Virtualizing Intermittent Computing

Çağlar Durmaz, Kasım Sinan Yıldırım, *Member, IEEE*, and Geylani Kardas

***Abstract*—Intermittent computing requires custom programming models to ensure the correct execution of applications despite power failures. However, existing programming models lead to the programs that are hardware dependent and not reusable. This article aims at virtualizing intermittent computing to remedy these problems. We introduce PureVM, a virtual machine that abstracts a transiently powered computer, and PureLANG, a continuation-passing-style programming language to develop programs that run on PureVM. This virtualization, for the first time, paves the way for portable and reusable transiently powered applications.**

***Index Terms*—Batteryless Internet of Things (IoT), domain-specific language, energy harvesting, intermittent computing, virtual machine.**

I. Introduction

In THE past decade, the progress in energy harvesting circuits and the decrease in power requirements of processing, sensing, and communication hardware promised the potential of freeing the Internet of Things (IoT) devices from their batteries. Recent works demonstrated several microcontroller-based devices that can work without the need for batteries by harvesting energy from ambient sources, such as solar and radio frequency [1]–[3]. Batteryless devices store the harvested ambient energy into a tiny capacitor that powers the microcontroller and the peripherals. A batteryless device can compute, sense, and communicate when the energy stored in its capacitor is above an operating threshold. It turns off and loses its volatile state (e.g., the contents of the CPU, peripheral registers, and the volatile memory) when the energy level drops below this threshold. The device can turn on only after charging its capacitor again. This phenomenon, i.e., the intermittent execution due to power failures, led to the emergence of a new computing paradigm, the so-called *intermittent computing* [4], [5].

During intermittent execution, batteryless devices use the harvested energy to perform a short burst of computation. To recover their computation state and progress computation forward after a power failure, they need to save the computation state in nonvolatile memory before a power failure. Recent studies proposed programming models for intermittent computing to support these state logging and recovery operations. The proposed programming models provide language constructs (i.e., either *checkpoints* [5] or *tasks* [6]) to: 1) maintain the forward progress of computation and 2) keep the memory (i.e., computation state) consistent. However, with these models, programmers need to deal with the low-level details of the intermittent execution [7]. In particular, existing models pose the following deficiencies.

Explicit Burst Management: Programmers need to design their programs as a set of computation bursts that should fit in the capacitor. Thus, they explicitly identify the boundaries of these bursts via checkpoint placement or task decomposition. This situation increases the programming effort considerably. Even though [8] can automatically generate burst boundaries without programmer intervention, it is limited to checkpoint-based programs developed via the programming model in [9].

Hardware Dependency: The active time of a batteryless device depends on its capacitor size and its power consumption, i.e., its hardware configuration [10]. Programmers might need to identify different burst boundaries to execute their programs on a new device with a different hardware configuration. Alternatively, programs should be reanalyzed (e.g., via [8]) every time the program source changes since even minor modifications might lead to burst boundaries that do not fit in the capacitor. Therefore, existing intermittent programs are not portable, and in turn, not reusable.

Explicit I/O Management: Power failures that occur during I/O or interrupt handling might leave the memory in an inconsistent state. With existing models, programmers need to manually ensure the atomic execution of interrupt handlers or I/O operations [11]. This situation increases the programming burden and makes programs error prone.

In this article, we aim at virtualizing intermittent computing to remedy the deficiencies mentioned above. We introduce PureVM virtual machine and its software interface PureLANG language that abstract away the complicated aspects of intermittent execution. Thanks to PureVM and PureLANG, programmers focus only on their application logic, forget about power failures as if they are programming continuously powered systems, and develop portable intermittent programs. Shortly, this article introduces the following contributions.

1) *PureVM Virtual Machine:* We introduce PureVM, the first *virtual machine* for intermittent systems, which abstracts a transiently powered computer. This abstraction hides the details of intermittent execution and enables platform-independent program development via its implementations targeting different hardware (see Fig. 1).
2) *PureLANG Language:* We introduce PureLANG, a *continuation-passing-style* programming language,

Manuscript received 28 November 2021; revised 19 April 2022; accepted 13 May 2022. Date of publication 19 May 2022; date of current version 24 October 2022. This work was supported by TUBITAK SME R&D Start-Up Support Program under Project 7190539. *(Corresponding author: Kasım Sinan Yıldırım.)*

Çağlar Durmaz and Geylani Kardas are with the International Computer Institute, Ege University, 35100 Izmir, Turkey (e-mail: caglar.durmaz@ege.edu.tr; geylani.kardas@ege.edu.tr).

Kasım Sinan Yıldırım is with the Department of Information Engineering and Computer Science, University of Trento, 38123 Trento, Italy (e-mail: kasimsinan.yildirim@unitn.it).

Digital Object Identifier 10.1109/JIOT.2022.3176587

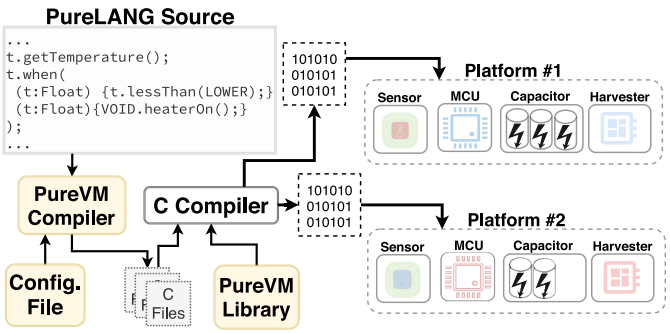

Fig. 1. Using PureLANG, a continuation-passing-style programming language, and PureVM, whose specification enables the execution of programs written in PureLANG, programmers can develop programs portable across several hardware platforms; without dealing with the intermittent execution.

intended to prevent programmers from reasoning about intermittent execution. PureLANG programs are translated into a set of recomposable and atomically executed PureVM instructions, i.e., primitive functions. In the continuation-passing style of programming, functions do not return values [12]. Rather, they pass control onto a continuation, which represents the current state of the computation [13]. The overhead of persisting the control flow via continuations is static and lower than that of persisting the whole call stack.

To the best of our knowledge, our work proposes the first *virtualization* attempt for intermittent computing. PureVM and PureLANG create the fundamental building blocks of intermittent computing from scratch that pave the way for portable transiently powered applications.

II. Background and Related Work

Frequent power failures lead to intermittent execution that poses several challenges to developing batteryless applications. We classify these challenges into two classes: 1) computation and I/O related challenges (A1–A3) and 2) programming challenges (B1 and B2). We describe them as follows.

A1—Nontermination and Memory Consistency: Power failures hinder the *forward progress* of the computation, which leads to nonterminating programs [14]. Nontermination occurs when the energy budget of the batteryless device is not enough to progress forward. This means that the energy stored in the capacitor is not sufficient to execute the instructions between two adjacent checkpoints or in the task boundaries. Therefore, the device keeps executing the same portion of code and cannot terminate. Power failures might also lead to *memory inconsistencies* [15]. To give an example, assume that a program is modifying the persistent variable `var` (i.e., a variable kept in nonvolatile memory) by executing the code block `{vector[var++]=10;}`. Since the variable `var` is updated after it is read, there is a *Write-After-Read* (W.A.R.) dependency. If a power failure occurs at this point, the reexecution of this code will increment the variable `var` again. Therefore, another element of the `vector` will be set to 10 in this case. Due to the W.A.R. dependency, there is a violation of *idempotency* since repeated computation produces different results. To prevent these issues, programmers should divide their programs into a set of idempotent code blocks (e.g., *tasks* [6]) that can execute *atomically* in a power cycle and can be safely restarted despite W.A.R. dependencies. A runtime library (e.g., [4] and [16]) is required to persist the program state in the nonvolatile memory, to manage power failures, and to reexecute the code blocks that could not complete in the previous power cycle.

A2—Control-Flow Inconsistencies: If the control flow depends on external inputs such as sensor readings, power failures might lead to erratic program behavior [17]. In particular, programmers need to pay special attention to implementing conditional statements that check persistent variables, whose values might be updated during I/O operations. For example, consider the case that a program reads a temperature value (`temp = read_sensor();`) and sets the variable `cooling` based on the temperature reading (`if(temp > limit) then cooling = true; else heating = true;`). If the temperature is less than a predefined limit, the variable `heating` will be set to true. If there is a power failure right after this operation and the program reexecutes, the program might read another temperature value higher than the limit. In this case, the program will set the variable `cooling` to true. At this point, both of the variables `cooling` and `heating` are true, which is logically incorrect [17]. These variables represent different actions, which should not be triggered simultaneously.

A3—Handling Interrupts: Interrupts cause dynamic branches, which move the control from the main thread of execution to the interrupt service routine. The main program and interrupt service routines might share the persistent state. If an interrupt service routine leaves the shared persistent variables partially updated due to a power failure, this situation might lead to memory inconsistencies [11].

B1—Platform Dependencies: The execution time of an intermittent program depends on several factors, such as available harvestable energy in the environment, the capacitor size (i.e., energy storage capacity), and the energy consumption profile of hardware and software. Intermittent programs need to be modified and restructured regarding these factors to eliminate nontermination and ensure computational progress [14].

B2—Reusability and Maintaining Difficulties: Platform and runtime dependencies make implementing reusable intermittent programs difficult [7]. For example, programmers using task-based models need to deal with task decomposition and task-based control flow [6]. Handling these issues is complicated and leads to the programs that are difficult to maintain.

A. State of the Art

We classify the prior art based on how they addressed the aforementioned challenges.

Checkpoint-Based Systems: Checkpointing runtime environments (e.g., [9] and [18]–[20]) persist the registers, global variables, and the call-stack of programs into nonvolatile memory at specific instants during program execution, to preserve the forward progress of computation (A1). Just-in-time

checkpointing runtimes (e.g., [19] and [20]) reactively save the computation state only when forward progress is compromised. A monitoring circuit checks the capacitor of the device, and an interrupt is triggered when the energy level is under a threshold to initiate a checkpoint. This strategy eliminates programmer intervention (to statically place checkpoints at compile time), but requires a hardware support [21]. In software-only approaches (e.g., [9] and [18]), programmers insert checkpoints manually at compile time. In particular, checkpointing overhead is closely related to the size of the call stack. Due to the call stack's dynamic nature (i.e., it grows and shrinks at runtime), the overhead of checkpoints is not static and varies during program execution. Thus, the energy stored in the capacitor might not be sufficient to execute the instructions between two adjacent checkpoints. Therefore, checkpoint placement is platform dependent and checkpointed programs are not reusable. Recent work [8] uses a statistical model of the energy consumption of each program path and performs automatic program transformation by determining checkpoint boundaries and placement in [9] to eliminate nontermination. There are also other studies (e.g., [5], [14], and [22]–[25]) that provide compilers to translate C programs into intermittent programs without programmer intervention. However, the C language does not provide abstractions for interrupt handling (A3) and atomic I/O (A2) operations on intermittent systems. The absence of these abstractions might lead to memory inconsistencies and nontermination.

Task-Based Systems: Task-based models (e.g., [4], [6], [11], [16], [26], and [27]) require programmers to structure their programs as a collection of idempotent and atomic tasks. They eliminate the need for the call stack and checkpoints by employing GOTO-style transitions among tasks, i.e., task-based control flow. However, this is an unstructured programming style that leads to programs that are not reusable and that are prone to bugs [28]. Task-based programming also leads to platform-dependent code since task sizes depend on the capacitor size of the platform. To the best of our knowledge, there is not any solution in the literature that can perform automatic task decomposition for task-based systems without programmer intervention. Recent work [7] provides tasks with parameters and continuation passing [12] via closures, which enables reusable code by delivering the control flow in a structured way, similar to function calls. However, it also leads to the platform-dependent code because of static task sizes.

B. Our Differences

Table I provides a comparison of our work with the state of the art. We propose an intermittent computing solution composed of a virtual machine (PureVM), a programming language (PureLANG), and a compiler. We design PureVM to abstract the intermittent execution details and give the programmer a continuously powered view of the target device. This abstraction provides platform-independent code via its multiple compilers for multiple hardware. PureLANG is the software interface of PureVM. PureLANG programs are translated into a sequence of *primitive functions*, which are the smallest computation blocks that PureVM executes atomically despite power failures. Thanks to this two-layered abstraction, our work overcomes all mentioned challenges of intermittent computing (i.e., A1–A3 and B1 and B2).

TABLE I
STATE-OF-THE-ART RUNTIMES AND MENTIONED CHALLENGES

Runtime	Type	A1	A2	A3	B1	B2
QuickRecall[19], Mementos[18]	Checkpt.	✗	✗	✗	✗	✓
DINO[9], Ratchet[22], Chinchilla[14]	Checkpt	✓	✗	✗	✗	✓
Samoyed[25]	Checkpt.	✓	✓	✗	✗	✓
TICS[5], CatNap[24]	Checkpt.	✓	✗	✓	✗	✓
Chain[6], Alpaca[16], Mayfly[26], Coala[27] Rehash [29]	Task-based	✓	✗	✗	✗	✗
PureMEM[7]	Task-based	✓	✗	✗	✗	✓
Coati[11], InK[4]	Task-based	✓	✗	✓	✗	✗
This Work (PureVM/PureLANG)	Virtual Machine	✓	✓	✓	✓	✓

III. PURELANG LANGUAGE

PureLANG is a statically typed and event-driven programming language. Programmers develop PureLANG applications via objects and functions that operate on them, and do not reason about intermittent execution. PureLANG employs continuation-passing style [12] where the program control flow is passed explicitly via *continuations*. Each function (or expression) takes one *flow-in object* in addition to its parameters and passes one *flow-out object* to the next function (or expression) that handles the rest of the computation. Therefore, each continuation contains a reference to a flow-in object, a set of object references as parameters, and a function reference to be applied. Events are continuations that are created by interrupt handlers.

PureVM persists continuations in nonvolatile memory to ensure forward progress. The overhead of persisting a continuation is static (since it contains only a function reference and a certain number of object references) compared to persisting the call stack whose size is dynamic in procedural languages. During program execution, PureVM applies the function on the flow-in object by using the information stored in the current continuation. Since PureLANG functions always operate on objects (kept in nonvolatile memory), PureVM can track the updates on these objects to preserve data consistency despite power failures.

A. PureLANG Types and Primitive Functions

Primitive functions are the high-level instructions, which execute atomically and form the interface between the PureLANG and PureVM. As an analogy with task-based systems [4], [6], [11], [16], [26], [29], primitive functions are the atomic tasks that are reusable building blocks. A PureLANG program is a composition of these atomic blocks.

Parameteric and Arrow Type: PureLANG has built-in object types `Int`, `Float`, `Bool`, and `Void`, which are reference types that point an address in memory. Moreover, PureLANG offers *parametric types* that are used to specify the type later. As an example, the `select` primitive function (Fig. 2, lines 1–3) returns one of two objects `t` or `f` of the same parametric

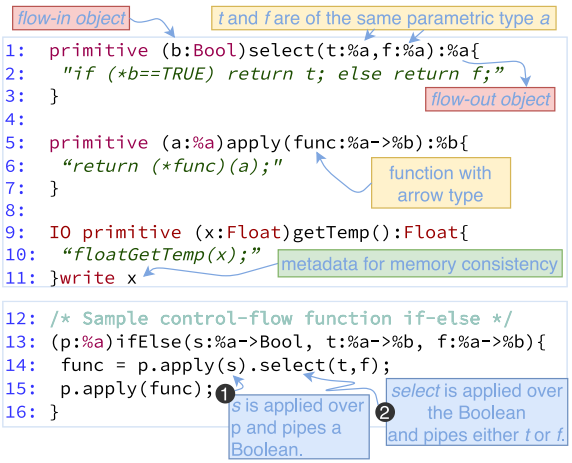

Fig. 2. Example PureLANG primitive and control-flow functions. The control-flow only depends on two primitive functions *apply* and *select* in PureLANG.

```
1: primitive (x:Int)add(y:Int, z:Int):Int{
2:   " *x = *y + *z;
3:     return x;     "
4: } write x
```

Fig. 3. `add` primitive function for `Int` type.

type `%a` as a flow-out object. The objects referenced by `t` and `f` can be of any type but same. The primitive function `select` makes the decision based on the value of the Boolean flow-in object `b`. Functions with the parametric type (also known as *parametric polymorphism*) eliminate code duplication. Without parametric polymorphism, `select` method needs different implementations for different types. As another example, the primitive function `apply` (Fig. 2, line 5) applies the given function on the flow-in object of parametric type `%a` and returns a flow-out object of parametric type `%b`. It also takes a function reference `func` as a parameter, which takes an object of parametric type `%a` and returns an object of parametric type `%b`. This is indicated by using *arrow type* decleration depicted as `%a->%b`. In the body of the primitive function (Fig. 2, line 6), `func` is called by passing the flow-in object `a`. Note that `func` returns an object of type `%b`, which is compatible with the flow-out object type of `apply`. It is worth indicating that every built-in type has its arithmetic operations—see `add` primitive function for `Int` type in Fig. 3. Primitive functions can implement any arithmetic and logic operation. Moreover, user-defined primitive functions are also possible.

IO Primitives: PureLANG introduces IO primitive functions to eliminate control-flow inconsistencies during I/O operations (A2). `IO` metadata (e.g., see `getTemp` function in Fig. 2, line 9) help the PureLANG compiler to handle these operations differently. The compiler splits PureLANG code blocks with IO primitives into three sections: 1) pre-IO primitive; 2) IO primitive itself; and 3) post-IO primitive. After each section executes, PureVM takes control to persist computational state, which ensures the atomic execution of the IO primitive.

Type Checking: Arrow and parametric type declarations help the PureLANG compiler for type inference and type checking. While decomposing the program into its primitive functions, the PureLANG compiler performs type checking by using input and output type metadata to eliminate (B2)-type bugs. The compiler also infers the variable types automatically when the programmer does not declare them explicitly.

Resolving W.A.R. Dependencies: Primitive functions also specify a metadata concerning write operations on objects. As an example, the *write* in the definition of `getTemp` function tells the compiler that this function modifies the flow-in object `x`. While decomposing the program into its primitive functions, the compiler can resolve W.A.R. dependencies using this metadata. This situation helps PureVM to execute the intermittent program correctly by preserving the memory consistency (to ensure A1). Considering the target PureVM implementation, PureLANG compiler instruments the bodies of the functions by inserting the necessary undo logging or redo logging code explained in Section IV.

B. PureLANG Statements and Control Flow

Since PureLANG employs structured programming, complex expressions are formed by composing other expressions (and primitive functions). The dot operator (`.`) enables expression composition as long as the output type of subexpression is compatible with the input type of the following subexpression. The last statement in a function body determines the output object that should be compatible with the output type of the function. Thanks to the continuation-passing style of the language, all statements forming the complex behavior of a function execute in order. Therefore, there is no need for the PureVM to check branches and early exits.

Control Flow: In PureLANG, every function related to the control flow is a composition of `select` and `apply` primitive functions. For example, `ifElse` function (Fig. 2, lines 14–16) enables a conditional branch by invoking the `apply` and `select` primitive functions in order. The first parameter `s` is a function that takes an object of parametric type `%a` and returns a Boolean object. First, the function `s` is applied on the flow-in object `p`, which pipes a Boolean object (i.e., `p.apply(s)` in line 11, which returns Boolean). Then, the returned object becomes a flow-in object for the `select` primitive, which returns one of the functions from `t` and `f` by considering the flow-in Boolean object. The returned function object is assigned to the variable `func`. Then, `func` is applied to the flow-in object `p`, and an object of `%b` type is returned (line 12). `select` and `apply` primitive functions, and the anonymous functions are enough to make PureLANG Turing-complete. Anonymous function is a language construct declaring continuation. `apply` primitive runs the continuations. `select` primitive, dynamically, selects the continuation (branch) to be applied. All other control-flow functions are the composition of select and apply. The implementations of `if` and `while` statements are presented in Fig. 4. The other control-flow functions in the PureVM standard library are `switch`, `for`, `map`, `reduce`, and `filter`. We omit their listings due to the interest of space.

C. Putting Things Together: Sample-Sensing Application

Fig. 5 presents an event-driven air monitoring application, which includes an application source code and a configuration

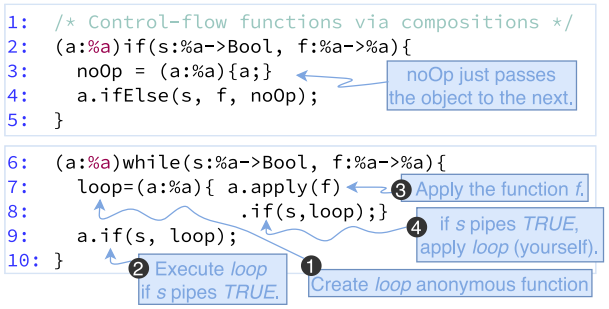

Fig. 4. Other control-flow functions `if` and `while`.

```
CONFIGURATION AirConditioningApp{
 boot-handler   Void{}.boot()
 reboot-handler Void{}.reboot()
 event-handler Float{}.control()
 ... /* other handlers */
 platform MSP430FR, runtime rewinding-vm, page 64B ...
}
```

```
1:  state = Int{IDEAL}; /* global object */
2:  ...
3:  /* reboot event handler */
4:  (a:Void)reboot(){
5:    a.initMCU();
6:  }
7:  /* control event handler (for heating) */
8:  (h:Float)control(){
9:    h.ifElse(isCold,heaterOn,heaterOff);
10:    ...
11: }
12: /* returns TRUE t < LOWER_TEMP */
13: isCold   =(t:Float){t.lessThan(LOWER_TEMP);};
14: heaterOn =(t:Float){state.set(COLD).AC_heat();};
15: ...
16: /* timer interrupt to sample temperature */
17: #interrupt isr_timer(){
18:   addEventQ(control,readTemp());
19: }
```

Fig. 5. Sample monitoring application in PureLANG. The application code contains all semantics of the computation, PureVM settings defining the events of the program and platform-specific attributes. Interrupt code is for receiving the values from environment (e.g., sensors).

file. PureLANG compiler (implemented using Xtext [30]) produces C code from the given PureLANG program. The generated code includes a single C source file and its header. The source file also contains the implementation of the target PureVM. The PureLANG compiler requires a configuration file, which mainly contains the list of event handlers, the name of the target hardware platform, and some specific parameters of the selected PureVM implementation (such as nonvolatile memory size and the size of the event queue).

The application code contains the objects, methods, and interrupt handlers. The event handlers `boot`, `reboot`, and `sleep` (which is not shown in the figure), are mandatory. The `boot` event occurs the first time the computer boots after being deployed. The `reboot` event occurs after recovery from a power failure, which triggers the reboot handler that restores the state of the computation. PureVM triggers the `sleep` handler when there is no event to be processed, which puts the processor in a low-power mode to save energy. The timer interrupt handler (lines 17–19) adds an event to the event queue of PureVM by calling `addEventQ` method with the sensed temperature value (via `readTemp`) and the corresponding event handler (which is `control` in this case) as parameters. PureVM processes this event by calling the `control` event handler (lines 8–10), which processes the events generated from the timer interrupt service routine. Inside this routine, the heater is turned on or off based on the received temperature value (line 9).

IV. PureVM Intermittent Virtual Machine

PureVM is a single-input/single-output system that specifies a runtime environment for event-driven intermittent programming. PureLANG programs execute, without any modification, on different hardware and runtime environments conforming PureVM specification.

PureVM specification comprises an event queue, a nonvolatile object memory, and a continuation executed by its runtime engine. PureVM pushes the events generated by the interrupt service routines to the event queue. PureVM removes the event at the head of the event queue and creates a continuation in object memory using that event. As mentioned, continuation represents the control state of the computer program, and it consists of a set of objects and methods to be applied. Running a continuation may create/return another continuation. PureVM runs the continuations until there is no returned continuation. When there is no event to consume in the queue, PureVM sleeps until an interrupt generates an event.

PureVM state (which represents the computational state) is composed of the events in the queue, the continuation of the running event, and the global objects. The object memory region in nonvolatile memory maintains the global objects and the running continuation. Before calling the next function (i.e., running the subsequent continuation), PureVM can decide to persist the state in object memory to preserve forward computation and not to lose the intermediate results in case of power failures.

PureVM specifies the artifacts (event buffer, object memory, runtime engine, and running program) and their relationships abstractly. For example, the event buffer can be implemented as a regular first-in–first-out (FIFO) queue or a priority queue, as long as the system's single-input behavior is not violated. Different design choices can lead to different PureVM implementations. In the following section, we describe RewindingVM, which is our main PureVM implementation.

A. RewindingVM: Undo-Logging PureVM

RewindingVM stores the execution context in the continuation stack and keeps the object memory consistent across power failures via the undo-logging mechanism. Events in the event queue, global objects, and continuation stack in the object memory represent the computational state. The metadata about the continuation stack are stored in the *runtime data* region of the object memory.

Event Handling: RewindingVM provides a queue for event buffering, which holds the event objects and the methods that need to be applied to these objects (*Event Queue* in

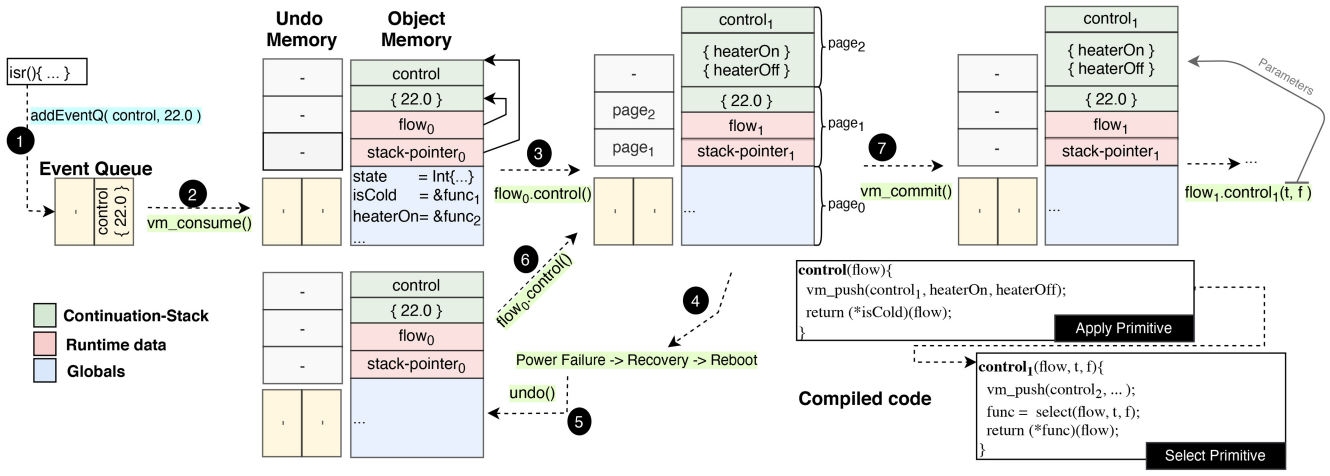

Fig. 6. Steps taken by RewindingVM to execute the `control` event described in Fig. 5. In the lower right corner of the figure, a part of the compiler generated code is presented. The compiler splits the `control` method into primitive functions and then recomposes into its basic blocks.

Fig. 6). RewindingVM runtime engine stores the continuations in a stack in object memory (*Continuation Stack* in Fig. 6). The runtime engine starts event execution by copying the event object (event data) and event method (event handler) to the continuation stack. It is worth mentioning that RewindingVM reduces the event handling to a single-producer-single-consumer problem, which eliminates the race conditions on the event buffer. During execution, the runtime engine pops a method from the continuation stack and runs this method on the flow object. Control passes to the method, which can modify the objects.

Undo-Log and Memory Consistency: The undo-log mechanism is activated when modifying objects to preserve memory consistency. The modifications on the objects are done only by calling primitive functions. Every primitive function calls the runtime engine's log function before modifying an object. The object memory comprises blocks called pages. The programmer, for efficiency, may configure the page sizes of the object memory. The log function copies the original page to undo-log memory before any modification on that page. When RewindingVM reboots after a power failure, it copies all the logged pages (including the pages of runtime data) into their corresponding pages in the object memory and continues the program. This mechanism ensures the consistency of the object memory by eliminating partial modifications.

Forward Progress and Execution Flow: The method being executed can push other methods to the continuation stack. The method execution finally returns an object and gives the control back to the runtime engine. The runtime engine saves the returned object as a flow object in the runtime data region of the object memory. The runtime atomically clears the undo-log memory as the last operation (i.e., commit operation). In this way, the runtime keeps memory consistent and guarantees the forward progress of the program.

I/O Handling: The PureLANG compiler already splits the program code blocks with I/O primitive functions into three sections, as described in Section III-A. Since this strategy ensures the atomic execution of I/O operations, the RewindingVM does not treat I/O operations in a specific way.

RewindingVM Compiler Optimizations: By default, RewindingVM executes one basic block in an execution cycle, which might be composed of several primitive functions. We implemented two compiler optimizations for RewindingVM: 1) merging more basic building blocks and 2) some loop optimizations to reduce PureVM overheads such as repetitive undo-logging (i.e., page copy) operations. With block optimizations, PureVM can execute multiple basic blocks in one execution cycle to reduce the persisting overhead. With loop optimizations, PureVM can execute an anonymous function multiple times and pack more loop execution into one execution cycle without repeating undo-logging operations. Programmers can modify the application configuration file to enable optimizations and indicate the maximum number of iterations to bypass PureVM logging operations within loops.

1) Example RewindingVM Execution: Fig. 6 shows how RewindingVM handles the `control` event described in Fig. 5. In the first step, the interrupt service routine adds the address of the control method to the event queue along with the sensed temperature value (which is a floating-point value of 22.0). In the second step, the VM copies the event from the event queue to the object memory (depicted as control and 22.0 in the continuation stack) and sets the runtime data of the flow-in object and stack pointer (depicted as flow_0 and stack pointer_0). flow_0 points the active object (i.e., {22.0}), which is the flow-in object for the function at the top of the continuation stack (i.e., control pointed by stack pointer_0). In the lower right corner of Fig. 6, a part of the compiler-generated code is presented. The compiler splits the `control` method into primitive functions and then recomposes into its basic blocks. As can be seen from the figure, the recomposed version of the `ifElse` method (line 10 in Fig. 5) is fragmented into two continuations that represent `apply` and `select` primitives. In the third step, the VM removes the first method from the continuation stack (`control` function in the lower right corner of Fig. 6) and runs it. Since this process updates the runtime data and the stack of the nonvolatile memory, VM copies the original values of page_1 and page_2 to undo memory

before the update. It is worth mentioning that the $page_0$ is not logged because there is no modification on any global variable in that particular execution period. The fourth step in the figure shows that the computer restarts after a power failure. In the fifth step, the VM calls the undo function because it detects that the undo memory is not empty. It brings the nonvolatile object memory to the last consistent state by copying $page_1$ and $page_2$ from log memory to main memory and then removing the pages from the log atomically. Then, in step 6, the method on the stack runs, as described in step 3. In the seventh step, the undo memory is cleared by committing it. The forward progress of the computation is ensured by executing the operations `vm_consume`, `vm_commit`, and `undo` atomically. In the next steps (not shown in the figure), the remaining methods in the continuation stack execute. They pass the returned flow objects to the next methods till the continuation stack becomes empty.

The logging mechanism provides a messaging environment between the power and execution cycles. If the log is not empty when the power cycle starts, undo function puts the computation into the last consistent state, and the computing cycle restarts at that point. The computing cycle logs the original pages before updating them and then operates on the main memory. Clearing the logs with the commit command indicates that the main memory has reached a consistent point, and there is no need for undo operation.

B. Other PureVM Implementations

We also implemented two different PureVM versions, named JustInTimeVM and TestVM. JustInTimeVM does not contain an undo log memory and requires hardware support to capture an interrupt when the voltage value on the capacitor crosses the lower threshold voltage. This interrupt persists the computational state in nonvolatile memory and puts the runtime into sleep mode to stop computation. This strategy prevents memory inconsistencies without the need for an undo log. JustInTimeVM's overhead is lower than RewindingVM's since JustInTimeVM does not include page copy operations between log memory and object memory. We implemented TestVM to test any PureLANG program on a personal computer with continuous power. This implementation allows us to test the correctness of the program logic without loading the code on a microcontroller.

V. Evaluation

We evaluated our PureVM implementations considering three compute-intensive applications cuckoo filter (CF), activity recognition (AR), and bit count (BC). These applications are used as the de facto benchmarks by most of the earlier studies on intermittently powered devices [4]–[6], [16]. CF stores and reads an input data stream using a CF with 128 entries and searches the inserted values. AR classifies accelerometer data, computing the mean and standard deviation of a window of accelerometer readings to train a nearest-neighbor model to detect different movements. We used a window size of three and read 128 samples from each class (shaking or stationary) in the training phase. BC counts the number of 1 s in a bitstream with seven different methods that are executed 100 times each.

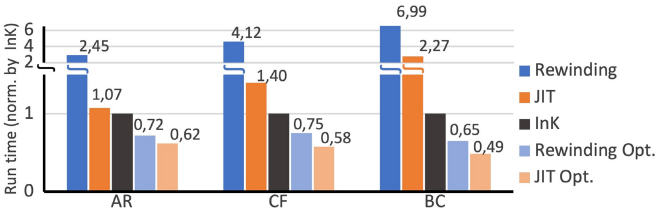

Fig. 7. Normalized execution times of AR, CF, and BC benchmarks with InK, RewindingVM, and JustInTimeVM under continuous power.

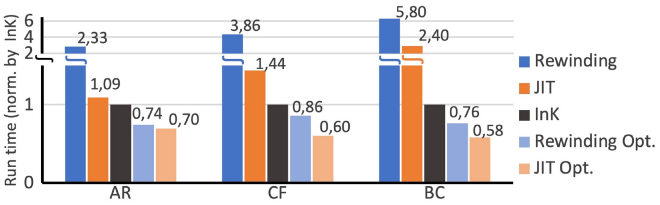

Fig. 8. Normalized execution times of AR, CF, and BC benchmarks with InK, RewindingVM, and JustInTimeVM under RF energy harvesting scenario.

We compiled these applications using the RewindingVM and JustInTimeVM compilers and executed them on the MSP430FR5994 evaluation board [31]. To power MSP430 evaluation boards intermittently, we used Powercast TX91501-3W transmitter [32], which emits radio-frequency (RF) signals at 915-MHz center frequency. P2110-EVB receiver, connected to our MSP430FR5994 evaluation board, harvests energy from these RF signals. We placed the P2110-EVB receiver at a distance of 60 cm from the RF transmitter. We used the 220-mF capacitor mounted on the MSP430FR5994 board. In this setup, the transmitted energy was quite stable. The harvested power from the transmitter was approximately 7.9 mW. The capacitor charging time was approximately 11 s.

A. Execution Time Overhead

We evaluated the performance of PureVM implementations of the benchmarks on harvested energy and continuous power by considering their execution times. For comparison, we used their InK [4] implementations since InK is one of the de facto task-based systems for intermittent computing. We directly used InK-based implementations of the benchmarks (CF, AR, and BC) from the InK repository [33]. Since the tasks have fixed sizes in InK, they may exceed the energy buffer of the device, or they may perform inefficiently due to frequent task transitions (causing redo logging of all shared variables to preserve memory consistency). To recalibrate the size of the tasks, the programmer must recompose all tasks for the new device. This way of programming limits the code portability.

Figs. 7 and 8 show the normalized execution times of the benchmarks with InK and PureVM (RewindingVM, RewindingVM optimized, JustInTimeVM, and JustInTimeVM optimized) on continuous power and intermittent execution on RF-power. The compiled code of RewindingVM and JustInTimeVM can run on the devices with a minimal energy buffer because the compiler recomposes primitive

TABLE II
NORMALIZED MAXIMUM BLOCK SIZES (I.E., EXECUTION TIME) OF THE APPLICATIONS WITH RESPECT TO THE AVERAGE EXECUTION TIME OF ADD PRIMITIVE FUNCTION

App.	InK	Rew.	JIT	Rew.Opt.	JIT.Opt.
AR	9.8	22.9	17.3	2443	2256.2
BC	1.9	9.9	2.4	114.9	88.5
CF	6.2	23.2	9.4	791.7	609.7

functions regarding basic blocks. Recomposing the code into its basic blocks leads to small continuations and consequently more (continuation) stack operations in RewindingVM and JustInTimeVM, and more undo-logging operations in RewindingVM. Optimized versions of the applications are composed of continuations with more operations. Therefore, these applications run more efficiently compared to InK apps. We applied the loop optimizations to all loops and branch optimizations to some branch method invocations in these applications.

Table II shows the normalized block sizes in the applications with respect to the execution time of the `add` primitive function in PureVM. In other words, it shows how many primitive functions (on average) fit within these blocks. It is worth mentioning that there is an almost linear relationship between execution time and energy cost for a block that does not contain any IO operation. As indicated in Table II, after compiler optimizations, the block size of AR is almost three times greater than that of CF. The reason is that loop optimization is more effective in AR since it has more loops, which can be combined to form bigger blocks. Therefore, the compiler could pack more loop execution into one execution cycle.

B. Different Capacitor Sizes

As justified by Table II, PureLANG compiler can automatically generate intermittent programs with different block sizes without programmer intervention. This feature ensures the platform independence of the developed applications since programs do not need to be refactored for each capacitor size. Contrarily, task-based implementations require a manual task decomposition for each platform, which makes programs capacitor size, and in turn, hardware dependent.

To further demonstrate how PureLANG code is portable across different capacitor sizes, we ran our applications without using all the energy in the capacitor in the MSP430FR5994 board. We emulated different capacitor sizes by triggering an interrupt when the capacitor voltage reaches the threshold value and running an empty loop runs until brownout. First, we reduced the usable capacitance value by about 6.4 times. In this case, the optimized versions of AR code encountered nontermination while nonoptimized versions of AR could still terminate. Similarly, optimized CF codes could not terminate when reducing the usable capacitance by about 12.8 times. This shows that even different applications have different minimum capacitor size requirements. However, nonoptimized versions in PureVM always terminate. Therefore, the programmer can compromise between the compiler optimizations and capacitor size.

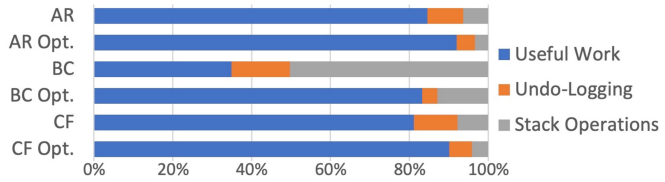

Fig. 9. RewindingVM overhead, split per operation.

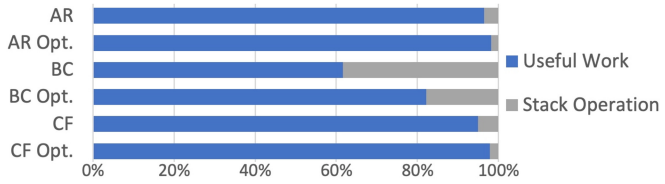

Fig. 10. JustInTimeVM overhead, split per operation.

TABLE III
MEMORY CONSUMPTION FOR THREE BENCHMARK APPLICATIONS WRITTEN IN INK AND PUREVM

App.	Memory(B)	InK	Rew.	Rew.Opt.	JIT	JIT.Opt.
AR	.text	3822	8624	6810	7950	6136
	.data	4980	694	694	420	420
BC	.text	3518	11258	9928	10714	9378
	.data	4980	882	882	676	548
CF	.text	2708	11980	10014	11302	9350
	.data	5213	886	694	804	420

C. PureVM Point-to-Point Overheads

Figs. 9 and 10 show the useful work and overheads (i.e., undo-logging and stack operations) of corresponding runtimes on continuous power. RewindingVM and JustInTimeVM use a stack to run the continuations. When there is more branching in the program, the continuation stack operation creates more overhead because RewindingVM and JustInTimeVM run the basic blocks in one cycle, which means every branch needs a continuation stack operation. In BC, the overhead of stack manipulation is higher due to many branch operations, which also causes the worst performance compared to other benchmarks. On the other hand, loop optimizations in BC had the most impact on the performance compared to other benchmarks, increasing the performance by a factor of 10.

Before modifying a page that has not been logged before, the undo-logging mechanism is triggered, which introduces page search and page copy overheads. Since the log memory size is small for the benchmarks, we chose a sequential page search in the log memory. Apparently, the page size affects the undo-logging performance. The virtual machine configuration file of RewindingVM and JustInTimeVM contains a page size setting. The page size of 32 and 64 bytes gave the best performance in these applications.

D. PureVM Memory Overheads

Table III shows the memory overheads of InK and PureVM implementations. Since primitive functions are translated into C codes, PureVM programs have larger code sizes than InK programs. InK uses global shared variables for communication among functions, and hence it has larger data memory

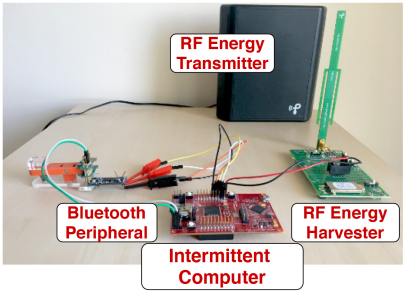

Fig. 11. Our testbed deployment used to evaluate our HVAC application.

than PureVM. The programmer configures the data memory of RewindingVM and JustInTimeVM via the configuration file. The stack size required by virtual machines contributes to their data memory requirements. The code size increase is the cost which PureVM implementations pay for their performance and platform independence. However, overall results show that the memory requirement of PureVM is comparable with InK's memory requirement.

E. Case Study: Heating, Ventilation, and Air Conditioning Controller

As a case study to demonstrate the applicability of PureVM, we developed an air condition controller for home automation (Fig. 11). The goal is to get the room temperature frequently enough and send a message to home automation to keep the room temperature in the ideal temperature range. The application uses the PureVM event mechanism to reduce energy consumption as much as possible: after controlling the room temperature, the application switches to sleep mode. A reboot from a power failure or a timer interrupt (with 30-s intervals) triggers the program, which estimates the room temperature using an analog–digital converter configured to the internal temperature sensor of the microcontroller. If the room temperature is not ideal, the application starts asynchronous communication via Nordic nRF52832 [34] (which supports BLE) using interrupts over the serial peripheral interface. We ran this application for 3 h. We observed that the temperature of the environment was measured 294 times, and 22 BLE advertisement messages were sent to the heating, ventilation, and air conditioning (HVAC) system. During the entire run, there were 18 power failures and recovery. It is worth indicating that during our experiments, nRF42832 was intermittently powered since it was powered by the MSP430FR5994 development kit through the 0.22 F capacitor on the board. The BLE communication was implemented by using nRF42832 driver (a C library). The SPI messaging between nRF42832 and PureVM was implemented using PureLANG by events (input messages for PureVM from nRF42832) and IO primitives (output messages from PureVM to nRF42832).

VI. Conclusion and Future Work

In this work, we introduced a new virtual machine (PureVM) that abstracts a transiently powered computer and a new continuation-passing-style programming language (PureLANG) used to develop programs that run on PureVM. This two-layer structure provided a loosely coupled architecture that facilitates the development of platform-independent and reusable event-driven sensor applications. We believe that this is a significant attempt to virtualize intermittent computing.

As followup work, we plan to add new language constructs to PureLANG to handle the expiration of sensor readings. Due to long charging times after power failures, sensed data might lose its validity. In this case, the sensor value becomes useless and can be discarded. While such a requirement is absent in continuous computing, it exists in intermittent computing [5], [26]. We also plan to port PureVM to different ultralow-power microcontrollers and introduce more sophisticated compiler optimizations. As of now, PureVM does not implement any task scheduling mechanism. We leave integrating scheduling mechanisms, e.g., real-time scheduling of tasks [24], [35], to PureVM as future work.

References

[1] J. D. Hester and J. Sorber, "Flicker: Rapid prototyping for the batteryless Internet-of-Things," in *Proc. 15th ACM Conf. Embedded Netw. Sensor Syst.*, 2017, pp. 1–13.
[2] K. S. Yıldırım, R. Carli, and L. Schenato, "Safe distributed control of wireless power transfer networks," *IEEE Internet Things J.*, vol. 6, no. 1, pp. 1267–1275, Feb. 2019.
[3] M. Nardello, H. Desai, D. Brunelli, and B. Lucia, "Camaroptera: A batteryless long-range remote visual sensing system," in *Proc. 7th Int. Workshop Energy Harvest. Energy Neutral Sens. Syst.*, 2019, pp. 8–14.
[4] K. S. Yıldırım, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester, "InK: Reactive kernel for tiny Batteryless sensors," in *Proc. 16th ACM Conf. Embedded Networked Sensor Syst.*, 2018, pp. 41–53.
[5] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawełczak, "Time-sensitive intermittent computing meets legacy software," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2020, pp. 85–99.
[6] A. Colin and B. Lucia, "Chain: Tasks and channels for reliable intermittent programs," in *Proc. ACM SIGPLAN Int. Conf. Object Orient. Program. Syst. Lang. Appl. (OOPSLA)*, 2016, pp. 514–530.
[7] C. Durmaz, K. S. Yildirim, and G. Kardas, "PureMEM: A structured programming model for transiently powered computers," in *Proc. 34th ACM/SIGAPP Symp. Appl. Comput.*, 2019, pp. 1544–1551.
[8] A. Colin and B. Lucia, "Termination checking and task decomposition for task-based intermittent programs," in *Proc. 27th Int. Conf. Compiler Construction*, 2018, pp. 116–127.
[9] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2015, pp. 575–585.
[10] A. Colin, E. Ruppel, and B. Lucia, "A reconfigurable energy storage architecture for energy-harvesting devices," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2018, pp. 767–781.
[11] E. Ruppel and B. Lucia, "Transactional concurrency control for intermittent, energy-harvesting computing systems," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2019, pp. 1085–1100.
[12] G. J. Sussman and G. L. Steele, "SCHEME: A interpreter for extended lambda calculus," *High. Order Symb. Comput.*, vol. 11, no. 4, pp. 405–439, 1998.
[13] L. T. van Binsbergen, E. Scott, and A. Johnstone, "Purely functional GLL parsing," *J. Comput. Lang.*, vol. 58, no. 1, pp. 1–17, 2020.
[14] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2018, pp. 129–144.
[15] B. Ransford and B. Lucia, "Nonvolatile memory is a broken time machine," in *Proc. Workshop Memory Syst. Perform. Correctness*, 2014, pp. 1–3.
[16] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," in *Proc. ACM Program. Lang.*, vol. 1, Oct. 2017, pp. 1–96.

[17] M. Surbatovich, L. Jia, and B. Lucia, "I/O dependent idempotence bugs in intermittent systems," in *Proc. ACM Program. Lang.*, vol. 3, 2019, pp. 1–31.
[18] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on RFID-scale devices," in *Proc. 16th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS XVI)*, 2011, pp. 159–170.
[19] H. Jayakumar, A. Raha, W. S. Lee, and V. Raghunathan, "QUICKRECALL: A HW/SW approach for computing across power cycles in transiently powered computers," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 12, no. 1, pp. 1–8, Jul. 2015.
[20] D. Balsamo *et al.*, "Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 12, pp. 1968–1980, Dec. 2016.
[21] J. Zhan, G. V. Merrett, and A. S. Weddell, "Exploring the effect of energy storage sizing on intermittent computing system performance," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 3, pp. 492–501, Mar. 2022.
[22] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *Proc. 12th USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, Savannah, GA, USA, Apr. 2016, pp. 17–32.
[23] N. A. Bhatti and L. Mottola, "HarvOS: Efficient code instrumentation for transiently-powered embedded sensing," in *Proc. 16th ACM/IEEE Int. Conf. Inf. Process. Sensor Netw. (IPSN)*, Apr. 2017, pp. 209–220.
[24] K. Maeng and B. Lucia, "Adaptive low-overhead scheduling for periodic and reactive intermittent execution," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2020, pp. 1005–1021.
[25] K. Maeng and B. Lucia, "Supporting peripherals in intermittent systems with just-in-time checkpoints," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2019, pp. 1101–1116.
[26] J. Hester, K. Storer, and J. Sorber, "Timely execution on intermittently powered batteryless sensors," in *Proc. 15th ACM Conf. Embedded Netw. Sensor Syst.*, 2017, pp. 1–13.
[27] A. Y. Majid *et al.*, "Dynamic task-based intermittent execution for energy-harvesting devices," *ACM Trans. Sensor Netw.*, vol. 16, no. 1, pp. 1–24, 2020.
[28] E. W. Dijkstra, "Letters to the editor: Go to statement considered harmful," *Commun. ACM*, vol. 11, no. 3, pp. 147–148, 1968.
[29] A. Bakar, A. G. Ross, K. S. Yildirim, and J. Hester, "REHASH: A flexible, developer focused, heuristic adaptation platform for intermittently powered computing," *Proc. ACM Interact. Mobile Wearable Ubiquitous Technol.*, vol. 5, no. 3, pp. 1–42, 2021.
[30] M. Eysholdt and H. Behrens, "XText: Implement your language faster than the quick and dirty way," in *Proc. ACM Int. Conf. Companion Object Oriented Program. Syst. Lang. Appl. Companion*, 2010, pp. 307–309.
[31] "Texas Instruments. MSP430FR5994 launchpad development kit." 2021. [Online]. Available: http://www.ti.com/tool/MSP-EXP430FR5994
[32] "Powercast corp." 2021. [Online]. Available: https://www.powercastco.com/products/development-kits/
[33] K. S. Yıldırım, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester. "TUDSSL/ink." 2019. [Online]. Available: https://github.com/tudssl/ink
[34] "Nordic Semiconductor. nRF52832 system-on-chip." 2021. [Online]. Available: https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF52832
[35] M. Karimi, H. Choi, Y. Wang, Y. Xiang, and H. Kim, "Real-time task scheduling on intermittently-powered Batteryless devices," *IEEE Internet Things J.*, vol. 18, no. 17, pp. 13328–13342, Sep. 2021.

Çağlar Durmaz received the M.Sc. degree in computer engineering from Dokuz Eylul University, Izmir, Turkey, in 2006. He is currently pursuing the Ph.D. degree with the International Computer Institute, Ege University, Izmir.

He is the Founder of Integra ICT Corporation, Izmir, Turkey. His research interests include programming languages, model-driven engineering, embedded systems, and intermittent computing.

Kasım Sinan Yıldırım (Member, IEEE) received the M.Sc. and Ph.D. degrees in computer engineering from Ege University, Izmir, Turkey, in 2006 and 2012, respectively.

He is currently an Assistant Professor with the Department of Information Engineering and Computer Science, University of Trento, Trento, Italy. His research interests include embedded systems, intermittent computing, wireless networks, and distributed algorithms.

Geylani Kardas received the B.Sc. degree in computer engineering and the M.Sc. and Ph.D. degrees in information technologies from Ege University, Izmir, Turkey, in 2001, 2003, and 2008, respectively.

He is currently an Associate Professor with the International Computer Institute (ICI), Ege University, and the Head of the Software Engineering Research Laboratory, ICI. His research interests include agent-oriented software engineering, model-driven engineering, domain-specific (modeling) languages, and low-code software development. He has authored or coauthored over 100 peer-reviewed papers in these research areas.

Dr. Kardas is an Associate Editor of the *Journal of Computer Languages* (Elsevier).