A Simpler, Safer Programming and Execution Model for Intermittent Systems

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

Benjamin Ransford
University of Washington, USA
ransford@cs.washington.edu

Abstract

Energy harvesting enables novel devices and applications without batteries, but intermittent operation under energy harvesting poses new challenges to memory consistency that threaten to leave applications in failed states not reachable in continuous execution. This paper presents analytical models that aid in reasoning about intermittence. Using these, we develop DINO (Death Is Not an Option), a programming and execution model that simplifies programming for intermittent systems and ensures volatile and nonvolatile data consistency despite near-constant interruptions. DINO is the first system to address these consistency problems in the context of intermittent execution. We evaluate DINO on three energy-harvesting hardware platforms running different applications. The applications fail and exhibit error without DINO, but run correctly with DINO's modest 1.8–2.7× run-time overhead. DINO also dramatically simplifies programming, reducing the set of possible failure-related control transfers by 5–9×.

Categories and Subject Descriptors C.3 [*Special-purpose and application-based systems*]: Real-time and embedded systems; D.4.5 [*Reliability*]: Checkpoint/restart

Keywords Intermittent computing

1. Introduction

Increasing energy efficiency has lowered the energy cost of computation so far that general-purpose microcontrollers can operate solely on energy they can scavenge from their surroundings [14, 25]. Unlike traditional machines with tethered power or batteries, energy-harvesting computers boot quickly from tiny energy buffers and operate *intermittently*. Execution can be interrupted by a power failure at any point.

Early prototypes of intermittently powered computers acted as programmable, sensor-laden RFID tags and used on-chip flash memory for program storage [31]. Subsequent efforts built applications such as handshake authentication [10], computer vision [40], user interaction [36], and data logging [38, 39]. These applications largely avoided using flash because of its unwieldy erase–write semantics, write latency that is orders of magnitude slower than RAM, limited durability, and high voltage requirements [30, 33].

Emerging nonvolatile memories ease the burden of on-chip persistent storage for microcontrollers. Ferroelectric RAM (FRAM) in production chips offers convenience, speed, durability, and energy characteristics closer to those of RAM [34]. Programmers can use memory-mapped FRAM to store variables that will survive power failures. Recent work noted that fast, accessible nonvolatile storage can simplify programming models by abstracting process lifecycles and working sets [4, 8], appearing to the programmer to offer persistence "for free."

This paper demonstrates that intermittent execution will thwart programmers tempted by "free" or cheap persistence. Embedded ISAs and compilers do not distinguish between writes to nonvolatile and volatile memory, exposing simple load/store interfaces that assume the programmer will use hardware correctly—and leaving programs responsible for data consistency. Under intermittent execution on real hardware platforms, partially executed code and repeated code result in consistency violations that can break program invariants or corrupt outputs. Power failures at arbitrary times introduce implicit control flow that stymies automated analysis and complicate programmer reasoning. Worse, sudden power failures can lead to program states that are *unreachable in any continuous execution*, rendering embedded systems unsafe or unusable.

This paper presents DINO (*Death Is Not an Option*), a new programming and execution model that addresses the challenges posed above. In DINO's programming model, programmers insert *task boundaries* to subdivide long-running computations into semantically meaningful shorter tasks, such as sampling a sensor or manipulating an important buffer. *Tasks* are dynamically formed spans of instructions between task boundaries. Tasks have well-defined transactional semantics: the program's state at a task boundary is guaranteed to be consistent with the completed execution of the task that preceded it. In contrast to software transactional memories (STMs) that clearly distinguish operations protected by transactions, in DINO *every* instruction executes in a transaction.

To support this programming model, DINO's execution model uses judicious checkpointing and recovery that tracks volatile *and* nonvolatile state. This approach is unlike previous systems that track only volatile state *and permit consistency violations involving nonvolatile state* [16, 28]. By executing all instructions in transactional tasks, DINO guarantees that *intermittent* execution behavior is equivalent to *continuous* execution behavior. This guarantee simplifies programming by eliminating potential failure-induced control transfers. When a failure occurs, execution simply resumes at the task boundary that began the current task.

PLDI'15, June 13–17, 2015, Portland, OR, USA
ACM. 978-1-4503-3468-6/15/06
http://dx.doi.org/10.1145/2737924.2737978

This paper makes the following contributions:

- We define the *Intermittent Execution Model* and present two ways to model intermittence, namely as concurrency and control flow. We use both models to characterize, for the first time, several problems that threaten application consistency on intermittently powered embedded devices.
- We resolve these problems with the DINO programming and execution model, which provides task-based programming and task-atomic execution to avoid consistency violations under intermittent power.
- We evaluate a working prototype of DINO, including a compiler and runtime system for embedded energy-harvesting platforms. We evaluate DINO on diverse real systems and applications and show that DINO provides its guarantees effectively and efficiently.

2. Intermittent Execution: Key Challenges

Intermittent execution presents fundamental, unsolved challenges to programmers of energy-harvesting systems that have volatile and nonvolatile state. This work's goal is to address these challenges and provide a reliable, intuitive foundation for such systems. This section describes an *intermittent execution model* to facilitate reasoning about programs on intermittently powered devices. It formalizes and enriches the implicit model of previous work [6, 16, 28] and explains with an example why prior approaches relying only on dynamic checkpointing are insufficient to prevent consistency violations. We then present two equivalent models for reasoning about intermittence: one that connects intermittence to concurrency and one that frames intermittence as a control-flow problem. We use the models to illustrate how intermittence can lead to inconsistent memory states that cannot occur in any continuous execution.

2.1 The Intermittent Execution Model

This work is premised on an intermittently powered hardware platform with volatile and nonvolatile memory, e.g., TI's Wolverine [34]. As on other embedded systems, there is no OS; the program has full access to all addresses and peripherals. The platform runs on harvested energy held in a small buffer that smooths fluctuations; it does not spend precious energy charging a battery. Section 6 describes three systems that fit this description.

The *intermittent execution model* describes the behavior of devices in this class. Figure 1 uses an example to contrast intermittence with conventional execution. An intermittent execution of a program is composed of periods of sequential execution interrupted by *reboots*. A key difference between an intermittent execution and a continuous one is that a reboot is not the end of an intermittent execution. Between reboots, instructions execute sequentially, as in a standard execution model. A reboot may occur before, during, or after any instruction. Rebooting has two effects on execution: all volatile memory is cleared, and control returns to the entry point of `main()`. Nonvolatile state retains its contents across reboots. Periods of execution are on the order of a few hundred to a few thousand machine instructions, in line with the intermittence characteristics of prototype intermittently powered devices [6, 16, 28, 31, 40]. Thus, failures are the common case in intermittent execution.

2.1.1 Periodic Dynamic Checkpointing

Recent efforts used *periodic dynamic checkpointing* [16, 28] to foster computational progress despite intermittent execution. Dynamic analysis determines when to copy the execution context—registers and some parts of volatile memory—to a reserved area in nonvolatile memory. After a reboot, execution resumes at the checkpoint rather than `main()`.

Checkpointing is insufficient. Dynamic checkpointing enables progress and ensures correctness for programs that use only volatile state. However, past work does not address two problems that are fundamental to the intermittent execution model. The first problem is that dynamic checkpoints are opaque and implicit: a programmer or static program analysis is forced to guess where execution will resume after a reboot, and every instruction that can execute is a candidate. Inferring the set of possible resumption states requires complex reasoning about many program scopes and functions. Non-idempotent operations, like I/O and nonvolatile memory accesses, may be unpredictably repeated or partially completed. Under dynamic checkpointing alone, these factors confound analysis by programmers and compilers.

The second fundamental problem is that, despite its *per*sistence, nonvolatile memory does not necessarily remain *con*sistent across reboots—potentially leaving it in a state that is *not permitted by any continuous execution*. Errors from partial or repeated execution can accumulate in nonvolatile memory, with results ranging from data-structure corruption to buffer overflows. Such problems may occur even in code that is correct under continuous execution. The potential for memory inconsistency forces the programmer to manually and onerously reason about, check, and enforce consistency wherever nonvolatile data is accessed.

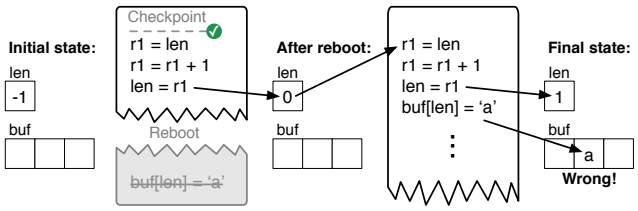

Figure 2: Intermittence causes surprising failures. The operations before the reboot update `len` but not `buf`. When execution resumes—e.g., from a checkpoint—`r1` gets the incremented value of `len` and the code writes a character into `buf`'s second entry, not its first. Updating `buf` leaves the data inconsistent: only `buf`'s second entry contains `a`, which is *impossible in a continuous execution.*

2.2 Intermittence Causes Surprising Failures

Figure 2 shows how intermittence causes a failure using the example code from Figure 1. The code updating `len` and `buf` should not be interrupted, but a reboot occurs after `len`'s update and before `buf`'s update. Execution resumes at the previous checkpoint and `len` is updated again. The timing of the reboot leads to data corruption: `len` is updated twice, despite only one call to `append()` in the program. When `buf` is finally updated using the value of `len`, its *second* entry is updated, not its first, as should be the case. Crucially, *the final memory state is impossible in a continuous execution*.

Reasoning only about sequential behaviors explicit in a program does not reveal failures like the one in Figure 2. Even a program that is provably free of buffer overflows can suffer one under intermittent execution. In response to the shortcomings of conventional reasoning, programmers need new models for reasoning about intermittent execution.

2.3 Reasoning about Intermittent Execution

We develop two new models for reasoning about intermittent execution, one framing intermittence as a form of concurrency, the other describing intermittence as control-flow. We later use these

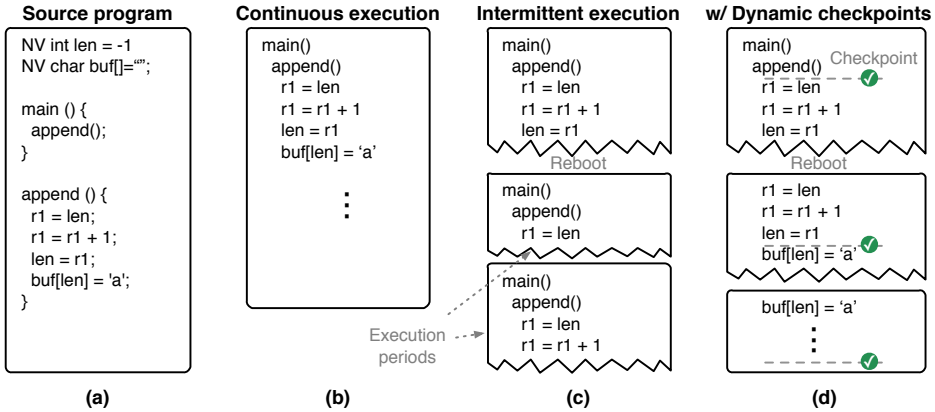

Figure 1: Comparing execution models. (a) The program has nonvolatile variables `len` and `buf`; `append()` modifies both. (b) Continuous execution runs the program to completion. (c) Intermittent execution experiences reboots that return control to the start of `main()`. (d) Intermittent execution with periodic checkpointing preserves state across reboots and the program makes progress, completing.

models to address the problems described above. Seen as *concurrency*, intermittence leads to *data races* that lead to unintuitive results. Seen as *control flow*, intermittence introduces new control-flows that are hard to reason about and can cause unintuitive data-flow. Both models assume periodic checkpoints of volatile state.

2.3.1 Modeling Intermittence as Concurrency

We model an intermittent execution as a collection of concurrent execution periods. Each period executes the same code, beginning at the most recent checkpoint. Concurrent periods are *scheduled*; At first, one period executes and all others unscheduled. A reboot corresponds to the executing period being *pre-empted* and a *pre-empting* period begins executing. Pre-empted periods never resume. Accesses to nonvolatile memory in *pre-empting* and *pre-empted* are *concurrent*. Figure 3(a) illustrates an intermittent execution as a concurrent execution of execution periods that preempt each other when the execution experiences a reboot.

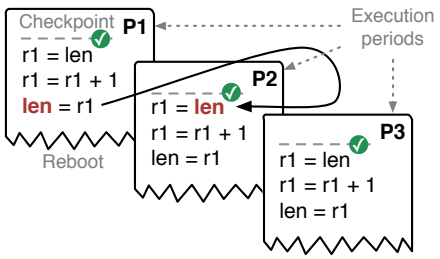

Figure 3: Intermittent execution as concurrent execution. The solid arrow highlights two unintuitive consequences of intermittent execution: (1) the second execution period P2 preempts the first period P1 after P1 completes the write to the nonvolatile variable `len`, causing a data race with P2's read of `len`. (2) Data flows along the solid arrow from P1 to P2, resulting in the read of `len` receiving a value from a write that is *lexically later* in the program—which is impossible in a continuous execution.

Concurrent accesses may form *data races*. A data race is a pair of memory accesses, at least one of which is a write, that are not happens-before ordered [18]. In general, data races can produce unintuitive results [1, 21], and which races are problematic depends on the system's memory model.

Racing accesses to non-volatile memory in an intermittent execution can be problematic. A race leads to an unintuitive result if a pre-empted execution period modifies a nonvolatile memory location, a pre-empting execution period reads the same location, and the pre-empting period's read precedes all of its writes to the location. Such a data race incorrectly exposes state modified by the pre-empted execution period to the pre-empting one. Making matters worse, errors compound over time because such unintuitive values are used by subsequent computation.

Figure 3(b) illustrates how a data race between a pre-empted and a pre-empting task can lead to such unintuitive outcomes. The value written to `len` in the pre-empted task appears in the read of `len` in the pre-empting task. Section 3.1 elaborates on the inconsistency in this example.

2.3.2 Modeling Intermittence as Control Flow

We model intermittence as control flow, observing that a reboot redirects control from an arbitrary failure point to some prior point. Reboots introduce edges in the control-flow graph (CFG) that are followed on a reboot. A CFG is a graph, $G = (V, E)$, where each node in V represents a static code point and E is the set of explicit control-flow edges in the program.

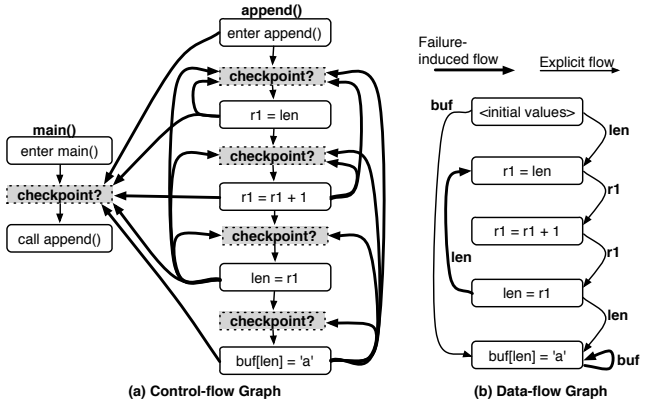

Figure 4: Intermittent execution as control-flow. (a) The CFG for the program in Figure 1(a) with implicit, failure-induced control-flow edges. Possible checkpoints are dotted boxes. A failure-induced edge begins at each code point and targets each possible checkpoint. **(b)** The DFG for the CFG in **(a)** with failure-induced data-flows added.

We extend G to $G' = (V, E, F)$ where F is a set of *failure-induced control-flow edges*. Each edge $f \in F$ originates at a node

$v_{reboot} \in V$ and terminates at another node $v_{resume} \in V$. For every v_{reboot} in V where a reboot may occur, there is an edge $f \in F$ for each possible v_{resume} in V where a checkpoint may be taken and $f = (v_{reboot}, v_{resume})$. A system may dynamically decide to collect a checkpoint at any time, making the number of edges in F approximately $|V|^2$. Note that in our formulation of G', a particular edge $(v_{reboot}, v_{resume}) \in F$ is followed when the system collects a checkpoint at v_{resume} and reboots at v_{reboot}. Figure 4 depicts a CFG augmented with failure-induced control-flow edges.

Failure-induced data flow. A CFG with nodes representing nonvolatile reads and writes has a corresponding *NV data-flow graph* (NV DFG). The DFG encodes which writes' results a read may read. We assume nonvolatile memory locations are uniquely, globally identified, which is reasonable for embedded programs, and build use–def chains as follows. An NV DFG, $D = (V_D, E_D)$, has vertices V_D corresponding to the CFG nodes that represent nonvolatile memory accesses. A node records its type (i.e., read vs. write), and the location it accesses. An edge in the DFG exists between nodes, $u_d, v_d \in V_D$, if u_d writes some memory location, v_d accesses the same location, and there is a path in the CFG from the node corresponding to u_d to the one corresponding to v_d that does not pass through any other write to the same location. Such a CFG path means v_d could read or overwrite what u_d wrote.

With intermittence, data can flow from one execution period to another across a reboot. Such a flow exists if the CFG path admitting a NV DFG edge *includes a failure-induced control-flow edge*. Such failure-induced data flow can leave nonvolatile data in a state that does not correspond to any continuous execution of the program. Section 3 describes the types of inconsistencies that arise.

3. Intermittence Causes Data Inconsistency

We consider the state of an intermittent execution *consistent* if it corresponds to one produced by some continuous execution, and *inconsistent* otherwise. Under Section 2's models of intermittent execution, intermittence can lead to several kinds of inconsistency in nonvolatile state. *Atomicity* violations occur when consecutive executions accidentally share state because the earlier execution did not complete. The broader category of *idempotence* violations encompasses atomicity violations along with other application-level problems that arise when updates occur more times than a continuous execution would permit.

3.1 Atomicity Violations

In our model, a sequence of actions is *atomic* if code outside the sequence—i.e., code executing after a reboot—cannot observe its effects until all operations in the sequence complete. Such an atomic sequence should also be *isolated*: operations in the sequence can observe the results of operations in the sequence, or initial values from the start of the sequence, but no results from operations that are not part of the sequence—including the same code executing before a reboot. For simplicity, we call code *atomic* when it meets *both* conditions.

Using our concurrency model of intermittence (§2.3.1), we can now reason about the surprising failure in Figure 2 as an atomicity violation. The updates to `len` and `buf` should be *atomic*. The pre-reboot and post-reboot periods are pre-empted and pre-empting periods respectively. The pre-empting period violates the atomicity of the pre-empted period because the reboot precedes the write to `buf`. The pre-empting period observes the partially updated pair of variables, manifesting the violation.

3.2 Idempotence Violations

A computation is idempotent if it can be repeatedly executed without producing a new result. We say that an execution experiences

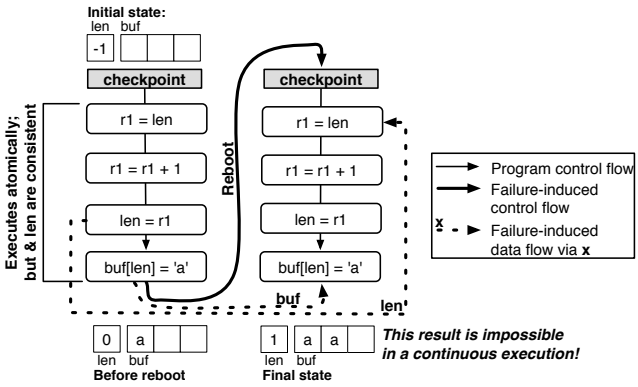

Figure 5: Intermittence causes idempotence violations. In this intermittent execution of the program in Figure 1(a), the updates to `buf` and `len` are atomic, so the two values remain internally consistent. However, the update is non-idempotently repeated after the failure. A failure-induced control-flow edge creates a loop around the update, leaving `buf` and `len` inconsistent with the execution context, unlike in any continuous execution.

an *idempotence violation* when non-idempotent computation is repeatedly executed, producing a new result each time. An intermittent execution manifests an idempotence violation when it executes non-idempotent code, reboots, resumes executing at a checkpoint, and re-executes the non-idempotent code.

Figure 5 uses our control-flow model of intermittence (Section 2.3.2) to illustrate an idempotence violation in an execution of the code in Figure 1(a). The execution atomically updates `len` and `buf` and then experiences a reboot, following a failure-induced control-flow edge to the beginning of `append()`. The execution updates `len` and `buf` a second time. The failure-induced control-flow edge creates a loop in the otherwise straight-line code. The updated value of the variables flows around the loop, from the accesses before the failure to the accesses after the failure. When the single call to `append()` in the source code completes, `len` and `buf` reflect the effect of *two* calls to `append()`. The resulting memory state is not possible in a continuous execution of the code shown. Importantly, this execution does not suffer an atomicity violation: both memory locations are updated atomically in each execution of `append()`. The key is that the updates to these variables are not idempotent. Each repeated execution of those updates incorrectly produces a new result.

4. DINO Programming and Execution Model

DINO is a programming model and an execution model that, together, make intermittently powered systems robust to the problems resulting from frequent interruptions. Programmers place *task boundaries* in their code. At runtime, dynamic spans of instruction between task boundaries form *tasks*. DINO's programming model assigns clear *task-atomic* semantics to programs to enable simple reasoning—both human and automatic—about data consistency and control flow. DINO's execution model ensures that tasks are atomic and data are consistent at task boundaries. DINO uses data-flow analysis and compiler-aided task-cost analysis to help programmers place task boundaries in a way that ensures consistency and minimizes overhead.

4.1 Programming Model

DINO's programming model assumes a C-like base language because such languages predominate in embedded development, though it does not depend on language features that are unique to C. DINO adds several features to the base programming model.

4.1.1 Task Boundaries and Atomic Tasks

DINO adds *task-atomic* semantics to C's programming model. The programmer statically defines task boundaries at code points where they expect data consistency. Conceptually, programmers should think of task boundaries like memory fences that are meaningful at compile time, rather than thinking of tasks as statically defined regions that happen to have delimiting boundaries, because task boundaries are defined *statically*, while tasks (paths between task boundaries) are formed *dynamically*. As with memory fences in concurrent programs, programmers must be careful to include sufficient task boundaries to provide the atomicity and consistency required by their application. Programmers should pay special attention to placing boundaries to cover all possible control-flow paths when an application needs atomicity of control-dependent code regions. Figure 6(a) shows part of an activity recognition (AR) program (§6.1) decorated with task boundaries and slightly modified for illustrative purposes. The figure shows statically placed task boundaries in line with program code. Figure 6(b) shows how a task is dynamically defined, during an execution, as a path from one task boundary to the next. In the figure, one task spans from operation 6 to operation 4 including code on just one side of a branch and the return from `upd_stats()`.

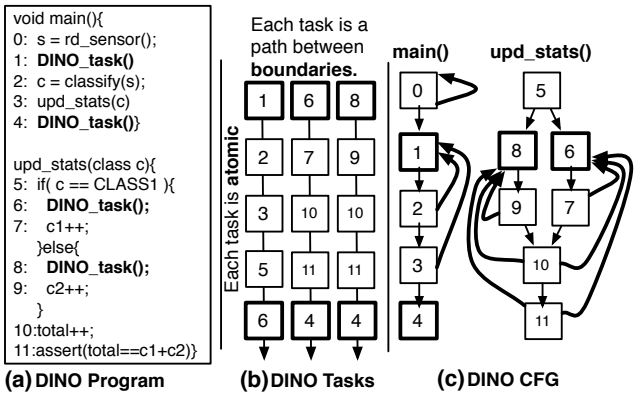

Figure 6: The DINO Programming Model. (a) shows static task boundaries in code. **(b)** shows tasks formed dynamically from the boundaries in (a). Bold boxes are task boundaries. **(c)** shows how task boundaries are failure-induced control-flow targets in a control-flow graph (CFG). Bold CFG edges are failure-induced.

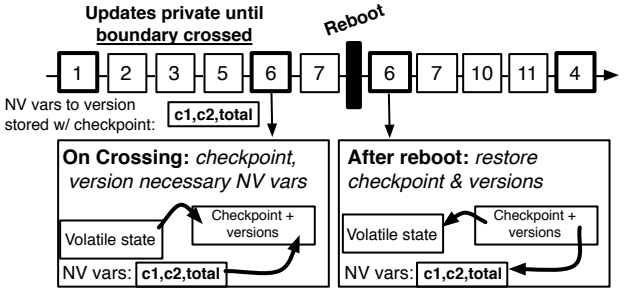

Figure 7: The DINO Execution Model. The boxes at the top show an execution from Figure 6. Bold boxes are task boundaries. The tiles at the bottom show how DINO checkpoints and versions data at a task boundary and recovers to a task boundary after a failure.

Task boundaries in DINO have several important aspects. First, operations in a task execute atomically and updates made in a task are isolated until the task completes. Isolation implies that a task sees the values in variables at its initial boundary or the results of its own updates. Atomicity implies that one task's updates are not visible to other tasks (i.e., after a reboot) until its terminal boundary executes. Second, both volatile and nonvolatile data are guaranteed to be consistent at task boundaries, whose function is analogous to that of synchronization, like memory fences or barriers in conventional concurrent programs. Third, the boundary from which execution will resume after a failure is the last one crossed dynamically before the failure.

DINO's tasks are akin to transaction regions in TCC [15], which require only that programmers insert transaction boundaries into their code and provides transactional semantics via hardware enhancements, or bulk-sequential chunks as in BulkSC [7]. Unlike TCC and BulkSC, DINO does not require hardware support, making it closer in spirit to software transactional memory (STM), but without the need to statically demarcate regions of code as transactions; DINO instead forms tasks dynamically and protects *every* instruction with a transaction.

4.2 Execution Model

DINO provides runtime support for *checkpointing* and *data versioning* to keep data consistent despite reboots.

Checkpointing. State preservation across reboots is necessary to ensure progress. At every task boundary, a checkpointing mechanism (like that of QuickRecall [16] or Mementos [28]) in the DINO runtime copies the registers and stack to a linker-reserved area in nonvolatile memory. Checkpoints are double buffered so that newly written checkpoints supersede prior ones only after being completely written. To protect against failures during checkpointing, the final operation of a checkpoint is to write a canary value that must be intact for a checkpoint to be considered restorable. In contrast to the dynamic checkpointing described above, DINO takes checkpoints only at statically placed *explicit* task boundaries. Capturing execution context in a nonvolatile checkpoint at each task boundary ensures that *volatile* data are consistent across reboots.

Data versioning. A mechanism we call *data versioning* ensures that *nonvolatile* data remain consistent across reboots. DINO's execution model ensures that, at task boundaries, nonvolatile data are both internally consistent and consistent with the task boundary's checkpoint of volatile state. When execution reaches a task boundary, immediately before checkpointing, DINO makes a volatile copy of each nonvolatile variable that is *potentially inconsistent*—i.e., may be written non-idempotently *after* a task boundary executes and before another task boundary executes. Any variable with a failure-induced data-flow edge is potentially inconsistent.

Copying nonvolatile variables to volatile memory on the stack ensures that checkpoints include these variables' values. DINO's implementation of task boundaries includes corresponding code to restore nonvolatile variables from their checkpointed volatile versions when execution resumes. Section 5.1.1 gives more details on the data-flow analysis that informs versioning decisions.

Figure 7 illustrates checkpointing and versioning during an execution with a reboot. The key idea is that data are preserved by DINO when execution passes a task boundary and restored by DINO at that task boundary after a reboot.

4.3 DINO Prevents Consistency Violations

DINO's versioning and checkpointing prevent atomicity violations and idempotence violations by design. From the perspective of a programmer or program analysis, a task boundary represents a point in the execution with guaranteed data consistency and equivalence to the state of some continuous execution.

Figure 8 shows how DINO prevents an atomicity violation due to intermittent execution. Note that when the failure occurs, mem-

ory is in an *inconsistent* state, but upon resumption DINO restores the consistent state that is equivalent to a continuous execution.

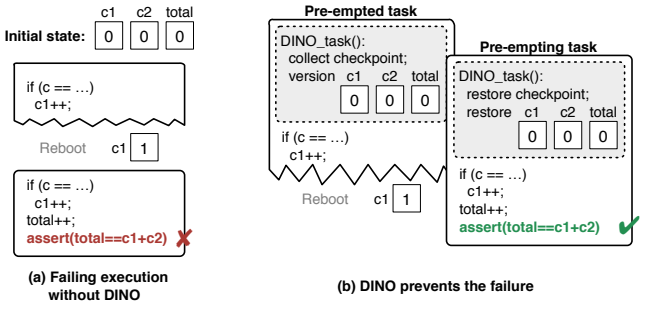

Figure 8: DINO prevents atomicity violations. (a) shows an execution of the code in Figure 6. Without DINO, the reboot violates atomicity, leaving `total`, `c1`, and `c2` inconsistent. **(b)** shows how DINO prevents the violation by versioning `c1`, `c2`, and `total` in the pre-empted task. After the reboot DINO restores the versions, isolating the pre-empting task from the incorrect partial update.

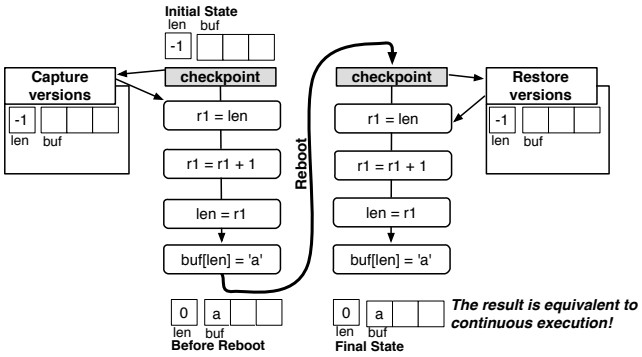

Figure 9: DINO prevents idempotence violations. The figure shows an execution of the program in Figure 1, where a reboot causes writes to `buf` and `len` to erroneously re-execute. Here, DINO versions `buf` and `len` with the checkpoint. After the reboot, DINO restores the versions, eliminating the failure-induced data-flow.

Figure 9 illustrates how DINO's data versioning prevents an idempotence violation. Note that `buf` and `len` are mutually *consistent* when the reboot occurs. The problem is that when execution resumes, the data together are inconsistent with the checkpointed, volatile execution context (i.e., register state and program counter). Before DINO resumes application code, it restores `buf` and `len` to the consistent values that correspond to a continuous execution.

Proof sketch: We sketch a formal proof that DINO is correct. We refer to the non-volatile memory as NV and the volatile memory as V. NV and V are consistent at a point in the execution if they are equivalent to NV and V at that point in a continuous execution. The proof is by induction on the number of failures.

—*Base case:* Before any failures, at a task T's initial boundary, B, DINO versions locations in NV written by writes reachable on any path starting from B that does not cross another task boundary, producing NV_v. At B, DINO also checkpoints V to V_c. NV_v and V_c are consistent at B, with no prior failures. If T is interrupted, execution reverts to B and DINO restores NV from NV_v and V from V_c. Thus, after one failure, execution is at B, and NV and V are consistent.

—*Induction case:* Arbitrary, repeated failures of task T each revert NV to NV_v and V to V_c at B, which is consistent. When T ends, its terminal boundary B' is the initial boundary of the next task T'. Upon reaching B', NV and V are consistent (possibly having been restored at B). B' is the initial boundary of T', thus DINO captures NV' to NV'_v and V' to V'_c at B', which is consistent. Subsequent failures in T' revert to NV'_v and V'_c at B', which is consistent. □

4.4 DINO Reduces Control- and Data-flow Complexity

DINO uses statically defined task boundaries, addressing the high control-flow complexity of systems that dynamically collect checkpoints. Figure 6(c) illustrates the CFG for the program in Figure 6(a). A key advantage of DINO is that the target of each failure-induced control-flow edge is statically known, lowering the CFG's complexity. With dynamic checkpointing, there is a CFG edge from each point in a program where a reboot can occur to each earlier point in the code that might be a checkpoint. With DINO, there is a CFG edge from each point where a reboot can occur to only those task boundaries that reach that point in the CFG *without first crossing another task boundary*. In Figure 6(c), there are nine failure-induced CFG edges. We omit the complex, full CFG for a dynamic checkpointing system, which includes 27 failure-induced edges, not including inter-procedural ones.

DINO's task-atomicity eliminates the failure-induced, non-volatile data-flow suffered by intermittent executions. These edges are eliminated because tasks are atomic and isolated. DINO's simpler control flow and absence of failure-induced data flow simplifies reasoning about failures.

Idempotence. The problem of preventing non-idempotent updates to data structures, as in Figure 9, is distinct from the problem of non-idempotent actions at higher layers. DINO cannot un-launch a rocket or un-toast bread. Programmers must decide whether intermittent power is appropriate for a given use case. With careful placement of task boundaries, DINO *can* allow for delicate handling of certain regions of code. For example, placing a task boundary immediately after a sensitive operation maximizes the probability that the operation will not repeat, i.e., that DINO can fully save the post-operation state before the next failure.

4.5 Feedback on Task Boundary Placement

DINO requires programmers to manually place task boundaries, which makes implicit control flow explicit. It also suggests that programmers think about the *cost* of tasks in two ways: task-boundary overhead and failure-recovery cost. DINO exposes costs with compiler warnings.

Task boundaries introduce checkpointing and versioning overheads. The checkpointing overhead increases with the size of the stack. Versioning overhead increases with the size of nonvolatile data accessed in a task. DINO's compiler emits a warning for each boundary that can be moved to a point of lower cost (e.g., where the stack is smaller).

The second cost dimension is that failures of irrevocable actions such as I/O may be expensive to repair. Making such tasks very short and minimizing unrelated actions minimizes this risk. DINO's compiler emits a warning when a task includes access to a peripheral register (indicating likely non-idempotent hardware use) and suggests that the programmer try to constrict task boundaries as close as possible around the access. Section 5.1.3 gives details on both cost analyses.

5. Architecture and Implementation

DINO implements the design described in Section 4. Its main components are (1) a compiler that analyzes programs and inserts DINO runtime code and (2) a runtime system that directly implements the DINO execution model, including checkpointing, data versioning, and recovery.

5.1 Compiler

The DINO compiler is a series of passes for the LLVM compiler framework [19]. The DINO compiler uses data-flow analysis to identify potentially inconsistent data that must be versioned at a task boundary. It translates programmer-defined task boundaries into calls into the DINO runtime library. Finally, it analyzes task boundary placement and suggests changes to reduce run-time checkpointing cost.

5.1.1 Identifying Potentially Inconsistent Data

The compiler determines, for each task boundary, which potentially inconsistent variables must be versioned to provide consistency upon resumption. It uses an interprocedural context- and flow-sensitive analysis to find these variables. For each nonvolatile store S to a location L_S, the analysis searches backward along all control-flow paths until it hits a "most recent" task boundary on each path. Since DINO tasks are *task atomic*, the analysis of each path can stop when it finds a task boundary. Our prototype assumes no nonvolatile accesses before the program's first task boundary. These task boundaries are points at which execution might resume if a reboot follows S's execution. Between each such task boundary TB and S, the analysis looks for loads from L_S that occur before S. If a load from L_S precedes the store S, then the analysis has found a failure-induced data flow that would allow the load to observe the value of S. The DINO compiler adds versioning for S at TB, guaranteeing that the load between TB and S will observe the value it had upon entry to TB—not the value written by S before the reboot. Figure 7 depicts the addition of versioning.

DINO explores CFG paths that include loop backedges at most once. When DINO encounters a call, its analysis descends into the call and continues exploring paths. Likewise, when DINO reaches the beginning of a function in its backward traversal, it continues along paths through call sites of the function. DINO's analysis therefore requires access to the full program code. Whole-program analysis is not likely to be problematic because in embedded systems, complete source code is usually available.

5.1.2 Compiling Task Boundaries

The programmer inserts task boundaries as calls to a `DINO_task()` function. Encountering such a call, the DINO compiler first determines the set of nonvolatile variables to version using the analysis in Section 5.1.1. It then replaces the call to `DINO_task()` with calls into the DINO runtime library that version the relevant nonvolatile state (`dino_version()`) and checkpoint volatile state (`dino_checkpoint()`). The program is linked to the DINO runtime library that implements these functions (§5.2).

5.1.3 Analyzing Task Cost

In this work, we use two simple heuristics to assess task cost. We leave the development of other heuristics for DINO's analysis framework for future work. The first heuristic computes the size of the stack DINO must checkpoint. It sums the size of LLVM IR stack allocations (`alloca` instructions) in the function that contains the task boundary, then compares that sum to the sizes of stack allocations in the function's callers. If the count in a caller is smaller, then moving the task boundary to the caller would decrease the amount of stack data that needs to be checkpointed. The compiler emits an actionable warning suggesting the programmer move the task boundary into the caller.

The second heuristic estimates the likelihood that a task will experience a reboot because it uses a power-hungry peripheral. The compiler identifies accesses to peripherals, which are statically memory mapped on most embedded systems, by finding memory accesses with peripheral addresses as their arguments. Encountering such an access, the DINO compiler emits a warning that the task accesses a peripheral and is more likely to experience a reboot.

Table 1: Overview of embedded systems used for evaluation. Each application works under continuous power but fails under intermittent power without DINO. The table also summarizes nonvolatile data use, lines of code, and number of DINO task boundaries.

App.	Description & Inconsistency	Power	NV Data	LoC	TBs
AR	Classifies accelerometer data; atomicity violation leaves stats inconsistent	RF	3x2B counters	250	5
DS	Logs data into histogram sorted by key; atomicity violation loses key's bin	RF	10x4B key/val	312	5
MI	Self-powered MIDI interface; idempotence violation corrupts buffer	Rotor	4x5B msgs. + 4B Tx buf	210	4

5.2 Runtime System

DINO's runtime system implements the checkpointing and versioning required by its execution model. Static checkpointing uses a version of the checkpointing mechanism from Mementos [28] that we ported to the Wolverine FRAM microcontroller [34]. This mechanism checkpoints the contents of registers and the stack. It does does not checkpoint heap data, and it currently checkpoints only global variables that are explicitly tagged by the programmer, but neither limitation is fundamental to our design.

When the program begins in `main()`, it first executes a call to `restore_state()` inserted by the DINO compiler. This runtime library function copies the data in the current checkpoint back into the original, corresponding registers and memory locations, saving the program counter, stack pointer, and frame pointer for last. The function copies the versioned nonvolatile variables in the checkpoint back into their original nonvolatile memory locations. After restoring volatile and nonvolatile data, the function restores instruction, stack, and frame pointer registers, redirecting control to the point where DINO collected the checkpoint. From there, execution continues.

5.3 Why *not* rely on hardware support?

Hardware support could be useful for implementing DINO. Hardware could accelerate checkpointing, provide transactional memory, or aid energy-aware scheduling. We opted not to rely on hardware support for several reasons. First, systems that suffer consistency problems are already widely available [27, 31, 34, 36]. Without special hardware requirements, DINO is applicable to these systems today, as well as to future systems. Second, new hardware features can increase energy requirements when underutilized [13], increase design complexity, and increase device cost. Third, specialized hardware support requiring ISA changes raises new programmability barriers and complicates compiler design [1, 32].

6. Applications

We built three hardware/software embedded systems to evaluate DINO (Table 1). Each runs on a different energy-harvesting front end. All three use nonvolatile data, demand consistency for correctness, and fail or suffer error because of inconsistency under intermittent execution.

6.1 Activity Recognition

We adapted a machine-learning–based activity-recognition (AR) system from published work [17] to run on the intermittently powered WISP hardware platform [31]. The WISP harvests radio-frequency (RF) energy with a dipole PCB antenna that charges a small capacitor via a charge pump. The WISP has an Analog Devices ADXL326z low-power accelerometer connected to an MSP430FR5969 MCU [34] via 4-wire SPI. AR maintains a time

series of three-dimensional accelerometer values and converts them to two features: mean and standard deviation vector magnitude. It uses a variant of a nearest-neighbor classifier (trained with continuous power) to distinguish shaking from resting (akin to tremor or fall detection [11, 20, 29]). AR counts total and per-class classifications in nonvolatile memory. Each experiment collects and classifies samples until the total count of classifications reaches 10,000, then terminates, lighting an LED.

After classifying, AR's code updates the total and per-class counts. The sequence of operations that updates these counts must execute atomically, or else *only one* of them will be updated. If the counters fall out of sync, AR makes progress, but its counts are inconsistent. We focus on that invariant of AR's output: the per-class counts should sum to the total count and any discrepancy represents error.

6.2 Data Summarizer

We implemented a data-summarization (DS) application on TI's TS430RGZ-48C project board with an MSP430FR5969 microcontroller [34]. We connected the board to a Powercast Powerharvester P2110 energy-harvester [27] and a half-wavelength wire antenna, all mounted on a breadboard.

DS summarizes data as a key–value histogram in which each key maps to a frequency value. One function adds a new data sample to the histogram by locating and incrementing the corresponding bin. Another function sorts the histogram by value using insertion sort. Our test harness for DS inserts random values, counting 2000 insertions with a nonvolatile counter and sorting after every 20 insertions. The sorting routine swaps histogram keys using a volatile temporary variable. If a swap is interrupted before the key in the temporary is re-inserted, that key is lost from the histogram, and two bins end up with the same key. The structural invariant for DS, which the test harness checks after each sorting step, is that each key appears exactly once in the histogram. If the check fails, the histogram is in an inconsistent state and DS halts, illuminating an LED to indicate the error state.

6.3 MIDI Interface

We implemented a radial MIDI (Musical Instrument Digital Interface [22]) interface (MI) on TI's TS430RGZ-48C project board with an MSP430FR5969 microcontroller [34]. We connected the project board to a Peppermill power front end [36] that harvests the mechanical power of a manually rotated DC motor for use by the project board. We drove the Peppermill with a repurposed drill motor. The Peppermill exposes an output pin whose voltage is proportional to the motor's rotational speed, which we connected to the project board's analog-to-digital converter.

MI generates batches of MIDI *Note On* messages with a fixed pitch and a velocity proportional to the motor's speed. It stores messages in a circular-list data structure and tracks the index of the message being assembled, incrementing the index to store each new message. When all entries are populated, MI copies the messages to a separate memory region,[1] clears the old messages, and resets the index. A power failure can trigger an idempotence violation that increments the index multiple times, eventually leaving it referring to a nonexistent entry. MI checks that the index is in bounds before using the buffer and aborts with an error LED if it is not. In each experiment, MI generated messages in batches of four and terminated after 10,000 messages.

7. Evaluation

We evaluated DINO using the applications described in Section 6. Our evaluation has several goals. First, we show that DINO keeps data consistent despite intermittence. Second, we show that DINO's overheads are reasonable, especially given its correctness gains. Third, combining experimental observations and our control-flow model for reasoning about intermittence 2.3, we show that DINO reduces the complexity of reasoning about control flow. Fourth, we show that the our task-cost analysis correctly reports costly tasks.

[1] This region would be a transmit buffer if our prototype included a radio.

Table 2: Failures and Error in AR, DS, and MI. The data count failing (✗) and non-failing (✓) trials with and without DINO for MI and for DS at 60 cm, and error at 40 cm for AR. The table also shows reboots and time overhead for AR at 40 cm and DS at 60 cm. Reboots and time overhead are not meaningful for MI, as it is a reactive UI.

	AR			DS				MI	
Config.	**Err.**	**Rbts.**	**Ovhd.**	**✗**	**✓**	**Rbts.**	**Ovhd.**	**✗**	**✓**
Baseline	6.8%	1,970	1.0x	3	7	13.4	1.0x	10	0
DINO	0.0%	2,619	1.8x	0	11	72.5	2.7x	0	10

7.1 DINO Enforces Consistency

The main result of our experiments is that DINO prevents all of the inconsistencies the applications suffer without DINO.

Columns 2–3 of Table 2 show the error AR experiences with and without DINO at 40 cm from RF power (mean of five trials). To provide RF power, we use an Impinj Speedway Revolution R220 commercial RFID reader with a circularly polarized Laird S9028PCR panel antenna emitting a carrier wave at 30 dBm (1 W). AR sees **no error** with DINO but suffers **nearly 7% error** without DINO. A Student's T-test confirms the difference is significant ($p < 0.05$).

Columns 4–5 of Table 2 show failure data for DS running with and without DINO at 60 cm from RF power. DS **does not fail** with DINO, but fails in three out of ten trials without DINO. A χ^2 test confirms the difference in failure rate is significant ($p = 0.05$). DS experiences no failures with DINO, but about 5.5× as many reboots. The explanation is that DINO's overheads (§7.2) extend run time, during which the device sees more charges and depletions. DINO keeps data consistent, regardless of the number of reboots.

Columns 9–10 of Table 2 show failure data for MI with and without DINO. MI **does not fail** with DINO, but fails **100% of the time** without DINO, processing just 975 messages on average before failing. MI reveals an important facet of DINO: naïvely porting code to an intermittent context threatens functionality. With DINO, failure-free porting requires placing just a few task boundaries.

To provide insight into how error accumulates under RF power without DINO, we repeated our experiments with AR at 10, 20, and 30 cm from RF power. AR experienced *no error* with DINO at any distance. Figure 10 plots the error rate versus distance for AR *without* DINO, showing that it is non-zero beyond a trivial distance (10 cm) and increases with distance. AR's error is strongly correlated with distance ($R = 99.64$). The plot also shows that, as expected, the number of reboots is correlated with distance ($R = 93.93$). The Friis transmission equation $P_r = P_t G_t G_r \left(\frac{\lambda}{4\pi R}\right)$ describes RF power availability at a distance. The available (analytical, not measured) power fell from 429.5 mW at 10 cm to 26.8 mW at 40 cm, explaining the increased reboots. Frequent reboots explain the error trend: More frequent reboots mean more frequent risk of interrupting sensitive code. Column 3 of Table 2 shows that at 40 cm, AR suffers even more reboots with DINO than with the baseline. Despite the increased number of reboots, DINO keeps data consistent, while the baseline's errors rise sharply.

7.2 DINO Imposes Reasonable Overheads

There are two main sources of run-time overhead. The first is time spent checkpointing and versioning data. The extra work costs

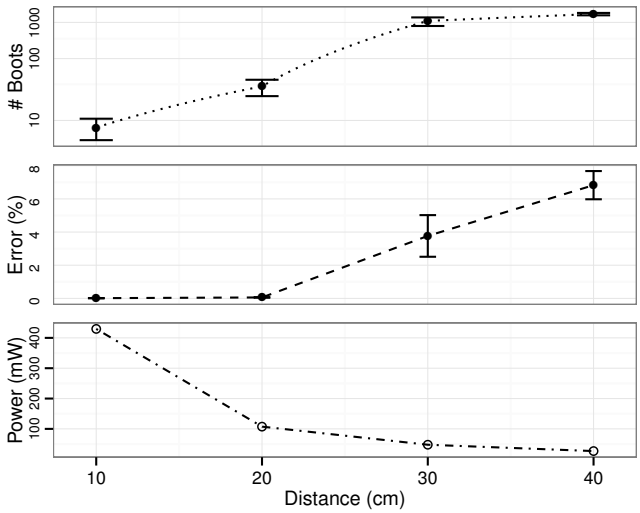

Figure 10: Error vs. Distance for AR without DINO. Under RF harvesting, AR without DINO experiences reboots (top, log scale) and error rates (middle) that scale with distance. Error bars represent standard error. The bottom plot shows analytical (computed) available power.

cycles and leads to more reboots. The second is the increase in reboots when running on intermittent power. Reboots cost cycles and require restoring data.

We externally timed each of our experiments' executions with and without DINO. The run time of DS with DINO is 2.7× higher than without DINO. The run time of AR with DINO is 1.8× higher. We omit time overhead for MI because MI is *reactive* and there is not a clear external notion of time overhead; qualitatively, the interface remained responsive throughout our trials. The number of reboots with DINO is higher by 30% for AR and about 550% higher for DS. There are two take-aways in these data. First, the overheads using DINO are reasonable: the applications fail to run (MI, DS) or suffer error (AR) without DINO. With DINO, they execute correctly and usably, despite the increased number of reboots and greater time overhead.

We also quantified DINO's storage overhead. DINO uses a checkpointing scheme that incurs a large, fixed 4 KB (2× RAM) storage cost to store checkpoints. DINO shares the statically allocated checkpoint space for versions, so there is no additional nonvolatile memory overhead for versioning.

7.3 DINO Reduces Control Complexity

DINO simplifies reasoning about failures because it has lower control-flow complexity than a dynamic checkpointing system like Mementos. We implemented a compiler analysis that counts the number of points at which execution may resume after a reboot for each instruction in each test program. For dynamic checkpointing, we treat loop headers and return instructions as potential checkpoint sites, following related work [16, 28]. For each instruction, we count sites reachable backwards, as execution may resume from any prior checkpoint, depending on the system's dynamic decisions. For DINO, we count the number of task boundaries that are reachable backwards from each instruction *without first crossing another task boundary*.

Figure 11 shows a CDF of points where execution may resume after a reboot for all instructions in AR, DS, and MI. For DINO, the number of points where execution might resume is small: never more than four and in most cases just two. For dynamic checkpointing like Mementos, the number is much higher, up to nine. For DS, a reboot at a majority of instructions could resume at any of seven

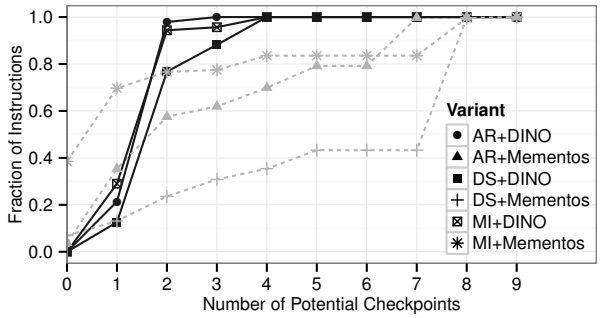

Figure 11: CDF of checkpoints reachable by each instruction. The x-axis shows at how many points execution may resume after a reboot for what fraction of instructions on the y-axis.

or more points. A reboot at 25% of instructions in AR and 20% of instructions in MI could resume at any of six or more points. Dynamic checkpointing has more points because the runtime may or may not take a checkpoint at each potential checkpoint site. The dynamism in this decision means execution could resume at *any reachable* site, rather than the just the nearest reachable site as with DINO. The more points that are reachable, the more complex the reasoning about the program.

7.4 DINO Helps Place Task Boundaries

DINO emits warnings when task boundaries could be easily moved for lower overhead or when tasks include peripheral accesses. To determine the extent to which DINO's suggestions are *actionable* and *correct*, we started with unmodified (non-DINO) application code and scripted the addition of a single task boundary at every possible program location that did not cause a compilation error.

We compiled each instrumented variant with DINO and parsed the compiler's output for warnings. First, we manually verified the correct presence of warnings for tasks with peripherals in AR, DS, and MI. Second, we characterized DINO's cost-reduction suggestions. For DS, 32% of variants generated a suggestion. Suggestions garnered a maximum reduction of checkpointed data of 50 bytes—the equivalent of 1.7× the minimum checkpoint size (registers only, empty stack). The minimum savings per task boundary was 2 bytes and the average was 8 bytes. For AR, following the compiler's suggestions yielded average savings of 6 bytes per task boundary, or 20% of the minimum checkpoint size. Importantly, for both AR and DS, all suggestions were *actionable* because they concretely described where to move each task boundary. For MI, there were only two suggestions, with a maximum potential savings of 2 bytes to checkpoint, or 7% the minimum checkpoint size. MI's suggestions were less useful than for the other programs' because MI uses less stack data than DS and has a shallower call tree than both AR and DS, meaning less potential stack savings.

8. Related Work

Consistency despite failures is a cornerstone of reliable computing, but most work assumes continuous power, leaving consistency in intermittent systems unstudied.

Saving state on machines that are *not* fundamentally failure prone is the focus of Whole-System Persistence [24], which uses capacitance-backed volatile memory to provide flush-on-fail semantics to an entire operating system. DINO provides persistence for arbitrary data on intermittent embedded platforms without additional fault-tolerance hardware.

QuickRecall [16] and Mementos [28] aim to make computational RFIDs (like the energy-harvesting WISP [31]) viable for general-purpose applications. Both dynamically checkpoint volatile data to nonvolatile memory, preserving it across failures. *Unlike DINO, neither system addresses inconsistencies in nonvolatile memory.* Also, both dynamically decide when to checkpoint, heuristically, at loop backedges and function returns. By contrast, DINO checkpoints at each programmer-defined task boundary. DINO has fewer failure recovery targets (§7.3) and they are explicit in the code.

Idetic [23] checkpoints state in ASICs that operate intermittently. It optimizes checkpoint placement and frequency of checkpointing at run time based on energy conditions. Idetic selectively checkpoints volatile data, whereas DINO selectively versions nonvolatile data. Like DINO, Idetic computes the cost of inserted checkpoint calls. Unlike DINO, Idetic does not target programmable platforms, does not consider nonvolatile state consistency, and suffers high control-flow complexity from dynamic checkpointing decisions.

Other systems have proposed accessible persistence mechanisms for nonvolatile memories, but none target embedded or intermittent systems, and some require hardware support. Mnemosyne [37] is an interface to persistent storage and heap data. It provides persistence keywords for variables and a persistent `mmap` that treats persistent memory as a block device. For consistency, Mnemosyne uses fences that strictly order memory, as well as transactional semantics that use hardware transactional memory. DINO targets intermittent embedded devices and must operate without OS or hardware support. Intermittence makes failures more common than Mnemosyne was designed to face efficiently. Additionally, DINO exposes a load–store interface to persistent memory rather than exposing it as block storage.

Memory persistency [26] characterizes memory consistency for persistent memory, focusing on allowable consistency relaxations and correctness despite explicit concurrency and failures. Its goals are complementary to DINO's, but persistency optimizations under intermittence are a second-order concern for DINO which we leave to future work. DINO adds another dimension to memory persistency's notions of *persist epochs* and *persist barriers*, which map well to DINO's tasks and boundaries: we consider *implicit* concurrency and frequent failures.

PMFS [12] adds a similar persistent-memory (PM) write barrier and implements a filesystem for block PM. Like persistency, PMFS's main consistency goal is to order concurrent writes reasonably—orthogonal to our consistency goal of constraining program states to those reachable under continuous execution.

Persistency and PMFS owe their heritage to BPFS [9], which addressed atomicity problems for filesystems on nonvolatile memory. BPFS proposed to add capacitance to DIMMs and write redundantly to ensure the integrity of writes, under the assumption that failures were relatively rare. In contrast, DINO embeds consistency mechanisms in programs and expects these mechanisms to be triggered frequently.

NV-Heaps [8] considers the safety of memory-mapped nonvolatile heaps, focusing on pointer integrity, and provides a programming construct for nonvolatile transactions based on BPFS's architectural support. DINO's task atomicity could be implemented atop NV-Heaps' *generational locks*, but DINO's operation is otherwise fundamentally different from heap management, especially since embedded systems generally avoid dynamic allocation. Bridging this gap may be profitable in future work.

Kiln [41] adds nonvolatile cache to the memory hierarchy to speed up persistent memory with in-place updates to large data structures that persist in cache until they are flushed to backing storage. DINO uses comparatively simple transactional mechanisms (tasks) built into programs, not the memory hierarchy, and can therefore analyze programs to determine how to apply versioning to nonvolatile data without hardware modifications.

Venkataraman et al. describe NVRAM consistency challenges from a data structures perspective and design *consistent and durable data structures* (CDDSs) to recover safely from aborted updates to in-NVRAM data structures [35]. They adopt a *data versioning* approach based on generational numbering. In contrast, DINO applies versioning to data of any kind in a program, rather than designing new data structures, and DINO uses static analysis to automatically identify things that need to be versioned.

DINO traces some of its lineage to the literature on *orthogonal persistence* (OP) [3], a form of persistence that is meant to be transparent to applications. While one of DINO's goals is to make run-time failures harmless, it requires the programmer to identify points where consistency is most important, placing DINO's form of persistence somewhere between that of orthogonal and non-orthogonal persistence.

DINO does not support concurrency, a simplification that allows it to sidestep the challenges that concurrency support poses for orthogonal persistence [2, 5]. Concurrent applications are perhaps a poor match for intermittently powered systems that resemble today's single-programmed embedded systems.

9. Conclusions and Future Work

Applications that work on continuously powered devices may exhibit surprising failure modes when run on intermittent power because of inconsistencies in nonvolatile memory. As energy harvesting becomes a viable way to power computing platforms with next-generation nonvolatile memory, ensuring data consistency under intermittent power will be crucial. DINO guarantees data consistency, constraining execution to states reachable in continuous execution even when running intermittently. DINO simplifies programming intermittently powered applications by reducing the complexity of reasoning about failures, paving the way for low-power platforms.

DINO lays the basic groundwork that can serve as a basis for future work on a programming and system stack for intermittently executing devices. Future efforts should focus on providing stronger correctness guarantees in applications with mixtures of idempotent and non-idempotent code. Future systems may also benefit from selectively allowing data inconsistency (e.g., by eliding task boundaries) in exchange for reduced run-time overhead. Such future research into DINO-like programming, system, and architecture support for simplifying intermittent devices is the key to bringing their full potential to bear for future applications.

Acknowledgments

We thank our anonymous reviewers for their insightful and supportive comments. Shaz Qadeer and Adrian Sampson provided valuable feedback on early drafts, as did members of the Sampa group at the University of Washington. This work was supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8), Aug. 2010. doi: http://dx.doi.org/10.1145/1787234.1787255.

[2] M. Atkinson and M. Jordan. Issues raised by three years of developing PJama: An orthogonally persistent platform for Java. In *Intl. Conf. Database Theory (ICDT)*, Jan. 1999.

[3] M. Atkinson and R. Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3), July 1995.

[4] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2011.

[5] S. Blackburn and J. N. Zigman. Concurrency – the fly in the ointment? In *3rd Intl. Workshop on Persistence and Java (PJW3)*, Sept. 1999.

[6] M. Buettner, B. Greenstein, and D. Wetherall. Dewdrop: An energy-aware task scheduler for computational RFID. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2011.

[7] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *ISCA*, June 2007.

[8] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, Mar. 2011. doi: http://dx.doi.org/10.1145/1950365.1950380.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Symposium on Operating Systems Principles (SOSP)*, Oct. 2009. doi: http://dx.doi.org/10.1145/1629575.1629589.

[10] A. Czeskis, K. Koscher, J. R. Smith, and T. Kohno. RFIDs and secret handshakes: defending against ghost-and-leech attacks and unauthorized reads with context-aware communications. In *ACM Conference on Computer and Communications Security (CCS)*, Oct. 2008. doi: http://dx.doi.org/10.1145/1455770.1455831.

[11] J. Dai, X. Bai, Z. Yang, Z. Shen, and D. Xuan. Mobile phone-based pervasive fall detection. *Personal Ubiquitous Computing*, 14(7), Oct. 2010. doi: http://dx.doi.org/10.1007/s00779-010-0292-x.

[12] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys '14*, Apr. 2014. doi: http://doi.acm.org/10.1145/2592798.2592814.

[13] C. Ferri, A. Viescas, T. Moreshet, I. Bahar, and M. Herlihy. Energy implications of transactional memory for embedded architectures. In *Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods (EPHAM'08)*, Apr. 2008.

[14] S. Gollakota, M. S. Reynolds, J. R. Smith, and D. J. Wetherall. The emergence of RF-powered computing. *Computer*, 47(1), 2014. doi: http://dx.doi.org/10.1109/MC.2013.404.

[15] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, June 2004.

[16] H. Jayakumar, A. Raha, and V. Raghunathan. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *Int'l Conf. on VLSI Design and Int'l Conf. on Embedded Systems*, Jan. 2014.

[17] A. Kansal, S. Saponas, A. B. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola. The Latency, Accuracy, and Battery (LAB) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing. In *OOPSLA*, Oct. 2013. doi: http://dx.doi.org/10.1145/2509136.2509541.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), July 1978. doi: http://dx.doi.org/10.1145/359545.359563.

[19] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, Mar. 2004.

[20] R. LeMoyne, T. Mastroianni, M. Cozza, C. Coroian, and W. Grundfest. Implementation of an iPhone for characterizing Parkinson's disease tremor through a wireless accelerometer application. In *Int'l Conf. IEEE Engineering in Medicine and Biology Society (EMBC)*, Aug. 2010. doi: http://dx.doi.org/10.1109/IEMBS.2010.5627240.

[21] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, Jan. 2005. doi: http://dx.doi.org/10.1145/1040305.1040336.

[22] MIDI Manuf. Assoc. Summary of MIDI messages. `http://www.midi.org/techspecs/midimessages.php`, 2014. Visited August 3, 2014.

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

[24] D. Narayanan and O. Hodson. Whole-system persistence with non-volatile memories. In *ASPLOS*, Mar. 2012.

[25] J. A. Paradiso and T. Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, 4(1):18–27, 2005. doi: http://dx.doi.org/10.1109/MPRV.2005.9.

[26] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *ISCA*, June 2014.

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

[28] B. Ransford, J. Sorber, and K. Fu. Mementos: System support for long-running computation on RFID-scale devices. In *ASPLOS*, Mar. 2011.

[29] L. Ren, Q. Zhang, and W. Shi. Low-power fall detection in home-based environments. In *ACM International Workshop on Pervasive Wireless Healthcare (MobileHealth)*, June 2012. doi: http://dx.doi.org/10.1145/2248341.2248349.

[30] M. Salajegheh, S. S. Clark, B. Ransford, K. Fu, and A. Juels. CCCP: secure remote storage for computational RFIDs. In *USENIX Security Symposium*, Aug. 2009. URL `https://spqr.eecs.umich.edu/papers/salajegheh-CCCP-usenix09.pdf`.

[31] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith. Design of an RFID-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement*, 57(11):2608–2615, Nov. 2008.

[32] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmers model for x86 multiprocessors. *Commun. ACM (Research Highlights)*, 53(7), July 2010. doi: http://dx.doi.org/10.1145/1785414.1785443.

[33] Texas Instruments Inc. MSP430 flash memory characteristics. `http://www.ti.com/lit/an/slaa334a/slaa334a.pdf`, Apr. 2008. Visited August 5, 2014.

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

[35] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, Feb. 2011. URL `https://www.usenix.org/legacy/events/fast11/tech/full_papers/Venkataraman.pdf`.

[36] N. Villar and S. Hodges. The Peppermill: A human-powered user interface device. In *Conference on Tangible, Embedded, and Embodied Interaction (TEI)*, Jan. 2010. doi: http://dx.doi.org/10.1145/1709886.1709893.

[37] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *ASPLOS*, Mar. 2011. doi: http://dx.doi.org/10.1145/1950365.1950379.

[38] D. Yeager, P. Powledge, R. Prasad, D. Wetherall, and J. Smith. Wirelessly-charged UHF tags for sensor data collection. In *IEEE Int'l Conference on RFID*, Apr. 2008.

[39] L. Yerva, B. Campbell, A. Bansal, T. Schmid, and P. Dutta. Grafting energy-harvesting leaves onto the sensornet tree. In *Conference on Information Processing in Sensor Networks (IPSN)*, Apr. 2012. doi: http://dx.doi.org/10.1145/2185677.2185733.

[40] P. Zhang, D. Ganesan, and B. Lu. QuarkOS: Pushing the operating limits of micro-powered sensors. In *HotOS*, May 2013.

[41] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *MICRO*, Dec. 2013.