored in the next snapshot. This is respectively achieved by function $\mathsf{CHKPT.saveNextMcu} \in \mathsf{CHKPT.t} \to \mathsf{MCU.t} \to \mathsf{CHKPT.t}$ and function $\mathsf{CHKPT.saveNextLog} \in \mathsf{CHKPT.t} \to \mathsf{LOG.t} \to \mathsf{CHKPT.t}$, characterized by:

$$\begin{aligned}\mathsf{nextMcu}\ (\mathsf{CHKPT.saveNextMcu}\ chkpt\ mcu) &= mcu\\ \mathsf{nextLog}(\mathsf{CHKPT.saveNextMcu}\ chkpt\ mcu) &= \mathsf{nextLog}\ chkpt\\ \mathsf{last}\ (\mathsf{CHKPT.saveNextMcu}\ chkpt\ mcu) &= \mathsf{last}\ chkpt\end{aligned}$$

$$\begin{aligned}\mathsf{nextLog}\ (\mathsf{CHKPT.saveNextLog}\ chkpt\ \ell) &= \ell\\ \mathsf{nextMcu}\ (\mathsf{CHKPT.saveNextLog}\ chkpt\ \ell) &= \mathsf{nextMcu}\ chkpt\\ \mathsf{last}\ (\mathsf{CHKPT.saveNextLog}\ chkpt\ \ell) &= \mathsf{last}\ chkpt\end{aligned}$$

Elements of CHKPT.t constitute the *only* piece of state that we intend to persist across reboots.

3.3 Interrupt-based checkpointing

We now define our model of interrupt-based checkpointing under the PLF namespace, which stands for "$\underline{\text{P}}$ower-$\underline{\text{L}}$oss & checkpoint $\underline{\text{F}}$ailure". Our model is a refinement [22, 24, 26] of the specification given in Section 2. It is defined as a state machine that emits instrumented traces upon reaching specific transitions.

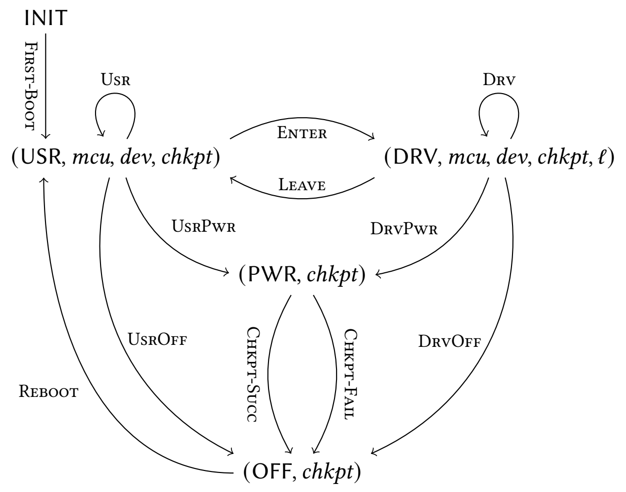

Figure 3. Checkpointing state machine

States. Our model operates over 5 kinds of states, whose union is termed PLF.state, conventionally ranged over by $\hat{s}$:

- INIT represents the initial state of the machine;
- (USR, *mcu*, *dev*, *chkpt*) represents a computation running in user mode;
- (DRV, *mcu*, *dev*, *chkpt*, ℓ) represents a computation running in driver mode, maintaining a volatile log ℓ of operations executed up to this point;
- (PWR, *chkpt*) represents a computation interrupted by a power failure in which all volatile state is lost;
- (OFF, *chkpt*) represents a computation that has been turned off.

The corresponding state machine is represented in Fig. 3. We explain below each of its transitions.

Instrumented traces. In this system, peripheral operations could be repeated either by the program – during a run – or by the run-time system – due to a power failure. In order to relate precisely the trace produced by the transiently-powered system with a trace produced by a continuously-powered execution of the system, it is crucial to distinguish (repeated) progress from failed attempts.

In the transiently-powered system, we must keep track of two kinds of information: (1) whether a power-continuous section has successfully completed or has been aborted due to a power failure; (2) whether the next snapshot could be set over the last one, or power ran out beforehand. We thus extend the notion of trace with four extra observable events:

Information 1. A power-continuous section
- completed before a power-failure interrupt: $\mathsf{Log}^{\top}$
- or was interrupted due a power failure: $\mathsf{Log}^{\perp}$

Information 2. The checkpointing
- completed before power ran out: $\mathsf{Chkpt}^{\top}$
- or could not complete before power ran out: $\mathsf{Chkpt}^{\perp}$

In this model, an instrumented trace (written $\hat{t}$) is thus a sequence of either of these events or peripheral actions:

$$\mathsf{PLF.trace} \triangleq (\{\mathsf{Chkpt}^{\top}, \mathsf{Chkpt}^{\perp}, \mathsf{Log}^{\top}, \mathsf{Log}^{\perp}\} \uplus \mathsf{DEV.ops})^{*}$$

In Section 4, we show that, from the traces produced by our checkpointing model, we can extract a meaningful subtrace of DEV.ops events that could have been produced by the specification, *i.e.,* within a single run.

State machine transitions. We model interrupt-based checkpointing with peripherals through a relational specification of the transitions of the state machine in Fig. 3. The transition relation defined in Fig. 4, and written

$$- \underset{\mathsf{PLF}}{\overset{-}{\rightsquigarrow}} - \subseteq \mathsf{PLF.state} \times \mathsf{PLF.trace} \times \mathsf{PLF.state}$$

indicates by $\hat{s} \underset{\mathsf{PLF}}{\overset{\hat{t}}{\rightsquigarrow}} \hat{s}'$ a computation taking input state $\hat{s}$ to output state $\hat{s}'$ emitting an instrumented trace $\hat{t}$. The trace semantics of the overall application is defined as the set of traces reachable from the initial state:

$$\hat{t} \in \mathsf{PLF.sem} \Leftrightarrow \exists \hat{s}, \mathsf{INIT} \underset{\mathsf{PLF}}{\overset{\hat{t}}{\rightsquigarrow}}{}^{*}\, \hat{s}$$

Transitions Usr, Drv, Enter and Leave are key to embedding the behavior of the continuous-power specification. They mirror their SPEC counterparts, in addition to retrieving (transition Enter) or storing (transition Leave) information to checkpointing storage, or to the volatile log (Drv). Crucially, transitions Enter, Drv and Leave keep the volatile log of operations in sync with the state of the physical device.

Transitions to PWR are responsible for producing a consistent checkpoint across the whole application (MCU *and* peripherals). Upon transition UsrPwr, the whole MCU is checkpointed so as to resume at the current program point. Upon transition DrvPwr, the volatile execution context is thrown away, relying on the fact that transition Enter has produced a valid checkpoint right before leaving user mode to enter the power-continuous section.

Transitions from state PWR to state OFF capture, non-deterministically, whether checkpointing succeeds or fails (transitions Chkpt-Succ and Chkpt-Fail). We also allow cases where power could run out before the power-failure interrupt is even raised. Therefore, in user mode, the state machine non-deterministically steps from state USR to either state PWR (transition UsrPwr) or state OFF (transition UsrOff). Similarly, in driver mode, the state machine non-deterministically steps from DRV to either state PWR (transition DrvPwr) or state OFF (transitions DrvOff). Either way, when the system restarts (transition Reboot), it reinstates the last MCU image and restores the peripheral's state thanks to the last stable log.

As a sanity check, we must validate our model against the physical objects it is supposed to represent. In particular, no volatile state is silently preserved across runs: *chkpt*

First-Boot

$$\frac{}{\mathsf{INIT} \overset{\epsilon}{\underset{\mathsf{PLF}}{\rightsquigarrow}} (\mathsf{USR}, \mathsf{MCU.init}, \mathsf{DEV.init}, \mathsf{CHKPT.init})}$$

Usr

$$\frac{(\mathcal{U}, mcu, dev) \overset{t}{\underset{\mathsf{SPEC}}{\rightsquigarrow}} (\mathcal{U}, mcu', dev')}{(\mathsf{USR}, mcu, dev, chkpt) \overset{t}{\underset{\mathsf{PLF}}{\rightsquigarrow}} (\mathsf{USR}, mcu', dev', chkpt)}$$

Drv

$$\frac{(\mathcal{D}, mcu, dev) \overset{t}{\underset{\mathsf{SPEC}}{\rightsquigarrow}} (\mathcal{D}, mcu', dev') \qquad \ell' = \mathsf{LOG.log}\ t\ \ell}{(\mathsf{DRV}, mcu, dev, chkpt, \ell) \overset{t}{\underset{\mathsf{PLF}}{\rightsquigarrow}} (\mathsf{DRV}, mcu', dev', chkpt, \ell')}$$

Enter

$$\frac{(\mathcal{U}, mcu, dev) \overset{t}{\underset{\mathsf{SPEC}}{\rightsquigarrow}} (\mathcal{D}, mcu', dev') \qquad chkpt' = \mathsf{CHKPT.saveNextMcu}\ chkpt\ mcu \qquad \ell = \mathsf{nextLog}\ chkpt}{(\mathsf{USR}, mcu, dev, chkpt) \overset{t}{\underset{\mathsf{PLF}}{\rightsquigarrow}} (\mathsf{DRV}, mcu', dev', chkpt', \ell)}$$

Leave

$$\frac{(\mathcal{D}, mcu, dev) \overset{t}{\underset{\mathsf{SPEC}}{\rightsquigarrow}} (\mathcal{U}, mcu', dev') \qquad chkpt' = \mathsf{CHKPT.saveNextLog}\ chkpt\ \ell}{(\mathsf{DRV}, mcu, dev, chkpt, \ell) \overset{t;\mathsf{Log}^{\top}}{\underset{\mathsf{PLF}}{\rightsquigarrow}} (\mathsf{USR}, mcu', dev', chkpt')}$$

UsrPwr

$$\frac{chkpt' = \mathsf{CHKPT.saveNextMcu}\ chkpt\ mcu}{(\mathsf{USR}, mcu, dev, chkpt) \overset{\epsilon}{\underset{\mathsf{PLF}}{\rightsquigarrow}} (\mathsf{PWR}, chkpt')}$$

DrvPwr

$$\frac{}{(\mathsf{DRV}, mcu, dev, chkpt, \ell) \overset{\mathsf{Log}^{\perp}}{\underset{\mathsf{PLF}}{\rightsquigarrow}} (\mathsf{PWR}, chkpt)}$$

Chkpt-Succ

$$\frac{chkpt' = \mathsf{CHKPT.set}\ chkpt}{(\mathsf{PWR}, chkpt) \overset{\mathsf{Chkpt}^{\top}}{\underset{\mathsf{PLF}}{\rightsquigarrow}} (\mathsf{OFF}, chkpt')}$$

Chkpt-Fail

$$\frac{}{(\mathsf{PWR}, chkpt) \overset{\mathsf{Chkpt}^{\perp}}{\underset{\mathsf{PLF}}{\rightsquigarrow}} (\mathsf{OFF}, chkpt)}$$

Reboot

$$\frac{mcu' = \mathsf{lastMcu}\ chkpt \qquad dev' = \mathsf{LOG.restore}(\mathsf{lastLog}\ chkpt) \qquad chkpt' = \mathsf{CHKPT.reset}\ chkpt}{(\mathsf{OFF}, chkpt) \overset{\epsilon}{\underset{\mathsf{PLF}}{\rightsquigarrow}} (\mathsf{USR}, mcu', dev', chkpt')}$$

UsrOff

$$\frac{}{(\mathsf{USR}, mcu, dev, chkpt) \overset{\mathsf{Chkpt}^{\perp}}{\underset{\mathsf{PLF}}{\rightsquigarrow}} (\mathsf{OFF}, chkpt)}$$

DrvOff

$$\frac{}{(\mathsf{DRV}, mcu, dev, chkpt, \ell) \overset{\mathsf{Chkpt}^{\perp}}{\underset{\mathsf{PLF}}{\rightsquigarrow}} (\mathsf{OFF}, chkpt)}$$

Figure 4. Interrupt-based checkpointing model $- \overset{-}{\underset{\mathsf{PLF}}{\rightsquigarrow}} -$

is the only piece of data that goes through OFF. Further, the physical device state, DEV.t, cannot magically be captured in non-volatile memory: only MCU.t and LOG.t can be used to produce CHKPT.t and none of these can contain a DEV.t. Our model faithfully accounts for checkpointing failure: Chkpt-Fail keeps the checkpointing storage intact. Our model supports power-failure interruptions in both user and driver mode. It also models the eventuality that the power-failure interruption itself cannot complete before power goes out, through UsrOff and DrvOff. Finally, traces produced by the transition system merely *record* the execution, *i.e.,* they play no role in the execution.

4 Correctness

Existing work only provides informal descriptions of what checkpointing is supposed to achieve, even in the simple setting of intermittent peripheral operations [2, 6, 9, 29]. Here, we relate our specification of intermittent computing with peripherals (Section 2) with our model of interrupt-based checkpointing (Section 3).

A benefit of this conceptual work is to lay out the key invariants relied upon by existing systems. Indeed, our work is unencumbered by the particulars of producing an MCU image (abstracted once and for all by MCU.t), or of logging peripheral operations (axiomatized once and for all by LOG.t), or even in the minutiae of transfering data to and from volatile and non-volatile state (axiomatized once and for all by CHKPT.t). By bringing conceptual clarity, we hope to provide a blueprint for future system designers.

Stuttering. Our correctness result consists in a semantic refinement [24, 26] between the model and the specification. We must prove that computation steps $\hat{s} \overset{\hat{t}}{\underset{\mathsf{PLF}}{\rightsquigarrow}} \hat{s}'$ in the model correspond to computation steps in the specification $s \overset{t}{\underset{\mathsf{SPEC}}{\rightsquigarrow}} s'$ – which executes in a continuously-powered environment.

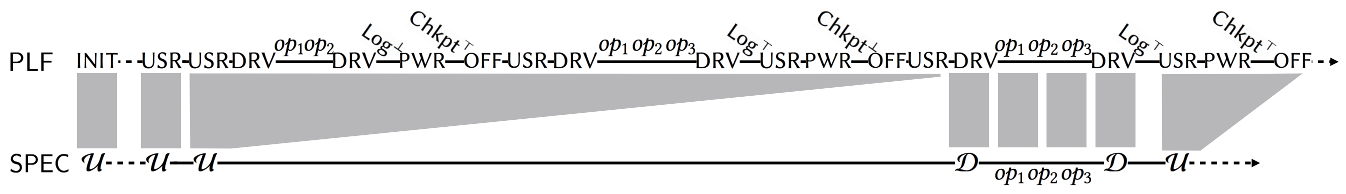

Figure 5. Matching $-\underset{\text{SPEC}}{\bar{\rightsquigarrow}}-$ with $-\underset{\text{PLF}}{\bar{\rightsquigarrow}}-$ executions. Grey areas mark the correspondance between execution states.

Because of power failures, the model may re-execute some computation steps. In this case, the specification can simply be put on hold, waiting for the model to make actual progress. In technical terms, SPEC *stutters* [22], *i.e.,* it silently takes no step while the model performs wasted work.

Fig. 5 illustrates how we match, in our refinement proof, a given execution in PLF (top) with one in SPEC (bottom). As long as PLF is going to fail to complete the next checkpoint, the specification has to stutter. As soon as the next checkpoint is guaranteed to succeed, the execution in SPEC should be able to follow, in lockstep, the execution in PLF. In our proof, we handle non-determinism in PLF by making the simulation relation depend on the future of the current PLF execution state.

Stuttering in SPEC must be handled with care. Indeed, we must make sure that it is not constantly stuttering: that would make our correctness theorem vacuously true. We therefore design a subtrace relation that precludes unwanted stuttering.

Subtrace relation. The cornerstone of our correctness result is a relation $- \succcurlyeq - \in$ PLF.trace $\times$ SPEC.trace between instrumented traces and specification traces. Due to the potential re-execution of operations, only a subset of the DEV.ops event emitted by PLF might be emitted by SPEC, as it is exemplified on Fig. 5. Relation $- \succcurlyeq -$ thus needs to enforce a *subtrace* relation. Besides, we want to ensure that sequences of operations delineated by Enter/Leave are eventually executed within a single run.

Intuitively, given a trace $\hat{t} \in$ PLF.trace, we first filter all subtraces where the checkpointing succeeds, leading a trace containing events in $\{\mathsf{Log}^\top, \mathsf{Log}^\perp\} \uplus$ DEV.ops, conventionally written as $\check{t}$. We write $\check{e}$ for any such observable event. Then, *within* each of these subtraces, we select the events of power-continuous sections that completed without a power failure, thereby obtaining a trace $t \in$ SPEC.trace.

We hence define the subtrace relation $- \succcurlyeq -$ as a composition of two subtrace relations (defined in Fig. 6):

$$\hat{t} \succcurlyeq t \Leftrightarrow \exists \check{t}, \hat{t} \succcurlyeq^{\mathsf{Chkpt}} \check{t} \wedge \check{t} \succcurlyeq^{\mathsf{Log}} t$$

Relation $- \succcurlyeq^{\mathsf{Chkpt}} -$ deals with re-execution of code upon a checkpointing failure (transition Chkpt-Fail), filtering out sub-sequences of events that end with $\mathsf{Chkpt}^\perp$, while retaining any other event leading to a $\mathsf{Chkpt}^\top$. Similarly, relation $- \succcurlyeq^{\mathsf{Log}} -$ deals with re-execution of code upon a

$$\hat{t} \succcurlyeq^{\mathsf{Chkpt}} \check{t} \Leftrightarrow \hat{t} \succcurlyeq_\top^{\mathsf{Chkpt}} \check{t} \vee \hat{t} \succcurlyeq_\perp^{\mathsf{Chkpt}} \check{t}$$

$$\frac{\hat{t} \succcurlyeq^{\mathsf{Chkpt}} \check{t}}{\mathsf{Chkpt}^\top ; \hat{t} \succcurlyeq_\top^{\mathsf{Chkpt}} \check{t}} \qquad \frac{\hat{t} \succcurlyeq_\top^{\mathsf{Chkpt}} \check{t}}{\check{e} ; \hat{t} \succcurlyeq_\top^{\mathsf{Chkpt}} \check{e} ; \check{t}}$$

$$\frac{}{\epsilon \succcurlyeq_\perp^{\mathsf{Chkpt}} \epsilon} \qquad \frac{\hat{t} \succcurlyeq_\perp^{\mathsf{Chkpt}} \check{t}}{\check{e} ; \hat{t} \succcurlyeq_\perp^{\mathsf{Chkpt}} \check{t}} \qquad \frac{\hat{t} \succcurlyeq^{\mathsf{Chkpt}} \check{t}}{\mathsf{Chkpt}^\perp ; \hat{t} \succcurlyeq_\perp^{\mathsf{Chkpt}} \check{t}}$$

$$\check{t} \succcurlyeq^{\mathsf{Log}} t \Leftrightarrow \check{t} \succcurlyeq_\top^{\mathsf{Log}} t \vee \check{t} \succcurlyeq_\perp^{\mathsf{Log}} t$$

$$\frac{\check{t} \succcurlyeq^{\mathsf{Log}} t}{\mathsf{Log}^\top ; \check{t} \succcurlyeq_\top^{\mathsf{Log}} t} \qquad \frac{\check{t} \succcurlyeq_\top^{\mathsf{Log}} t}{op ; \check{t} \succcurlyeq_\top^{\mathsf{Log}} op ; t}$$

$$\frac{}{\epsilon \succcurlyeq_\perp^{\mathsf{Log}} \epsilon} \qquad \frac{\check{t} \succcurlyeq_\perp^{\mathsf{Log}} t}{op ; \check{t} \succcurlyeq_\perp^{\mathsf{Log}} t} \qquad \frac{\check{t} \succcurlyeq^{\mathsf{Log}} t}{\mathsf{Log}^\perp ; \check{t} \succcurlyeq_\perp^{\mathsf{Log}} t}$$

Figure 6. Subtrace relations $- \succcurlyeq^{\mathsf{Chkpt}} -$ and $- \succcurlyeq^{\mathsf{Log}} -$

power-failure interrupt in driver mode (transition DrvPwr), filtering out sub-sequences of operations ending with a $\mathsf{Log}^\perp$, while retaining the remaining operations.

Example 4.1. In Fig. 5, the trace emitted by SPEC.sem is $t = op_1; op_2; op_3$ and the trace emitted by PLF.sem is

$$\begin{aligned}\hat{t} = \;& op_1; op_2; \mathsf{Log}^\perp; \mathsf{Chkpt}^\top; \\ & op_1; op_2; op_3; \mathsf{Log}^\top; \mathsf{Chkpt}^\perp; \\ & op_1; op_2; op_3; \mathsf{Log}^\top; \mathsf{Chkpt}^\top\end{aligned}$$

We have $\hat{t} \succcurlyeq t$ thanks to the intermediate subtrace

$$\check{t} = op_1; op_2; \mathsf{Log}^\perp; op_1; op_2; op_3; \mathsf{Log}^\top$$

Indeed, we have $\hat{t} \succcurlyeq^{\mathsf{Chkpt}} \check{t}$ by filtering out from $\hat{t}$ the checkpointing failure subtrace $op_1; op_2; op_3; \mathsf{Log}^\top; \mathsf{Chkpt}^\perp$. And $\check{t} \succcurlyeq^{\mathsf{Log}} t$ by filtering out from $\check{t}$ the first aborted power-continuous section $op_1; op_2; \mathsf{Log}^\perp$.

Example 4.2. We illustrate the precision of the $- \succcurlyeq -$ relation. For the trace $\hat{t} = op_1; op_2; op_3; \mathsf{Log}^\top; \mathsf{Chkpt}^\top$, only one trace $t \in$ SPEC.trace satsifies $\hat{t} \succcurlyeq t$, and it is $t = op_1; op_2; op_3$. Only strict prefixes of $\hat{t}$ would have ϵ as a $\succcurlyeq$-subtrace.

Correctness theorem. Our correctness theorem takes the form of a trace-refinement between the checkpointing model and the continuous-power specification:

Theorem 4.3 (Correctness)**.** *For any trace* $\hat{t} \in \mathsf{PLF.sem}$, *there exists a trace* t, *such that* $t \in \mathsf{SPEC.sem}$ *and* $\hat{t} \succcurlyeq t$.

Informally, this statement reads as: "any behavior exhibited by the model (*i.e.,* $\hat{t} \in \mathsf{PLF.sem}$) could have been observed with the specification (*i.e.,* $t \in \mathsf{SPEC.sem}$), ignoring the operations that had to be re-executed (*i.e.,* $\hat{t} \succcurlyeq t$)". The fact that power-continuous sections are indeed preserved by our model follows directly from the definition of $- \succcurlyeq -$.

Note that Theorem 4.3 holds for any possible trace in the non-deterministic semantics of PLF. As long as the model does not progress, due to either an aborted power-continuous section or a checkpointing failure, the specification will stutter, emitting the empty trace ϵ. However, our precise subtrace relation ensures, by its very definition, that any observable progress in PLF.sem *is indeed* reflected in SPEC.sem.

One difficulty in the proof of Theorem 4.3 is to decide whether SPEC needs to stutter or to make progress. It requires to know whether the next power failure will lead PLF to re-executing that portion of the execution. To handle this difficulty, we introduce an auxiliary *oracle-semantics*: non-determinism is replaced by an oracle determining how the state machine behaves. We prove the existence of such an oracle for any possible execution in PLF.sem, and prove that the specification can simulate any possible oracle. Further details of the proof can be found in our Coq development [4].

5 Applications

We now relate our model to propositions from the literature, illustrating how they fit within our conceptual framework.

Restop [2]. Restop is a middleware supporting peripherals in a transiently-powered context. It specifically targets devices connected through SPI and I^2C buses. The authors define the notion of a "peripheral instruction" ([2, §3]), akin to our set of operations (DEV.ops), as the "information required by the system to issue the operation on the peripheral".

At run-time, peripheral instructions are stored in an "instruction history table", which effectively implements our LOG.t interface as a sequence of instructions to be replayed on reboot. To keep the size of this log to a minimum, instructions are equipped with a choice of 4 semantics ([2, §3.1]): "not-save", "save", "save-but-replace", "preserve". The choice of a particular semantics reflects the effect of the instruction on the peripheral (*e.g.,* save-but-replace expects the operation to overwrite the effect of a previous instance).

The Restop designers do not commit to a specific checkpointing scheme ("the history table can either be: (1) placed in main memory; or (2) directly located in NVM" [2, §3]). Similarly, our model supports both: our LOG.t interface leaves unspecified whether the log is physically maintained in volatile or non-volatile memory.

Sytare [6]. Sytare is a library operating system targeting a wide range of peripherals in a transiently-powered context. Interactions with the peripherals are mediated by a system call interface ("syscall") delineating sequences of instructions allowed to interact with peripherals ("drivers", running in "kernel mode") from application code (running in "user mode"), which may only access volatile memory. The syscall interface enables the OS implementers to maintain a distinct "kernel stack" when interacting with peripherals. As a result, the system can always backtrack to the entry point of a syscall by throwing away this stack. This allows them to easily replay a syscall as a single unit of code, thus providing a power-continous section mechanism.

Sytare supports any peripheral as long as it registers an API through the syscall mechanism and exposes a "device context". A device context is a C data-structure (a `struct`) that records device-specific data necessary to reconfigure it upon reboot. Each operation exposed to the syscall interface updates the device context accordingly. The collection of peripherals thus defines an array of device contexts, which all together amount to our LOG.t interface. As witnessed by Fig. 4 of Berthou et al. [6], Sytare naturally fits our model.

Karma [9]. Karma provides a run-time system to ensure consistency of peripheral state across reboots. Karma provides a mechanism to implement power-continuous sections, dubbed "atomicity" there, stating that "two options exist to integrate Karma with such a system support: i) changing the conditions that make checkpoints take place in a way to prevent executions lacking the required atomicity, or ii) rolling back executions to recover the non-atomic cases" [9, §C.2]. No assumption is made on how checkpoints are triggered (statically or dynamically). Karma proposes, in addition, an integrated task scheduler for user tasks and peripheral state updates. A peripheral state is represented by a state machine and a queue of operations that corresponds to our DEV.ops operations. After a power failure, peripheral state is restored by replaying the operations stored in the queue. To ensure atomicity of peripheral updates, Karma proposes, as Sytare does, a wrapper for peripheral driver functions. But instead of storing a device context, a procedure rolls back the code, which effectively implements power-continuous sections.

Unlike Sytare, Karma allows several tasks to access a peripheral. This is achieved through a dedicated task management system, which amounts to a specific "user mode" program, running in $\mathcal{U}$ mode in our model. From our formal standpoint, the difference between Sytare and Karma thus lies solely in how they instantiate (*i.e.,* implement) the transition relation of the *specification abstract machine*: the former only allows the execution of a single `main` task whereas the latter supports several concurrent tasks. This difference is orthogonal to power failures.

6 Related Work

Failure-atomicity run-times. In the general case, *i.e.,* distributed systems with volatile and non-volatile RAM, ensuring correctness and consistency of execution is a complex problem because volatile and non-volatile memories are out of sync after a crash [34]. New programming models are proposed using locks [8, 11] or extended transactional memory [14]. These legacy software systems provide minimally invasive change to programming models and environments, hence they propose concurrency-like abstractions. These concepts have been used in the context of low-power transiently-powered sensors [25, 32] that use NVM both for checkpointing and for regular program storage. Currently, our specification SPEC does not account for NVM state, hence we cannot model applications that assume that part of the program memory, *e.g.,* a stack, is in NVM.

Static and Interrupt-based checkpointing. Following the development of NVM and harvesting technologies [15, 17, 23, 33], numerous systems have been proposed to allow low-power sensors to be deployed in transiently-powered environments. Among the most well-known are chronologically: Mementos [31], QuickRecall [20], Hibernus [3], Dino [25], Ratchet [37], HarvOS [7], Clank [18], Alpaca [27] and Coati [32]. Most of these systems used checkpointing. The insertion of checkpoints can be static [7, 25], dynamic [28, 37] or, when power failures can be detected by voltage drop, interrupt-based [3, 6, 20]. The IBIS tool suite [35] is able to detect, statically or dynamically, memory inconsistencies that may occur in these applications without relying on Assumption (A1) (*i.e.,* the MCU may directly manipulate NVM).

Handling peripherals. Peripherals are not handled by classical checkpointing techniques. Peripherals are very important in embedded computing and there is also, as for memory, a state consistency problem when power is lost [25, 29]. Few systems have proposed a complete mechanism to ensure peripheral device restoration.

Sytare [5, 6] proposed a solution to recover the state of both the processor and the peripherals. Restop [2] solves a similar problem with a middleware library for off-chip peripherals accessed by SPI or I^{2}C. Karma [9] proposes another implementation. These three approaches are captured by our checkpointing model and can benefit from the modeling proposed here to verify their correctness.

Samoyed [29] proposes, under some restrictions (no interrupt allowed, stateless peripherals) to ensure peripheral state consistency by introducing user-controlled atomicity. This work makes use of NVM in application code, and hence is not captured by our current model.

Formal models, correctness proofs. Chen et al. [13] discuss some approaches for specifying and certifying crash-safety for a persistent file system. The FSCQ system they later verified [12] uses a Hoare-logic style for specifying the system: crash conditions specify the disk state right before a crash, and a recovery procedure ensures the absence of data loss. A first difference between this line of work and ours is that we consider crash-safety in systems handling peripherals. A second difference is in the general specification methodology. We adopt the so-called DSL approach [13]: we phrase the specification and the model in terms of state machines and prove a refinement between the two. Chajed et al. [10] also derive a refinement result from the Hoare-logic style specification. As they do not handle peripherals, their refinement is a simple trace inclusion. In contrast, to derive a useful and precise correctness result, we must resort to a subtrace relation.

Koskinen and Yang [21] also employ a DSL approach. Interestingly, their notion of recoverability is expressed in terms of an un-crashed program: after a crash, the program will eventually reach a state that simulates another state that an un-crashed program already reached (*i.e.,* in the trace prefix). In our model, instrumented traces and the subtrace relation help us state simply to which prefix it corresponds.

Another line of work focuses on proving the linearizability of fine-grained concurrent data-structures subject to whole-system crash [16, 19]. *Durable linearizability* requires that upon a crash, only completed operations are guaranteed to remain visible. *Buffered durable linearizability* expresses that the state after the crash must be consistent, but not necessarily up-to-date. Our subtrace refinement result is similar in spirit to a buffered durable linearization property.

7 Conclusion

We proposed a specification of intermittent computing with peripherals, together with a model of interrupt-based checkpointing that ensures the consistency of the whole system, *i.e.,* including peripherals, after reboot. Our model contains the minimal conditions that an implementation of checkpointing should satisfy to handle correctly peripherals. We formally proved the correctness of our model: behaviors of intermittent executions are as prescribed by a continuously-powered specification, modulo the necessary replays of certain peripheral operations, due to reboots. Finally, we showed that our model captures three proposals satisfying our working assumptions (A1-4): Restop [2], Sytare [6] and Karma [9].

Throughout this work, we have assumed that the application code does not interact with non-volatile memory (A1). We are currently working on extending our specification and our checkpointing model to account for NVM application state. This would allow us to model more systems from the literature [18, 25, 27, 29, 37].

Acknowledgments

We would like to thank the anonymous reviewers for their thoughtful comments. This work was supported by Inria (IPL ZEP) and the Émergence(s) program of the City of Paris.

References

[1] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2019. Efficient Intermittent Computing with Differential Checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Phoenix, AZ, USA) *(LCTES 2019)*. Association for Computing Machinery, New York, NY, USA, 70–81. https://doi.org/10.1145/3316482.3326357

[2] Alberto Rodriguez Arreola, Domenico Balsamo, Geoff V. Merrett, and Alex S. Weddell. 2018. RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems. *Sensors* 18, 1 (2018), 172. https://doi.org/10.3390/s18010172

[3] Domenico Balsamo, Alex S. Weddell, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli, and Luca Benini. 2015. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *Embedded Systems Letters* 7, 1 (2015), 15–18. https://doi.org/10.1109/LES.2014.2371494

[4] Gautier Berthou, Pierre-Evariste Dagand, Delphine Demange, Rémi Oudin, and Tanguy Risset. 2020. Intermittent Computing with Peripherals, Formally Verified – Companion Coq Development. https://gabertho.gitlabpages.inria.fr/icp-model/

[5] Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. 2017. Peripheral state persistence for transiently-powered systems. In *Global Internet of Things Summit (GIoTS)*. IEEE, New York, NY, USA, 1–6. https://doi.org/10.1109/GIOTS.2017.8016243

[6] Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. 2019. Sytare: A Lightweight Kernel for NVRAM-Based Transiently-Powered Systems. *IEEE Trans. Comput.* 68, 9 (Sep. 2019), 1390–1403.

[7] Naveed Anwar Bhatti and Luca Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-Powered Embedded Sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks* (Pittsburgh, Pennsylvania) *(IPSN '17)*. Association for Computing Machinery, New York, NY, USA, 209–219.

[8] Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence Programming Models for Non-Volatile Memory. *SIGPLAN Not.* 51, 11 (June 2016), 55–67. https://doi.org/10.1145/3241624.2926704

[9] Adriano Branco, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. 2019. Intermittent Asynchronous Peripheral Operations. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems* (New York, New York) *(SenSys '19)*. Association for Computing Machinery, New York, NY, USA, 55–67.

[10] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying Concurrent, Crash-Safe Systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 243–258. https://doi.org/10.1145/3341301.3359632

[11] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. *SIGPLAN Not.* 49, 10 (Oct. 2014), 433–452.

[12] 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* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402

[13] Haogang Chen, Daniel Ziegler, Adam Chlipala, M. Frans Kaashoek, Eddie Kohler, and Nickolai Zeldovich. 2015. Specifying Crash Safety for Storage Systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Switzerland) *(HOTOS'15)*. USENIX Association, USA, 21.

[14] 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. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 105–118.

[15] Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 767–781.

[16] John Derrick, Simon Doherty, Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. 2019. Verifying Correctness of Persistent Concurrent Data Structures. In *Formal Methods – The Next 30 Years*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer International Publishing, Cham, Switzerland, 179–195.

[17] 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* (Delft, Netherlands) *(SenSys '17)*. ACM, New York, NY, USA, Article 17, 13 pages.

[18] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 228–240.

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

[20] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. 2015. QuickRecall: A HW/SW Approach for Computing across Power Cycles in Transiently Powered Computers. *JETC* 12, 1 (2015), 8:1–8:19. https://doi.org/10.1145/2700249

[21] 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* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/2837614.2837648

[22] Leslie Lamport. 2008. Computation and state machines.

[23] Yoonmyung Lee, Suyoung Bang, Inhee Lee, Yejoong Kim, Gyouho Kim, Mohammad Hassan Ghaed, Pat Pannuto, Prabal Dutta, Dennis Sylvester, and David Blaauw. 2013. A Modular 1 mm^3 Die-Stacked Sensing Platform With Low Power I^{2}C Inter-Die Communication and Multi-Modal Energy Harvesting. *IEEE Journal of Solid-State Circuits* 48 (2013), 229–243.

[24] Xavier Leroy. 2009. A Formally Verified Compiler Back-End. *J. Autom. Reason.* 43, 4 (Dec. 2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[25] Brandon Lucia and Benjamin Ransford. 2015. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. ACM, Portland, OR, USA, 575–585.

[26] Nancy Lynch and Frits Vaandrager. 1996. Forward and Backward Simulations. *Inf. Comput.* 128, 1 (July 1996), 1–25. https://doi.org/10.1006/inco.1996.0060

[27] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: intermittent execution without checkpoints. *PACMPL* 1, OOPSLA (2017), 96:1–96:30. https://doi.org/10.1145/3133920

[28] Kiwan Maeng and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, Carlsbad, CA, 129–144.

[29] Kiwan Maeng and Brandon Lucia. 2019. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design*

and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. ACM, New York, NY, USA, 1101–1116.

[30] Benjamin Ransford and Brandon Lucia. 2014. Nonvolatile Memory is a Broken Time Machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness* (Edinburgh, United Kingdom) *(MSPC '14)*. Association for Computing Machinery, New York, NY, USA, Article 5, 3 pages. https://doi.org/10.1145/2618128.2618136

[31] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: system support for long-running computation on RFID-scale devices. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Newport Beach, California, USA). ACM, New York, NY, USA, 159–170.

[32] Emily Ruppel and Brandon Lucia. 2019. Transactional concurrency control for intermittent, energy-harvesting computing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, New York, NY, USA, 1085–1100. https://doi.org/10.1145/3314221.3314583

[33] Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. 2008. Design of an RFID-Based Battery-Free Programmable Sensing Platform. *IEEE Transactions on Instrumentation and Measurement* 57, 11 (2008), 2608–2615.

[34] Thomas Shull, Jian Huang, and Josep Torrellas. 2018. Defining a high-level programming model for emerging NVRAM technologies. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang 2018, Linz, Austria, September 12-14, 2018*, Eli Tilevich and Hanspeter Mössenböck (Eds.). ACM, New York, NY, USA, 11:1–11:7. https://doi.org/10.1145/3237009.3237027

[35] Milijana Surbatovich, Limin Jia, and Brandon Lucia. 2019. I/O dependent idempotence bugs in intermittent systems. *PACMPL* 3, OOPSLA (2019), 183:1–183:31. https://doi.org/10.1145/3360609

[36] The Coq Development Team. 2018. The Coq Proof Assistant, version 8.8.0. https://doi.org/10.5281/zenodo.1219885

[37] 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 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, Savannah, GA, 17–32.