Time-sensitive intermittent computing meets legacy software

Kortbeek, Vito; Yildirim, Kasim Sinan; Bakar, Abu; Sorber, Jacob; Hester, Josiah; Pawelczak, Przemyslaw

DOI
10.1145/3373376.3378476

Publication date
2020

Document Version
Final published version

Published in
ASPLOS 2020 - 25th International Conference on Architectural Support for Programming Languages and Operating Systems

Time-sensitive Intermittent Computing Meets Legacy Software

Vito Kortbeek
Delft University of Technology
Delft, The Netherlands
v.kortbeek-1@tudelft.nl

Kasım Sinan Yıldırım*
University of Trento
Trento, Italy
kasimsinan.yildirim@unitn.it

Abu Bakar
Northwestern University
Evanston, IL, USA
abubakar@u.northwestern.edu

Jacob Sorber
Clemson University
Clemson, SC, USA
jsorber@clemson.edu

Josiah Hester
Northwestern University
Evanston, IL, USA
josiah@northwestern.edu

Przemysław Pawełczak
Delft University of Technology
Delft, The Netherlands
p.pawelczak@tudelft.nl

Abstract

Tiny energy harvesting sensors that operate intermittently, without batteries, have become an increasingly appealing way to gather data in hard to reach places at low cost. Frequent power failures make forward progress, data preservation and consistency, and timely operation challenging. Unfortunately, state-of-the-art systems ask the programmer to solve these challenges, and have high memory overhead, lack critical programming features like pointers and recursion, and are only dimly aware of the passing of time and its effect on application quality. We present Time-sensitive Intermittent Computing System (TICS), a new platform for intermittent computing, which provides simple programming abstractions for handling the passing of time through intermittent failures, and uses this to make decisions about when data can be used or thrown away. Moreover, TICS provides predictable checkpoint sizes by keeping checkpoint and restore times small and reduces the cognitive burden of rewriting embedded code for intermittency without limiting expressibility or language functionality, enabling numerous existing embedded applications to run intermittently.

CCS Concepts. • **Computer systems organization** → **Embedded software**; • **Hardware** → *Emerging architectures*; *Impact on the environment*; • **Software and its engineering** → **Runtime environments**; **Source code generation**.

*Also with Ege University, İzmir, Turkey.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378476

Keywords. Legacy Code, Compiler, Source Transformation, Runtime, Energy Harvesting, Intermittent, Batteryless

ACM Reference Format:
Vito Kortbeek, Kasım Sinan Yıldırım, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. 2020. Time-sensitive Intermittent Computing Meets Legacy Software. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378476

1 Introduction

Tiny embedded computing systems and sensor networks have created a revolution [41]—changing how we monitor buildings and other infrastructure, treat disease, and protect endangered wildlife—but, the decades-old vision of ubiquitous computing (and now the Internet of Things [4]) are frustrated by energy storage issues. Today, most untethered devices rely on batteries—fragile, short-lived, bulky, relatively expensive chemical energy stores [24, 35]. Enabled by improvements in energy harvesting technologies and low-power circuit design, as well as the commercialization of byte-addressable non-volatile memories (like FRAM, MRAM, and ReRAM [8]), batteryless devices with minimal energy storage that run solely off ambient scavenged energy, promise a more scalable and sustainable alternative [19, 40, 46, 47].

Reducing energy storage to near-zero comes with consequences across the stack, from the architecture to the programmer and user [27]. As energy harvesting conditions fluctuate, power failures can occur frequently—as often as many times per second [43, Fig. 1], [39, Fig. 1]. Power failures clear the volatile state of the processor; i.e. call stack, program counter, heap and volatile registers, making it difficult to ensure *forward progress* (see Figure 1) and in some cases endangering *memory consistency* [38]. As power outages increase, data gathered and processed before a power failure may no longer be relevant when a device turns back on [20, 46], as the length of power failures can vary significantly. Punctual computing is difficult when operation is intermittent and clocks are imprecise. These issues have

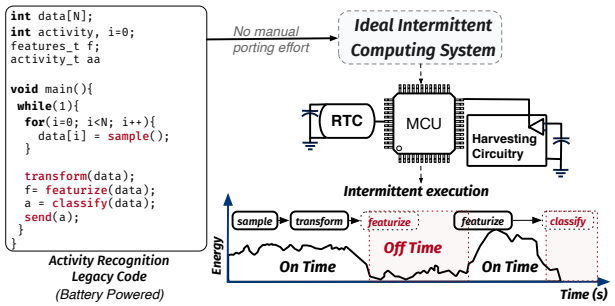

Figure 1. An ideal programming model for intermittent computing systems should remove the cognitive burden of porting legacy software.

motivated numerous approaches for building power failure resilient programs [27].

Today's intermittent computing systems are either: (i) traditional standard C programs paired with automatic or manual *checkpointing*, e.g. [28, 39, 45], where the volatile state of the processor is logged into non-volatile memory and upon recovery from a power failure the computation is continued from the latest logged volatile state, and (ii) *task-based programs*, e.g. [6, 23, 31, 46], where developers break programs into idempotent and atomic tasks (that can be restarted when interrupted by a power outage) and describe the control and data flow between tasks.

Unfortunately, with the aforementioned approaches, porting the massive set of legacy software that runs on continuous power to work with intermittent power, e.g. TinyOS [25] programs from the past two decades of wireless sensor network deployments, is cumbersome and requires massive re-engineering. First, task-based programming requires significant developer effort to transform a program to fit the programming model [10]—developers are forced to decompose logical operations (e.g. "classify activity") into multiple sub-tasks which are executable with the available energy storage. This procedure is difficult for developers, especially non-experienced ones—porting old code (or developing new code) becomes time-consuming and applications become incomprehensible and hard to debug due to task count explosion. Second, task-based programming does not allow common programming language constructs, such as pointers since it enforces a static memory model [46]—making automatic (even non-automatic) transformation of legacy code into task-based model impossible. On the other hand, checkpointing systems remove the cognitive burden of porting, but have high memory overhead and performance penalties due to large checkpoints [9, 45]. Moreover, these systems cannot execute all C-programs: in particular, pointers and recursion might lead to checkpoints that will not fit into the available energy storage and prevent the progress of computation [10]— causing a *system starvation*. Even worse, checkpointing systems do not allow semantics to handle *elapsed time* and in turn they cannot handle time-sensitive data that might be expired after a long power failure. Developers have no way to easily inject decision points into legacy software based on the time elapsed since failure can occur in-between any lines of the code.

These issues beg the question: is there a way to bridge the gap between time-sensitive intermittent computing and legacy software designed for continuously-powered systems? As of now we are still far from an *ideal* intermittent computing system that (i) removes the cognitive burden of porting legacy software and enables unaltered C programs (with standard programming constructs and any typical compiler optimizations enabled) to be executed on intermittent power; (ii) provides semantic and syntactic mechanisms to handle data freshness (and passing of time in general) for *timely execution* of the application; and (iii) introduces low memory impact and little performance penalty. These requirements are necessary to enable widespread adoption of intermittent computing.

In this paper, we propose **TICS** (Time-sensitive Intermittent Computing System), a new intermittent computing system designed with the goal of running time-sensitive code on intermittent platforms via *automatic* checkpoints. TICS enables programmers to (i) execute any kind of *unaltered C program* (including pointers and recursion) by greatly reducing, as well as bounding, the overhead of checkpoint/restore times—eliminating system starvation, and (ii) optionally annotate the program with structures to specify custom *timing requirements*—protecting against timing errors that are never seen in continuously-powered programs. The core scientific contributions of this work are:

- *Time sensitivity semantics* for checkpoint-based intermittent systems—enabling, for the first time, declarative annotations for intermittent applications to handle the passing of time in-between power failures and to eliminate *time consistency violations* particular to intermittent systems;
- *Memory consistency management* for checkpoint-based intermittent systems by combining *data versioning* and *stack segmentation* to bound checkpoint/restore times—enabling, for the first time, execution of unaltered C-programs–including pointers and recursion–*without system starvation* and endangering memory consistency, and providing foundation for memory isolation, I/O access and interrupt handling;
- An open source portable runtime for developers (released via [2])—enabling, for the first time, widespread adoption of intermittent computing by allowing *porting* of several unmodified, as well time annotated legacy code, to the intermittent computing platforms.

2 Battery-free Intermittent Systems

The community around wireless sensor networks has worked together to enable long-term, affordable, sustainable sensing across many application domains, ranging from

wildlife tracking [15, 42, 48] to infrastructure monitoring [7, 14], health and human sensing [17, 26, 34], autonomous vehicles [13], and even space exploration [33]. Recently, wireless sensing devices that work without batteries have become viable [18, 40] due to advances in miniaturization of energy harvesters for solar, RF, and kinetic energy, and energy-efficient microcontrollers with non-volatile memory [44]. Making these devices work is challenging, because of power failures caused by unreliable and sporadic energy sources as shown in Figure 1. However, the benefits are well worth it—besides reduced cost, size, and weight, these devices promise decade long lifetimes in the field without requiring maintenance or replacement [19].

Batteryless platforms like WISP [40], Capybara [11], and Flicker [18] use ultra-low-power micro-controllers (MCUs); e.g. MSP430FR* [44], whose main architectural components; e.g. registers and main memory, are *volatile*, while the code and data retained from power failure to power failure are held in non-volatile byte addressable RAM (e.g. FRAM). These platforms harvest and then store energy in small capacitors for all tasks. When the energy stored is depleted, the device dies—computation cannot progress and output correct results. The naive approach to solve this would be to save the entire volatile memory and registers to non-volatile right before a power failure, but this has high memory and performance penalty, requires reliable brownout detection and does not solve consistency issues described in prior work [28]. Instead, smaller checkpoints at compiler or programmer-defined boundaries like in Figure 1 reduce wasted computation and allow programs to complete. Many systems have been explored [5, 6, 23, 28, 32, 39, 45] which checkpoint at statically defined or dynamically decided points. Of course, the main challenge is to determine *what* to backup and *when*. The alternative is to change the programming model completely, asking programmers to decompose a C program into atomic and idempotent sections called *tasks*[1] with defined control and data input and output variables that are shared by tasks connected in a *task graph*. Only the state of the active task is backed up at each task transition to ensure forward progress [9, 20, 31, 46].

2.1 The Big Jump to Legacy Software

In a perfect world, the developer of a battery-powered platform would just recompile the application for intermittent-power. However, so far no existing systems enable this.

2.1.1 Task Decomposition. There is a massive set of existing applications and legacy code written e.g. TinyOS [25] or Contiki [12] in the past two decades of wireless sensor network deployment. Unfortunately, the existing corpus of applications is difficult, or impracticable, to port to work with

[1]It should be noted that tasks in intermittent computing must not be confused with the tasks in TinyOS [25]. Tasks of TinyOS define self-contained *work to be undertaken*, while tasks in intermittent domain define a piece of code that is *atomic and idempotent* [9].

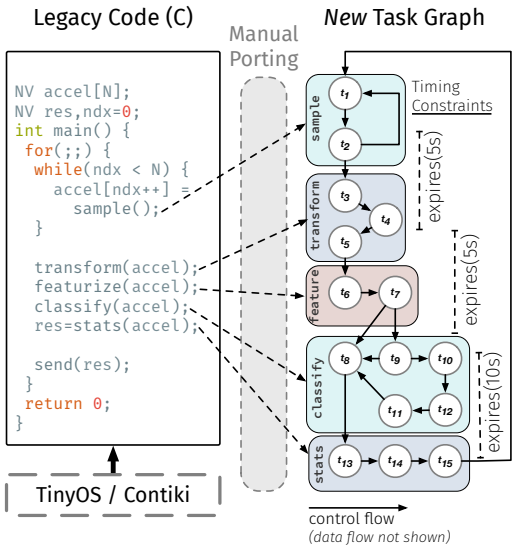

Figure 2. Porting legacy code to a task graph is not trivial. *Left*: an activity recognition program, *Right*: one possible port to a task graph, with timing constraints defined. The final task graph does not always match the original program as tasks must be small enough to allow forward progress; reducing readability, and increasing graph complexity. Timing constraints, control flow, and data flow must also be determined.

battery-free platforms and applications—*especially* when porting to a task-based programming model. Figure 2 illustrates these challenges. Task decomposition is hard [10], as task boundaries are not defined by logical breaks in a program (such as those shown on the left part of Figure 2: `classify()`, `transform()`, etc.) but are best set based on the energy cost of blocks of code, and/or data interconnections between code. Moreover, task-based programming models are *static* in their memory model, as they allow data exchange among the tasks by using only data interconnections to ensure memory consistency. This prohibits pointers. That is, porting programs with pointers is difficult (or sometimes even impossible).

2.1.2 Automatic Checkpointing. Checkpointing systems do not occupy the programmer with the decomposition of legacy code into intermittent tasks, and there is no need to re-implement original libraries. However existing checkpointing systems sacrifice memory overhead, and language features to allow for automatic insertion of checkpoints [32, 45]. Current practice is to over-instrument a program with potential places to checkpoint, and these are triggered based on some signal, e.g. low energy interrupt. Storing copies of the stack and global data in multiple versions of checkpoints, as well as this over-instrumentation, leads to *high memory impact* for resource-constrained MCUs [28]. What is worse, since dynamic memory manipulations via pointers cannot be

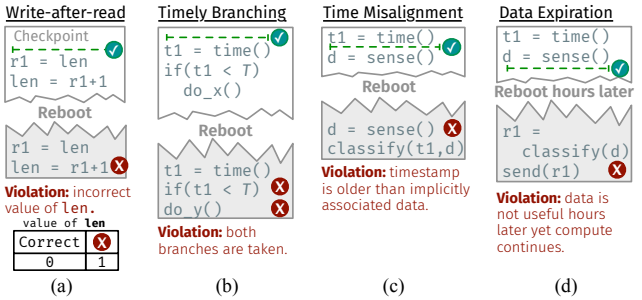

Figure 3. Four types of consistency violations encountered with automatic checkpointing. These violations occur because of incorrect execution caused by bad checkpoint placement, leading to an execution that is not possible on a continuously powered device. With this work we introduce a new class of violations, i.e. *time-based violations*, that have not been previously explored in checkpointing systems—refer to figures (b)–(d).

determined at compile time, the whole main memory should be checkpointed after each pointer manipulation—the checkpointed state grows with the size of the main memory and unfortunately leads to a *system starvation* since it might not fit into the device's energy reservoir. Supporting full C language functionality is non-trivial and crucial to port/reuse legacy software.

2.1.3 Time Consistency. Consistency violations identified in previous work [28] include only memory consistency violation; see Figure 3(a): after a checkpoint, non-volatile global variable `len` is changed, but these actions are not included in the checkpoint. When the checkpoint is restored, `len` is again updated, leading to an incorrect value of `len` due to the Write-After-Read (WAR) dependency. We identify *three other types of consistency violations*, all having to do with time. The errors stem from the fact that clocks internal to the MCU are reset after each power failure, meaning that devices have difficulty tracking how long they have been off [21, 37]; even when using external timekeepers, time-sensitive portions of a program must be handled differently in checkpointing systems by careful checkpoint placement or time management.

1. **Timely Branching.** If a checkpoint is placed in a line of code before a timestamp is gathered, and that timestamp is used in a predicate statement, execution can execute both branches if the timestamp elapses; see Figure 3(b);
2. **Time and Data Misalignment.** Often in embedded programs, a timestamp is gathered every time sensor data is obtained. If a checkpoint is placed between the timestamp and the data gathering, the timestamp will be inaccurate. After a power failure recovery at that checkpoint, *new* data will be gathered associated with an *old* timestamp—causing incorrect execution of the program; see Figure 3(c);
3. **Data Expiration.** Data gathered in one power cycle may not be fresh enough for the next power cycle. This phenomenon [20] has not been handled by any automatic checkpointing systems to date; see Figure 3(d).

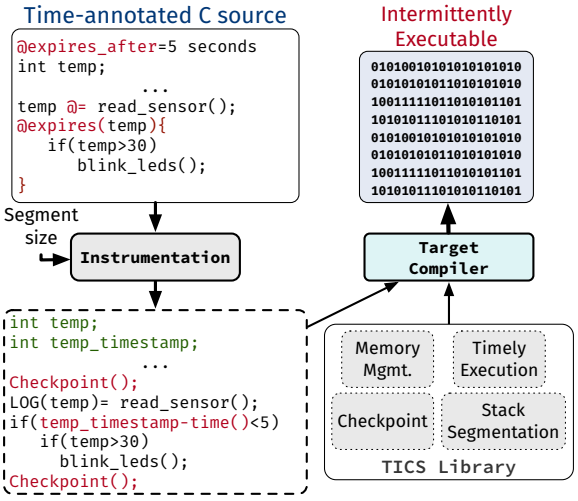

Figure 4. TICS overview: A runtime combined with code instrumentation ensures memory consistency via *data versioning* and *stack segmentation*; progress of computation via *checkpointing*; and timely execution via *time annotations*.

3 TICS: System Design

TICS consists of a runtime combined with code instrumentation for the C language—Figure 4 presents the logical flow and the main components of the TICS system. The main motivation behind TICS is to provide the view of a continuously-powered system to the programmer—so that legacy C code can be run without any modification to the program source. TICS allows the programmer (i) to focus on the correct and timely execution of the application—eliminating the explicit need for intermittency handling, and (ii) to perform few modifications to the original program, specifically, to annotate their code *only* to define timing constraints.

Task-based versus Checkpointing. Conversion of a C program into a task-based program requires significant manual labor (as discussed in Section 2); automatic transformation of a pointer-based C program is incredibly difficult due to memory burden created by a multitude of versions of memory locations/variables. Therefore, instead of task-based transformation, TICS uses checkpointing in order to get rid of manual code transformation and its limitations.

Building an Efficient Stack. As the amount of state that is checkpointed grows, the checkpoint overhead increases, potentially leading to overheads that may exceed the device's capabilities and energy budget. Since functions often manipulate local variables in their stack frame, there is no need to

checkpoint the whole stack. TICS employs a novel strategy by segmenting the stack into fixed and predetermined size blocks. The stack segment that is directly manipulated at a time instant by the program is called the *working stack* and it will be the only one among others that needs to be logged into a *segment checkpoint*—since other segments are not modified. By segmenting the stack TICS can provide a fixed worst-case checkpoint time, as the variable stack size is fixed to the size of a stack segment. It is worth mentioning that the programmer is completely unaware of the underlying stack segmentation but the desired size of stack segments can be chosen at compile time for the sake of performance—see Section 5.

Pointer Handling. As pointer access cannot be determined at compile time, existing systems need to checkpoint the whole main memory in order to keep memory consistent—leading to huge checkpoints and in turn *system starvation* due to limited energy reservoir. TICS implements a data versioning scheme to handle pointers and ensure memory consistency: it keeps track of only manipulated memory locations by keeping the original values in a non-volatile *undo log*. The undo log is cleared upon a successful checkpoint, otherwise TICS restores the original contents of the memory using the undo log—ensuring the memory consistency despite power failures.

Memory Impact. Checkpointing the device's volatile state requires an atomic *two-phase commit* operation to ensure its consistency [31]: in the first phase the checkpointed data is copied to a temporary buffer in non-volatile memory; then in the second phase the buffered data is committed to the original location. Existing checkpointing systems double buffer the stack, `.bss` and `.data` sections—their memory requirements increase with the volatile state. On the other hand, TICS only requires the segment checkpoint and the modified memory locations in `.bss` and `.data` sections to be double buffered—significantly reducing the memory impact.

Timely Execution. C does not provide any keyword/statement to express time constraints of the data and handle it—programmers must explicitly timestamp data and handle data expiration. This complicates application development as well as might lead to bugs due to manual timing and expiration checks, control flow delivery and recovery due to data expiration—as given in Section 2.1.3. TICS provides annotations to relate data and time as well as special statements to change control flow and perform recovery upon data expiration—all underlying time management is performed at run-time without programmer intervention.

3.1 Efficient Automatic Checkpoints

Existing works, e.g. [6, 22, 45] exploit architectural support and ensure constant and scalable checkpointing overhead. For example, Ratchet [45] uses non-volatile memory as the main memory so that stack and global variables are already persistent—leading to constant checkpoint time since only the volatile registers of the processor are checkpointed. This requires decomposing programs into idempotent code sections via the compiler using *static analysis* at the instruction level and gluing them together with checkpoints. However, dynamic memory manipulations that cannot be determined at compile time, e.g. write operations via pointers, require a checkpoint after each instruction, leading to a considerable checkpointing frequency and, in turn, overhead. TICS targets devices with non-volatile main memory—a checkpoint operation logs *only* the registers and the stack in a dedicated double-buffered area in non-volatile memory via a two-phase commit. Since stack grows/shrinks dynamically, checkpointing overhead grows with the size of the stack. Moreover, recovery time, i.e. restoring the state after a power failure, is not fixed and might exceed the device's energy budget—leading to *system starvation*. TICS remedies this with *stack segmentation* and *data versioning*.

3.1.1 Stack Segmentation. The stack allocation within the execution of the applications might vary significantly, in particular when a lot of memory space is allocated/deallocated at function entries/returns. The stack size requirement depends on dynamic program flow (that might be unknown at compile time) and in turn, it is not possible to guarantee a worst-case checkpoint size. TICS segments the stack into blocks of fixed size selected at compile time—maximum stack frame in a program (determined during compilation) dictates the minimum block size. TICS maintains the segmented stack of a program as a *segment array* in non-volatile memory—see Figure 5. The size of the stack array is fixed at compile time by considering the stack requirements and exceeding the size at runtime leads to a stack overflow. The program interfaces with the top segment of the segmented stack; so-called the *working stack*: the program modifies only the working stack, and upon a checkpoint, only the working stack is two-phase committed into the double-buffered *segment checkpoint*—this enables a fixed checkpoint time. Moreover, recovering from a power failure only requires the working stack to be restored from the segment checkpoint, instead of restoring the whole stack.[2]

During program execution, the stack grows/shrinks make the working stack point to different segments in the segment array. When a function is entered, the stack pointer is adjusted: TICS inserts a check before the modification of the stack pointer to determine whether there is enough space in the working stack to execute the function. When enough space is in the working stack, the execution resumes and the function interfaces with the working stack. Contrary, if there is not enough space left on the working stack, a *stack grow procedure* is initiated so that the working stack points

[2]Differential checkpoints [3] log only modified part of the stack—but they can still be large for nested function calls each using a lot of stack.

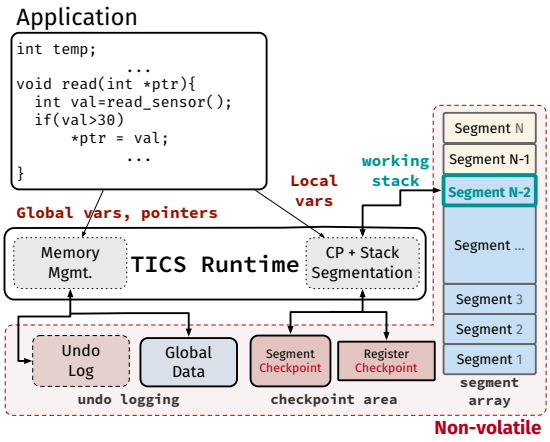

Figure 5. TICS architecture. With TICS only the working stack and the registers during a checkpoint are logged—ensuring deterministic worst-case overhead. Previously checkpointed segments belonging to the lower parts of the stack are maintained in a segment array. The global variable and pointer access are handled by the memory manager which implements undo logging to keep memory consistent.

to the next segment in the segment array. It is worth mentioning that a checkpoint after this point requires only the new working stack to be saved into the segment checkpoint since the previous segments remain unmodified. When a function that triggered a stack grow returns, a *stack shrink* is initiated so that the working stack points to the previous segment in the segment array. TICS can enforce an implicit checkpoint if the current working stack was not saved into the segment checkpoint yet. This is because if the currently checkpointed segment is out of program stack, the working stack should be checkpointed first so that the modifications can be rolled back upon power failures and the stack consistency is ensured.

It is worth mentioning that no special attention is needed when TICS executes *recursive functions*. However, as in general embedded systems, the depth of the recursive calls is limited by the size of the stack memory, which is represented by the fixed size of the segment array in TICS architecture.

3.1.2 Memory Management and Pointers. TICS maintains global variables; i.e. `.data` and `.bss` sections in non-volatile memory. Intermittent execution might create inconsistencies if the application modifies non-volatile memory directly and the modified locations are not versioned, i.e. double buffered [28, 38]. TICS instruments non-volatile memory write operations and enables on-demand versioning: *undo logging* is employed so that if any memory location outside of the working stack has been modified, the original version is saved in an undo log. After a successful checkpoint, the undo log is cleared. Upon power failure, the contents of the undo log are written back to the original locations. Since the undo log is also fixed in size, TICS forces a checkpoint when

```
@expires_after=1s /* data expires in 1 second */
int temperature[WINDOW_SIZE];
...
/* data & timestamp alignment (assign timestamp)
    */
temperature[i] @= read_sensor();
...
/* catch data expiration */
@expires(temperature[i]){
  if(temperature[i] > max) {max = temperature[i
      ]};
}
...
/* branch in time (before the send deadline) */
@timely(SEND_DEADLINE){ send(max); } else {...}
...
```

Figure 6. An overview of TICS annotations for timely execution of intermittent applications. TICS supports timely branches, ensures data and time alignment and catches data expiration.

the undo log is full to eliminate the overflow and ensure forward progress.

In TICS, pointer writes to global variables within the `.data` and `.bss` sections in non-volatile memory are managed at runtime. Additionally, *pointers* to the stack can manipulate memory locations, in particular, stack segments other than the working stack. A pointer to the working stack can directly modify its contents since the working stack is checkpointed separately. Conversely, if it points to other segments in the segment array or global variables in `.data` and `.bss`, the memory manager employs undo logging.

3.2 Semantics for Timely Execution

TICS provides annotations; i.e. `@expires_after`, to denote the expiration constraints of the data and necessary keywords for checking if time constraints are met—see Section 6. A timestamp value is associated with each programmer annotated variable and the write operations on these variables are instrumented by the compiler. TICS can update the value of the timestamp automatically upon writes using a persistent timekeeper which keeps track of time across power failures [20]—see Section 4 for details. Programmers can check the expiration of the data using `@expires` block—TICS compares the current time with the timestamp to identify if programmer-defined timing constraints are met. Programmers can also use `@expires_after=0s` statement for any variable that requires a timestamp associated with it but does not have any expiration constraint. It is the responsibility of the programmer to provide necessary logic within these syntactic structures.

3.2.1 Supporting Timely Branches. In order to prevent timely branch violations as depicted in Figure 3(b), TICS introduces `@timely/else` block that takes a time value as an input. This block disables automatic checkpoints, reads the *current time* using the (persistent) timekeeper and checks if the given time value is greater than the current time. If this is the case, the branch is taken, a checkpoint is placed

at the end of the branch and automatic checkpoints are enabled. Otherwise, the branch is not taken and automatic checkpoints are enabled.

3.2.2 Ensuring Data and Time Alignment. As depicted in Figure 3(c), if a checkpoint is placed between the timestamp assignment and the data gathering (or vice versa), the timestamp can be inaccurate after a power failure. In particular, this issue is problematic if checkpoints are done automatically, e.g. with a periodic timer. To remedy this, timestamp assignment and data gathering operations should form an atomic block. TICS ensures the atomicity by (i) disabling automatic checkpoints so that timestamp assignment and data gathering cannot be split; and (ii) placing a checkpoint right after these operations (and enabling automatic checkpoints thereafter, if needed) so that the consistency of timestamp and data is guaranteed despite a power failure.

TICS introduces operator `@=` for the atomic assignment of the data and timestamp—see Figure 4. TICS makes this assignment explicit via `@=` since there is no need to update the timestamp of the associated data per each write, e.g. the sensed temperature value can be converted from the raw ADC value to the degree in Celsius and this conversion should not lead to the update of the associated timestamp.

3.2.3 Catching Data Expiration. In TICS, `@expires` and `@expires/catch` blocks are used to work with the data within a certain time frame and to catch data expiration—Figure 3(d). For the sake of implementation simplicity, we remark that these blocks consider only one variable.

Conditional-based @expires. TICS implements `@expires` block by using an `if` statement at the beginning that checks if the data is still valid; see Figure 4. If the condition is met, the rest of the operations will be executed within this block. Due to automatically-inserted checkpoints and arbitrary power failures, `@expires` block might not be atomic. If a checkpoint is placed inside an `@expires` block, a power failure might lead to data expiration—TICS disables automatic checkpoints at the beginning of the `@expires` block so that computation starts from the `if` statement after each power failure. TICS places a checkpoint at the end of `@expires` block and enables automatic checkpoints thereafter. It is worth mentioning that these operations ensure the atomicity, but data can still expire since the instructions within the `@expires` block can be long enough to violate data freshness constraints.

Exception-based @expires/catch. In order to catch data expiration while executing an `@expires` block, TICS sets a timer at the beginning that fires when the data expires. Upon timer fire and in turn data expiration, TICS restores the original contents of the modified variables inside the `@expires` block by using the original values in undo log. TICS delivers the control flow to the `catch` block that handles specific logic to handle data expiration. Since undo logging is required for exception-based implementation, its implementation is parallel to the rest of TICS for the sake of memory consistency.

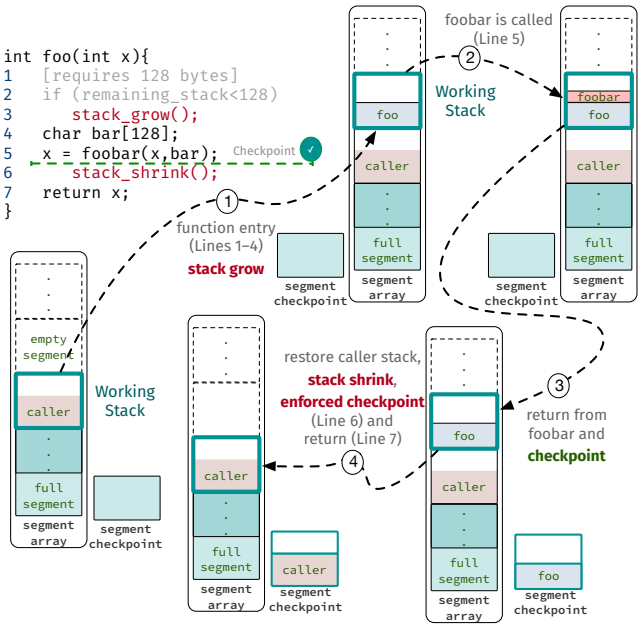

Figure 7. TICS stack segmentation and checkpointing. In pseudocode: lines in light gray and red colors represent the code inserted during the compiler pass; the red lines only execute when the working stack needs to grow or shrink.

4 TICS: Implementation

TICS is built around the MSP430FR5969 [44] MCU with 64 KB non-volatile (FRAM) and 2 KB volatile (SRAM) memory. The compiler back-end *instruments* the assembly to support stack segmentation. The code instrumentation is done via the LLVM utility library LibTooling [1], which is intended for both static analysis and code transformations. We employed *code transformation* rather than compiler support, to allow for portability, enabling the use of multiple compilers, and in turn eliminating the need for re-implementing the instrumentation. In order to produce the target binary, we used MSP430-GCC version 7.

Stack Segmentation. In TICS, the stack segmentation is employed at function entries and exits. Before the stack grows or shrinks, TICS checks the stack frame size of the corresponding function (known at compile time) to determine if the function can be executed by using the current working stack. If there is not enough space in the working stack, a stack grow procedure is initiated so that the working stack points the next segment in the segment array. Since the arguments of the function remain in the previous segment, these arguments are copied from this segment to the empty working stack. If a stack shrink is needed, the caller stack is restored, the working stack is changed and a segment checkpoint is performed if the previously checkpointed data belongs to a segment lower than the current working stack.

All these operations are depicted by steps 1–3 in Figure 7. To enable these operations, we modified the compiler back-end to insert the required stack availability check and argument copying operations—the size of a stack segment is determined at compile time and its minimum size depends on the minimal stack requirements of the functions in the source.

Memory Consistency Management. To implement undo logging so that the changes in non-volatile memory locations (other than the working stack) can be undone, global variable and pointer manipulations are instrumented. Since pointers can point not only to global data but also to the working stack or segment checkpoint in non-volatile memory, the instrumentation is done by checking if the physical address is in the working stack or not. If so, the memory manager logs the contents of the memory cell in the undo log[3].

Automatic Checkpoints. To keep consistency of checkpointed data—as the system can die while performing a checkpoint—the checkpoint data is double buffered in non-volatile memory. A flag is used to provide an exact barrier after which the checkpoint is ready to be used as a restore point. These enable checkpoint operations to be atomic. Checkpoint restoration happens when the system reboots due to a power failure. Current implementation supports: (i) *timer-driven* checkpointing; where the runtime interrupts program execution and checkpoints the system state periodically at a given frequency; (ii) *hardware-assisted* checkpointing, e.g. [5] where a voltage level based interrupt triggered upon a low-energy state to perform a checkpoint; and (iii) manual checkpoints. It is worth mentioning that TICS disables (automatic) checkpoints before *interrupt service routines* and places an implicit checkpoint right after *return-from-interrupt* (ISRs) instruction. This is sufficient to prevent memory inconsistency while servicing interrupts—if a power failure prevents the completion of an ISR, the system will continue as if interrupt did not occur (the corresponding ISR will not be executed again) right after the recovery from the power failure.

Time Annotations. Each write to a time-annotated variable is instrumented so that the timestamp value associated with the variable is updated. To implement exception-based time annotations, we instrumented `@expires/catch` block so that a timer is set considering the data expiration constraints. Moreover, we instrumented necessary instructions for undo-logging the memory modifications and changing the control flow upon data expiration. TICS with time-sensitive programs requires the ability to measure time across power outages using a remanence-based timer [21, 37] or a Real-Time Clock with a small capacitor [18]—persistent timekeeping is mandatory to update timestamps and to handle time annotated source files.

[3]Memory management is implemented fully in software as microcontrollers, e.g. MSP430FR59* [44], do not have a memory management unit.

Intermitt.		'Greenhouse monitoring' routines: Sense Moisture	Sense Temp.	Compute	Send	Consist.
4%	plain C	9	9	9	0	✗
	plain C + TICS	0	0	0	0	✓
	TinyOS	0	0	0	0	✓
	TinyOS + TICS	0	0	0	0	✓
48%	plain C	29	29	29	20	✗
	plain C + TICS	20	20	20	20	✓
	TinyOS	29	29	29	20	✗
	TinyOS + TICS	20	20	20	20	✓
100%	plain C	47	47	47	47	✓
	plain C + TICS	45	45	45	45	✓
	TinyOS	47	47	47	47	✓
	TinyOS + TICS	44	44	44	44	✓

Table 1. Real-world program with TICS on intermittent power (4%, 48% and 100% intermittency rate). We ran four applications implementing greenhouse monitoring (GHM): in C and in TinyOS (with and without TICS instrumentation). We measured how many times each GHM routine executed. Only these programs that consistently executed the same number of routines were considered correct.

5 Evaluation

We investigate the execution overhead of TICS for various applications, comparing to the state-of-the-art intermittent runtimes. We demonstrate how TICS enables porting of arbitrary C programs as well as TinyOS code—for the first time we demonstrate *successful execution of legacy code for sensor networks into intermittently-powered domain*. We also show results from a user study we conducted comparing TICS to task-based programming. We found that TICS has comparable overhead to state-of-the-art runtimes while providing **a complete set of features available to the regular C programmer**.

5.1 Porting Legacy Code: TinyOS to Intermittent World

To prove the claim that TICS enables automatic porting of existing/legacy C code for non-intermittently powered systems, we instrument an unmodified TinyOS program for Greenhouse Monitoring (GHM). GHM executes in an infinite loop *sense moisture* of soil, *sense temperature* of ambient, *Compute* measurement averages and *Send* over a wireless interface. We compare Plain C and TinyOS [25] versions of GHM with and without TICS instrumented checkpoints. Both apps were executed on the same microcontroller as before (MSP430FR5969 [44] evaluation board) with artificially generated power *intermittency traces*, i.e. MCU was brought to hardware reset following a pre-programmed pattern. We compare the results of executing the Plain C and TinyOS versions of GHM in Table 1 for varying levels of intermittency. We measured how many times each GHM routine executed successfully. Only these programs that consistently executed the same number of all routines were considered correct.

Results. We observe that TICS allows to work at *any* intermittency conditions and it executes legacy code correctly. This shows that TICS can run semi-sophisticated legacy

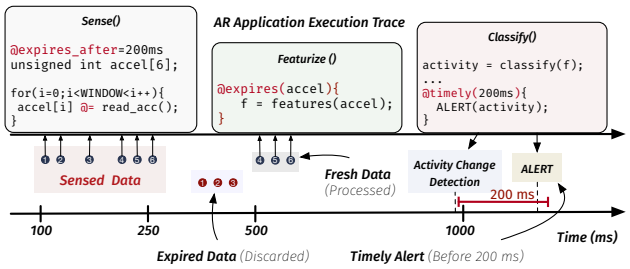

Figure 8. Timely execution of the sample AR application: TICS catches data expiration, discards stale data and ensures timely branches by following the programmer annotations.

TinyOS programs without any manual program porting need. It is worth mentioning that TinyOS is an event-based operating system and porting event-based legacy code might require some manual modifications for the sake of the semantically correct execution of the application—in particular, timely-sensitive handling of the events should be implemented by the time annotations provided by the TICS in order to guarantee semantically correct results. However, if the programmer omits such manual modifications, TICS still guarantees forward progress of the computation as well as the memory consistency of the event-based applications. In Section 5.3, we also demonstrate the porting of existing computation-based benchmarking applications. Apart from injecting time annotations (if required), all porting is handled by TICS automatically without any manual intervention. Therefore, the evaluation results later on support and complement this result.

5.2 Time-sensitive Intermittent Computation

For the evaluation of time-sensitive execution of intermittent programs, we considered an existing activity recognition (AR) application used in prior work [9, 28, 31] (this application is also used for benchmarking in Section 5.3). The AR application obtains a window of three-axis accelerometer sensor readings and determines whether the device is moving or stationary. In the training phase, the mean and standard deviation features of a window of samples are extracted. Then, in the recognition phase, the activity is determined by performing a nearest neighbor classification. In order to observe the time consistency violations described in Section 2.1.3, we provided two versions of the AR application: (i) manual management of time (and using MementOS-like checkpoints); and (ii) TICS annotated application. We run these applications by powering our MSP430FR5969 [44] wirelessly with 915 MHz Powercast TX91501-3W transmitter [36]. The microcontroller was connected to a Powercast P2110-EVB receiver (with on-bard 10 μF storage capacitor). We tested the execution of these applications at the same distances resulting in almost the same (i) power failure rates, (ii) charging and (iii) off-time. We observed the number of time consistency violations.

Time Consistency Violation	Potential Count (during experiment)	Observed Violations w/o TICS	w/ TICS
Timely Branch	256	32 ✗	0 ✓
Time Misalignment	870	78 ✗	0 ✓
Data Expiration	870	173 ✗	0 ✓

Table 2. Time consistency violation statistics for the AR application running intermittently. Our results indicate that TICS eliminates these violations by demanding little modifications on the legacy software.

The lines of code where the accelerometer is sampled and the corresponding timestamp is assigned are the potential points for time misalignment violation. Specifically, a timestamp can be assigned to the sensed data relatively long time after the sensor sampling, due to a power failure and long charging time—in both applications there were 870 accelerometer sampling where time misalignment violations could potentially occur. The obtained samples are also subject to data expiration violation while they are consumed for training and classification. In these applications, we considered data to be fresh and useful if it is consumed within 200 ms time window—it is considered to be stale otherwise (see Fig. 8). In order to keep track of the duration of the recognized activities, both applications maintain timestamp. A timely branch that uses this timestamp is required to alert about activity changes; e.g. if the duration of the activity is less than 200 ms this indicates an activity switch. There were 256 points in the execution where a potential timely branch violation could occur.

Results. Table 2 summarizes our results. We observed that TICS prevents all time consistency violations, thanks to easily injected time annotations, where the other application led to 32 timely branch violations, 78 time misalignment violations and 173 data expiration violations. Our results indicate that TICS ensures timely intermittent execution by providing little modifications on the legacy software via its time annotations.

5.3 TICS System Efficiency

TICS supports all C language features—including pointers and recursion— thanks to its memory consistency manager. This implementation eliminates system starvation by allowing porting any kind of legacy software to the intermittent computing world—breaking the limitations of the prior work. Here we provide a performance comparison of TICS with the prior work to explore its execution overhead.

We have compared TICS against three state-of-the-art task-based systems: InK [46], MayFly [20] and Alpaca [31]. In addition, we compared TICS against naïve checkpoint-based system that logs the complete stack and all global variables (which closely resembles what MementOS [39] does) and Chinchilla [32]—state-of-the-art checkpoint-based

system that promotes all variables to global data and statically logs these. Chinchilla was re-compiled from its GitHub source [32] with LLVM version 3.8 (the strict requirement for Chinchilla). InK, MayFly, and Alpaca were compiled with the standard GCC compiler (msp430-gcc version 6.2.1.16). Finally, for completeness, we compare all systems to plain C.

Application Benchmarks. We chose three representative applications, used earlier by most studies on systems for intermittently-powered devices: (i) *bitcount* (BC), (ii) *Cuckoo filter* (Cuckoo) and (iii) *Activity Recognition* (AR) (as indicated in Section 5.2) [16]. BC implements bit counting in a random string with seven different methods (including recursion), later cross-verifying for correctness; Cuckoo implements cuckoo filter over a set of pseudo-random numbers, then performing sequence recovery using the same filter; AR implements physical activity recognition based on machine learning with locally stored accelerometer data. For a fair comparison, the experiments were conducted using a continuously-powered TI MSP-EXPFR5969 evaluation board [44]. Each application was verified for correctness at the end of each execution. Cuckoo cannot be implemented in MayFly since loops are not allowed in a MayFly task graph. Also, BC used for the evaluation of Chinchilla, see e.g. [32, Fig. 8–10], was not the original one, as the authors have *manually removed the recursion* to make it work with their system.

5.3.1 TICS against Chinchilla. Chinchilla converts each local variable of a function to a corresponding global variable in non-volatile memory at compile time. This conversion prevents stack manipulation via pointers and in turn checkpointing the whole stack due to pointer manipulations. Chinchilla must know in advance the local variables in order to allocate corresponding global variables in non-volatile memory—recursive function calls and in turn, existing applications that exploit recursive implementations cannot be supported. Moreover, due to the local-to-global conversion via bypassing stack allocation of local variables, there is an explosion in the number of global variables—decreasing the scalability of memory requirements. Inline functions further complicate this issue: the corresponding global variables are needed to be allocated per every line where the function is inlined. As an example, if an inline function of one local variable is called 100 times, then 100 different global variables need to be created. These issues are the major limitations of Chinchilla, making it an incomplete system. Inspecting our results presented in Figure 9, TICS is able to execute *all* benchmarks, while Chinchilla cannot run recursion-based code, i.e. BC. Due to the dynamic memory logging employed by TICS, the execution time overhead will vary per benchmark. Additionally, the compiler optimization level has a significant effect because the runtime code is also affected by the lack of optimization.

	InK		Chinchilla		TICS	
	.text	*.data*	*.text*	*.data*	*.text*	*.data*
AR	3442	4459	12870	8986	6878	1364
BC	2922	4433	10902	8658	5944	1488
CF	2648	4693	12128	9050	7178	1948

Table 3. The memory consumption (in B) for three applications written in InK, Chinchilla and TICS.

5.3.2 Micro-benchmarking TICS. The execution time overhead of TICS with the number of checkpoints for different working stack sizes is given in Figure 9 (center row). As working stack size gets bigger, the number of working stack change driven checkpoints decreases since on-demand stack requirement of the applications are fulfilled—S2 configuration did not lead to a working stack changes and in turn checkpoints and S1 led to considerable number of working stack changes and therefore also more checkpoints. On the other hand, increasing the working stack size also increases the overhead of a single checkpoint since the logged data is bigger—there will be always a trade-off. Among benchmarking applications, AR led to a considerable amount of working stack change driven checkpoints with configuration S1 due to its varying stack size requirements. We also enabled timer-driven checkpoints with a frequency of 10 ms that ensure the forward progress—configurations S1* and S2* indicate the configurations S1 and S2 with timer-driven checkpoints enabled. TICS checkpoints do not introduce significant overhead since only the working stack and registers are logged.

5.3.3 TICS Against Task-based Systems. We selected configurations S1* and S2* to asses the execution time performance of TICS considering the task-based runtimes—right column of Figure 9. For the fairness of comparison against task-based systems, apart from timer-driven checkpoints in S1* and S2*, we placed checkpoints to configuration S2 at task-boundaries for TICS (shown as ST) and our naïve MementOS-like [39] implementations. We observed that selecting a reasonable working stack size, TICS reaches almost the performance of existing task-based systems.

5.3.4 TICS Memory Overhead. Table 3 presents a comparison of memory overhead of the benchmarking applications implemented in InK (task-based system), Chinchilla (checkpoint-based system) and TICS. The `.data` section overhead of TICS depends on the size of the configurable stack segment array (which was 2048 B) and undo log (which was 2048 B) both are excluded from the `.data` section. The code size in selected applications is dependent on not only the application source but also on the stack segmentation and memory consistency management implementations in TICS. Overall, we see that for all benchmarks TICS has *significantly lower* memory overhead than Chinchilla—more than twice `.text` and more than six times for `.data`. Comparing to InK, TICS `.data` is also significantly lower, except for `.text`.

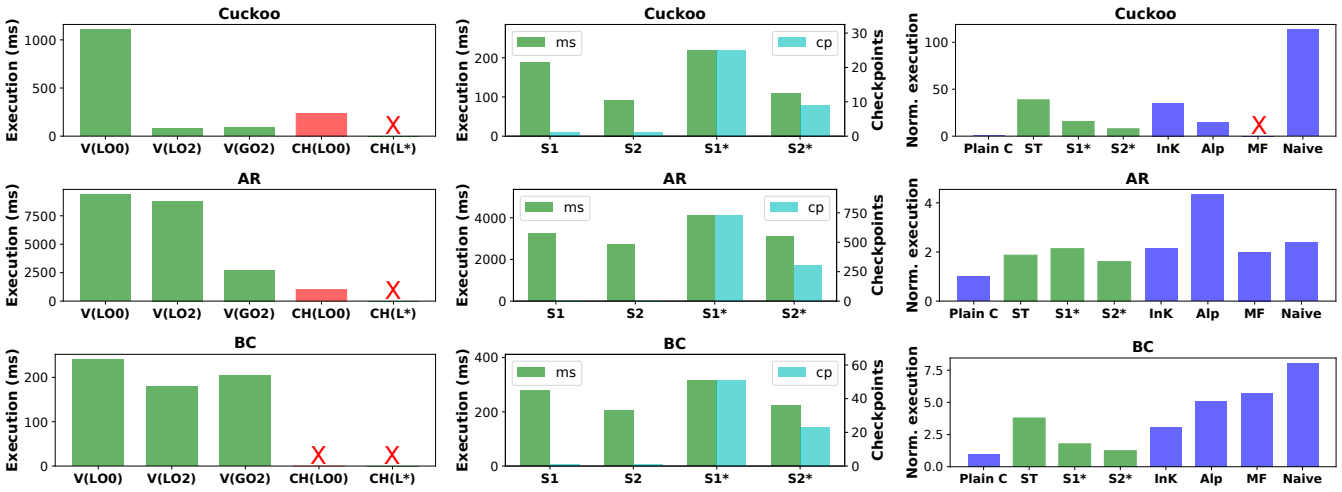

Figure 9. Benchmark performance. *Left column*: TICS to Chinchilla comparison; *Center column*: micro-benchmarking of TICS; *Right column*: TICS to task-based systems comparison. **T**, **CH**, **Alp**, **MF**: TICS, Chinchilla, Alpaca, MayFly; **LO0, LO2**: LLVM-compiled code with –O0, and –O2 optimization. **L***: all compilations options of GCC and all of LLVM except –O0. **GO2**: GCC-complied code with –O2 optimization. **S1, S2**: configurations with different working stack sizes imposing a different checkpoint frequency and checkpoint time—**S1**=50 B, **S2**=256 B; **S1***, **S2***: the same stack configurations as **S1, S2** but with an additional timer checkpointing every 10 ms if there was no checkpoint due to the working stack. **ST** denotes **S2** with checkpoints at the task boundaries. Red cross (✗) denotes the code did not compile with the chosen compiler/optimization.

Operation	Configuration Variables	Duration (μs)
Stack grow/shrink		max 345
Checkpoint logic	0 B seg. | 64 B seg. | 256 B seg.	264 | 464 | 656
Restore logic	0 B seg. | 64 B seg. | 256 B seg.	273 | 475 | 664
Pointer access	no log | log 4 B (64 B)	13 | 308 (371)
Roll back from undo log	4 B | 64 B	234 | 294

Table 4. TICS overhead, split per runtime operation. Results obtained with GCC (optimization –O2) at 1 MHz.

5.3.5 TICS Point-to-point Overheads.

Table 4 presents the detailed overhead of TICS runtime operations. The checkpoint and restore operations include saving registers and working stack in non-volatile memory using a two-phase commit operation—the working stack size has a direct impact on the checkpoint overhead. The constant checkpoint overhead without saving the working stack segment is depicted as 0 B size in the table. The *stack grow/shrink* operations update the working stack to point another segment in the segment array. During pointer manipulation, TICS checks the pointer address to see if the working stack is targeted. If this is the case, there is no need for the undo logging and the working is stack directly manipulated. Otherwise, TICS logs the original value in undo log—the overhead of different variable sizes are depicted in the table. The time it takes to recover the original value of a variable from the undo log depends on the variable sizes.

5.4 User Study and Developer Effort

We have designed a large online user study. The goal was to objectively assess the time to design a TICS application.

Methodology. At the beginning of an online survey each participant was given an introduction to intermittent execution and to TICS and InK [46]. Then, we have then asked participants to find bugs in three simple programs: (i) *swap of two variables (with no use of a temporary variable)*, (ii) *bubble sort*, and (iii) *program that considers variable expiration based on time*. Each program was written separately in TICS and in InK and had *exactly the same type of bug*, at *exactly one line of the program*. Users were asked to point to a line that contained that bug and specify the correct statement.

Each program with a bug was presented to a user on a separate page. Additionally, *time* spent on finding a bug in each of the programs was measured. No corrections of the given answers were possible once the answer was submitted. We randomized the order in which each program appeared at the respondent's screen in order to remove presentation bias against one language and objectify bug finding time.

User Pool. At the time of writing this paper, a total of 90 responses were collected. 78% of all respondents had at least two years of programming experience. Almost 83% of respondents had average or below average knowledge of embedded systems powered by energy harvesting technologies.

Result. Results are shown in Figure 10. We observe that in all cases it was (i) *harder to find a bug* and (ii) *users were more prone to error* when exposed to a task-based language. Statistically, Wilcoxon T Test on all programs' bug search time rejected the hypothesis that TICS/InK results were the same with p-value below 0.001. In other words, TICS is a *more*

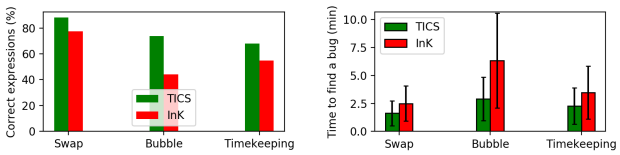

Figure 10. TICS user study results. For all three test programs, *Swap*, *Bubble* and *Timekeeping*, users found that it is easier with TICS to identify a bug and were more accurate in correcting the TICS program than that of InK [46]. Whiskers in the right-hand side figure denote standard deviation.

Runtime	Pointer Support	Recursion Support	Scalability	Timely Execution	Porting Effort
Mayfly [20]	No ✗	No ✗	Poor ✗	Yes ✓	High ✗
Alpaca [31]	No ✗	No ✗	Poor ✗	No ✗	High ✗
Ratchet [45]	Yes ✓	No ✗	Poor ✗	No ✗	High ✗
Chinchilla [32]	Yes ✓	No ✗	Poor ✗	No ✗	None ✓
Ink [46]	No ✗	No ✗	Poor ✗	Yes ✓	High ✗
TICS (*this work*)	**Yes ✓**	**Yes ✓**	**High ✓**	**Yes ✓**	**None ✓**

Table 5. State of the art programming models.

user-friendly system than a task-based one. As the complexity of a program increased users had difficulty finding a bug in an InK program (for *Bubble Sort* in half of the cases users were wrong). Regarding the subjective evaluation of TICS against InK, participants considered TICS to be *more intuitive, easier and conciser* than InK.

6 Related Work

In Table 5 key characteristics of TICS are compared to those of Mayfly [20], Alpaca [31], Ratchet [45], Chinchilla [32] and InK [46]. In this section we compare some of these characteristics from the state of the art to TICS.

Checkpointing Systems. Systems that automatically determine checkpoint placement at compile-time like [6, 32, 45] are most closely related to this work. HarvOS parses the control flow graph of a program and instruments with energy-aware checkpoints, requiring a small amount of programmer intervention to place effectively. Ratchet functions by placing checkpoints at the boundaries of idempotent sequences of instructions. Chinchilla over-instruments programs with checkpoints by storing some variables in non-volatile memory, and disabling/enabling checkpoints heuristically. Apart from the aforementioned studies, Mementos [39] was the first checkpointing scheme, using intermittent voltage checks to decide when to save state. QuickRecall [23] and Hibernus [5] extended this work with newer non-volatile memories. DINO [28] laid out the memory consistency problems that will arise with intermittent computing for mixed-volatility processors. TICS builds on these early techniques, however, these systems do not consider timely execution of the applications.

Task-based Programming Models. Alpaca [31] and related works [9] focus on providing control flow and data flow mechanisms while reducing the memory footprint from multi-versioning. Mayfly [20] provides explicit semantics for specifying timing constraints on sensor data in a task-based language. InK [46] provides a way to handle events and interrupts from clock sources, sensors, and energy in the environment, despite power failures. Task-based systems require a custom programming model, which leads to added programmer intervention and complexity. Task decomposition is a manual process that is error-prone and not resilient to changes in the availability of energy in the environment.

Non-volatile Processors. Integration of non-volatile components, e.g. non-volatile registers, to the processor architecture provides automatic management of forward progress and memory consistency [29, 30]. This eliminates the need for handling these properties explicitly by the programmer. However, non-volatile architectures consume more power, they have increased area and decreased frequency as compared to general-purpose volatile processors with SRAM-based flip-flops [22]. TICS targets off-the-shelf processors with hybrid volatile and non-volatile memory in the market.

7 Conclusion and Future Work

TICS is a runtime for intermittently powered systems that enables full use of C features like pointers and recursion through a memory consistency management scheme (data versioning and stack segmentation) and provides semantics for easily porting time-*sensitive* programs to the intermittent domain while maintaining correctness. Guarantees on worst case checkpointing time are provided, ensuring TICS scales as applications become more complex. We evaluated TICS against the state of the art, showing reasonable overhead nearly matching the performance of task-based systems. We conducted a user study, where participants found TICS more intuitive than the task-based approach. In the future, we anticipate exploring ways to automatically import or infer timing semantics and rules from legacy code in TinyOS or other systems. Virtualizing the I/O interface across power failures could also lead to better ported applications.

8 Acknowledgments

We thank our anonymous reviewers for their useful comments. This research was supported by Netherlands Organisation for Scientific Research, partly funded by the Dutch Ministry of Economic Affairs, through TTW Perspectief program ZERO (P15-06) within Project P4, and by National Science Foundation through grants CNS-1850496 and CNS-1453607. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Clang 7 libtooling. https://github.com/llvm-mirror/clang/blob/master/docs/LibTooling.rst, March 2019. Last accessed: Jan. 20, 2020.

[2] TICS website. https://github.com/tudssl/tics, January 2020. Last accessed: Jan. 16, 2020.

[3] Saad Ahmed, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, Naveed Anwar Bhatti, and Luca Mottola. Towards smaller checkpoints for better intermittent computing. In *Proc. IPSN*, pages 132–133, Porto, Portugal, 2018. ACM/IEEE.

[4] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Commun. Surveys Tuts.*, 17(4):2347–2376, Fourth Quarter 2015.

[5] Domenico Balsamo, Alex S. Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M. Al-Hashimi, Geoff V. Merrett, and Luca Benini. Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 35(12):1968–1980, 2016.

[6] Naveed Bhatti and Luca Mottola. HarvOS: Efficient code instrumentation for transiently-powered embedded devices. In *Proc. IPSN*, pages 209–219, Pittsburgh, PA, USA, 2017. ACM/IEEE.

[7] John Burgess, Brian Gallagher, David Jensen, and Brian Neil Levine. MaxProp: routing for vehicle-based disruption-tolerant networks. In *Proc. INFOCOM*, Barcelona, Spain, 2006. IEEE.

[8] An Chen. A review of emerging non-volatile memory (NVM) technologies and applications. *Solid-State Electronics*, 125:25–38, November 2016.

[9] Alexei Colin and Brandon Lucia. Chain: Tasks and channels for reliable intermittent programs. In *Proc. OOPSLA*, pages 514–530, Amsterdam, The Netherlands, 2016. ACM.

[10] Alexei Colin and Brandon Lucia. Termination checking and task decomposition for task-based intermittent programs. In *Proc. Conference on Compiler Construction*, pages 116–127, Vienna, Austria, 2018. ACM.

[11] Alexei Colin, Emily Ruppel, and Brandon Lucia. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proc. ASPLOS*, pages 767–781, Williamsburg, VA, USA, 2018. ACM.

[12] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proc. LCN*, pages 455–462, Tampa, FL, USA, 2004. IEEE.

[13] Salma Elmalaki, Huey-Ru Tsai, and Mani Srivastava. Sentio: Driver-in-the-loop forward collision warning using multisample reinforcement learning. In *Proc. SenSys*, pages 28–40, Shenzhen, China, 2018. ACM.

[14] Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, and Hari Balakrishnan. The pothole patrol: Using a mobile sensor network for road surface monitoring. In *Proc. MobiSys*, pages 29–39, New York, NY, USA, 2008. ACM.

[15] A. Fioravanti-Score, Sarah V. Mitchell, and J. Michael Williamson. Use of Satellite Telemetry Technology to Enhance Research and Education in the Protection of Loggerhead Sea Turtles. In *Proc. Annual Symp. on Sea Turtle Biology and Conservation*, South Padre Island, TX, USA, 1999. NOAA.

[16] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. Workload Characterization Workshop*, pages 3–14, Austin, TX, USA, 2001. IEEE.

[17] Josiah Hester, Travis Petersy, Tianlong Yuny, Ronald Petersony, Joseph Skinnery, Bhargav Golla, Kevin Storer, Steven Hearndon, Kevin Freeman, Sarah Lordy, Ryan Haltery, David Kotz, and Jacob Sorber. Amulet: An energy-efficient, multi-application wearable platform. In *Proc. SenSys*, pages 216–229, Stanford, CA, USA, 2016. ACM.

[18] Josiah Hester and Jacob Sorber. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proc. SenSys*, pages 19:1–19:13, Delft, The Netherlands, 2017. ACM.

[19] Josiah Hester and Jacob Sorber. The future of sensing is batteryless, intermittent, and awesome. In *Proc. SenSys*, pages 21:1–21:6, Delft, The Netherlands, 2017. ACM.

[20] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proc. SenSys*, pages 17:1–17:13, Delft, The Netherlands, 2017. ACM.

[21] Josiah Hester, Nicole Tobias, Amir Rahmati, Lanny Sitanayah, Daniel Holcomb, Kevin Fu, Wayne P. Burleson, and Jacob Sorber. Persistent clocks for batteryless sensing devices. *ACM Trans. Embed. Comput. Syst.*, 15(4):77:1–77:28, August 2016.

[22] Matthew Hicks. Clank: Architectural support for intermittent computation. In *Proc. ISCA*, pages 228–240, Toronto, ON, Canada, 2017. ACM.

[23] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. Quickrecall: A HW/SW approach for computing across power cycles in transiently powered computers. *J. Emerg. Technol. Comput. Syst.*, 12(1):8:1–8:19, July 2015.

[24] Fredrik Larsson and Bengt-Erik Mellander. Abuse by external heating, overcharge and short circuiting of commercial lithium-ion battery cells. *Journal of The Electrochemical Society*, 161(10):A1611–A1617, 2014.

[25] Philip Levis, Sam Madden, Joseph Polastre, Rober Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. TinyOS: An operating system for sensor networks. In Werner Weber, Jan M. Rabaey, and Emile Aarts, editors, *Ambient intelligence*, pages 115–148. Springer, Berlin, Germany, 2005.

[26] Hong Lu, Jun Yang, Zhigang Liu, Nicholas D. Lane, Tanzeem Choudhury, and Andrew T. Campbell. The jigsaw continuous sensing engine for mobile phone applications. In *Proc. SenSys*, pages 71–84, Zürich, Switzerland, 2010.

[27] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent computing: Challenges and opportunities. In *Proc. SNAPL*, pages 8:1–8:14, Alisomar, CA, USA, 2017.

[28] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proc. PLDI*, pages 575–585, Portland, OR, USA, 2015. ACM.

[29] Kaisheng Ma, Xueqing Li, Karthik Swaminathan, Yang Zheng, Shuangchen Li, Yongpan Liu, Yuan Xie, John Jack Sampson, and Vijaykrishnan Narayanan. Nonvolatile processor architectures: Efficient, reliable progress with unstable power. *IEEE Micro*, 36(3):72–83, May/Jun. 2016.

[30] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. Architecture exploration for ambient energy harvesting nonvolatile processors. In *Proc. HPCA*, pages 526–537, Burlingame, CA, USA, 2015. IEEE.

[31] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent execution without checkpoints. In *Proc. OOPSLA*, pages 96:1–96:30, Vancouver, BC, Canada, 2017. ACM.

[32] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *Proc. OSDI*, Carlsbad, CA, USA, 2018. USENIX.

[33] Sujay Narayana, R. Venkatesha Prasad, Vijay S. Rao, and Chris Verhoeven. SWANS: sensor wireless actuator network in space. In *Proc. SenSys*, pages 23:1–23:6, Delft, The Netherlands, 2017. ACM.

[34] Anh Nguyen, Raghda Alqurashi, Zohreh Raghebi, Farnoush Banaei-Kashani, Ann C. Halbower, and Tam Vu. A lightweight and inexpensive in-ear sensing system for automatic whole-night sleep stage monitoring. In *Proc. SenSys*, pages 230–244, Stanford, CA, USA, 2016. ACM.

[35] M. R. Palacín and A. de Guibert. Why do batteries fail? *Science*, 351(6273), 2016.

[36] Powercast Corp. Powercast hardware development kits website. https://www.powercastco.com/products/development-kits/, 2014. Last accessed: Jan. 20, 2020.

[37] Amir Rahmati, Mastooreh Salajegheh, Dan Holcomb, Jacob Sorber, Wayne P. Burleson, and Kevin Fu. TARDIS: time and remanence decay in SRAM to implement secure protocols on embedded devices without clocks. In *Proc. Security Symposium*, pages 36–36, Bellevue, WA, USA, 2012. USENIX.

[38] Benjamin Ransford and Brandon Lucia. Nonvolatile memory is a broken time machine. In *Proc. Workshop on Memory Systems Performance and Correctness*, pages 5:1–5:3, Edinburgh, Scotland, 2014. ACM.

[39] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System support for long-running computation on RFID-scale devices. In *Proc. ASPLOS*, Newport Beach, CA, USA, 2011. ACM.

[40] Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. Design of an RFID-based battery-free programmable sensing platform. *IEEE Trans. Instrum. Meas.*, 57(11):2608–2615, November 2008.

[41] Mahadev Satyanarayanan, Wei Gao, and Brandon Lucia. The computing landscape of the 21st century. In *Proc. HotMobile*, pages 45–50, Santa Cruz, CA, USA, 2019. ACM.

[42] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: A language and runtime system for perpetual systems. In *Proc. SenSys*, pages 161–174, Sydney, Australia, 2007.

[43] Jethro Tan, Przemysław Pawełczak, Aaron Parks, and Joshua R. Smith. Wisent: Robust downstream communication and storage for computational RFIDs. In *Proc. INFOCOM*, San Francisco, CA, USA, 2016. IEEE.

[44] Texas Instruments. MSP430FR5994 launchpad development kit. http://www.ti.com/tool/MSP-EXP430FR5994. Last accessed: Jan. 20, 2020.

[45] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *Proc. OSDI*, pages 17–32, Savannah, GA, USA, 2016. ACM.

[46] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemysław Pawełczak, and Josiah Hester. InK: Reactive kernel for tiny batteryless sensors. In *Proc. SenSys*, pages 41–53, Shenzhen, China, 2018. ACM.

[47] Hong Zhang, Jeremy Gummeson, Benjamin Ransford, and Kevin Fu. MOO: A batteryless computational RFID and sensing platform (technical report um-cs-2011-020). https://web.cs.umass.edu/publication/docs/2011/UM-CS-2011-020.pdf, 2011. Last accessed: Jan. 20, 2020.

[48] Pei Zhang, Christopher M. Sadler, Stephen A. Lyon, and Margaret Martonosi. Hardware design experiences in ZebraNet. In *Proc. SenSys*, pages 227–238, Baltimore, MD, USA, 2004. ACM.

A Artifact Appendix

A.1 Abstract

TICS is a framework that allows for C programs to be executed on intermittent power harvested from the environment. TICS consists of multiple components that together make sure that the program that is being executed continues where it left off after a power failure. Additionally, TICS does this in a way that leads to checkpoint times that can be bounded to a reasonable upper limit, making reasoning about checkpoint placement dynamically possible (although this is not explored in the current version).

TICS is intended to be used with the MSP430FR5969 microcontroller but can be adapted to work with any MSP-based microcontroller that consists of non-volatile main memory.

The main components of TICS are:

- TICS runtime for memory logging and checkpoint management;
- TICS compiler backend (GCC and LLVM) for stack segmentation management;
- TICS source instrumentation for variable instrumentation.

A.2 Artifact Check-list (Meta-information)

- **Program:** msp430-gcc, llvm, memlog, benchmarks
- **Compilation:**

```
## Building GCC
# Required packages:
build-essential flex bison texinfo
ncurses-dev zlib1g-dev bash curl

## Build commands:
$ cd msp430-gcc-tics
$ ./build.sh

## Building LLVM
# Required packages:
make gcc cmake python zlib1g-dev

# Build commands:
$ cd llvm-tics
$ ./build.sh

## Building Source Instrumentation Tool
# Build commands:
$ cd tics/source-instrumentation/\
  memory-log-instrumentation
$ mkdir build
$ cd build
$ cmake ../
$ make
```

- **Transformations:**

```
memlog <benchmark>.c
```

- **Binary:**

```
ftest_cuckoo, ftest_ar, ftest_bitcount, greenh_temp_tinyos
```

- **Hardware:** Texas Instruments MSP430FR5969 launchpad development kit
- **Experiments:** Hardware breakpoints and cycle counter
- **How much disk space required (approximately)?:** 30 GB (compilers)
- **How much time is needed to prepare workflow (approximately)?:** one day
- **How much time is needed to complete experiments (approximately)?:** 2 hours
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** Yes 10.5281/zenodo.3563082 (https://doi.org/10.5281/zenodo.3563082).

A.3 Description

A.3.1 Hardware Dependencies

Texas Instruments MSP430FR5969 launchpad development kit.

A.3.2 Software Dependencies

Download and extract the MSP430 GCC support files from Texas Instruments website.

Further instructions described in README.md in archive (DOI): 10.5281/zenodo.3563082 (https://doi.org/10.5281/zenodo.3563082).

A.3.3 Data Sets

A.4 Installation

Described in README.md in archive (DOI): 10.5281/zenodo.3563082 (https://doi.org/10.5281/zenodo.3563082).

A.5 Experiment Workflow

Described in README.md in archive (DOI): 10.5281/zenodo.3563082 (https://doi.org/10.5281/zenodo.3563082).

A.6 Evaluation and Expected Result

Described in README.md in archive (DOI): 10.5281/zenodo.3563082 (https://doi.org/10.5281/zenodo.3563082).