Virtualizing Intermittent Computing
 Çağlar Durmaz, Kasım Sinan Yıldırım, IEEE Member, 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 programs that are hardware-dependent and not reusable. This paper 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—Energy Harvesting, Batteryless IoT, Intermittent Computing, Domain-specific Language, 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 radiofrequency [1], [2], [3]. Batteryless devices store the harvested ambient energy into a tiny capacitor that powers the microcontroller and 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, 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 non-volatile 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 [6] or tasks [6]) to (i) maintain the forward progress of computation and (ii) 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.

Çağlar Durmaz and Geylani Kardas are with the International Computer Institute, Ege University, Turkey. e-mail: {caglar.durmaz, geylani.kardas}@ege.edu.tr. Kasım Sinan Yıldırım is with the Department of Information Engineering and Computer Science, University of Trento, Italy. e-mail: kasimsinan.yildirim@unitn.it

![Fig. 1. Using PureLANG, a continuation-passing-style programming language, and PureVM, whose specification enables the execution of programs written in PureLANG, programmers do not reason about the intermittent execution and they develop programs portable across several hardware platforms.](image)

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.

Hardware Dependency. The active time of a batteryless device depends on its capacitor size and its power consumption, i.e., its hardware configuration [3]. Programmers might need to identify different burst boundaries to execute their programs on a new device with a different hardware configuration. 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 [9]. This situation increases the programming burden and makes programs error-prone.

In this paper, 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 paper 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 Figure 1).

2) **PureLANG Language.** We introduce PureLANG, a
continuation-passing-style programming language, intended to prevent programmers from reasoning about intermittent execution. PureLANG programs are translated into a set of re-composable and atomically-executed PureVM instructions.

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: computation and I/O related challenges (A1–A3) and programming challenges (B1–B2). We describe them as follows.

A1- Non-termination and Memory Consistency. Power failures hinder the forward progress of the computation, which leads to non-terminating programs [10]. Non-termination occurs since the device loses the intermediate results and restarts the computation from scratch upon each reboot. Power failures might also lead to memory inconsistencies [11]. To give an example, assume that a program is modifying the persistent variable `var` (i.e., a variable kept in non-volatile memory) by executing the code block `if(temp > limit) then alarm = true; else tempOk = true;`. 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 re-execution of this code will increment the variable `var` again. Therefore, another element of the variable 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., [12], [4]) is required to persist the program state non-volatile memory, to manage power failures, and to re-execute 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 [13]. 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 `alarm` based on the temperature reading `if(temp > limit) then alarm = true; else tempOk = true;`. If the temperature is less than a pre-defined limit, the variable `tempOk` will be set to true. If there is a power failure right after this operation and the program re-executes, the program might read another temperature value higher than the limit. In this case, the program will set the variable `tempOk` to true. At this point, both of the variables `alarm` and `tempOk` are true, which is logically incorrect [13].

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 [9].

B1- Platform Dependencies. The execution time of an intermittent program depends on several factors, such as available harvestable energy in the environment, 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 non-termination and ensure computational progress [10].

B2- Reuse 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 programs that are difficult to maintain.

A. The 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., [14], [15], [16], [17]) persist the registers, global variables, and the call-stack of programs into non-volatile memory via programmer-inserted checkpoints to preserve the forward progress of computation (A1). In particular, due to the call stack’s dynamic nature (i.e., it grows and shrinks dynamically), the programmers need to place checkpoints carefully to eliminate non-termination. In particular, the energy required to execute the instructions between two checkpoints should not exceed the maximum energy stored in the capacitor. Therefore, checkpoint placement is platform-dependent and checkpointed programs are not reusable. There are studies (e.g., [18], [19], [5], [20], [21], [10]) 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 non-termination.

Task-based Systems. Task-based models (e.g., [6], [12], [4], [22], [23], [9]) 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 [24]. Task-based programming also leads to platform-dependent code since task sizes depend on the capacitor size of the platform. Recent work [7] provides tasks with parameters and continuation-passing [25] via closures, which enables reusable code by delivering the control flow in a structured way, similar to function calls. However, it also leads to platform-dependent code because of static task sizes.
Figure 2. PureLANG example primitive and control-flow functions. The control-flow only depends on two primitive functions apply and select in PureLANG.

### 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. PureVM employs continuation-passing style [23] 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 non-volatile 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 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 non-volatile 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 [6], [4], [12], [9], [22], [26], primitive functions are the atomic tasks that are reusable building blocks. PureLANG programs are a composition of these atomic blocks.

#### Parametric 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 same parametric 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 \( \text{func} \) 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 declaration depicted as \( %a \rightarrow %b \).

In the body of the primitive function (Fig. 2 Line 6), \( \text{func} \) is called by passing the flow-in object \( a \). Note that, \( \text{func} \) returns an object of type \( %b \), which is compatible with the flow-out object type of apply.

#### IO Primitives

PureLANG introduces IO primitive function functions to eliminate control-flow inconsistencies during I/O operations (A2). IO metadata (e.g., see getTemp function in Fig. 2 Line 9) helps the PureLANG compiler to handle these operations differently. The compiler splits PureLANG code blocks with IO primitives into three sections: pre-IO primitive, IO primitive itself, and 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 meta-data 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 [V]

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 sub-expression is compatible with the input type of the following sub-expression. 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. Firstly, 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 variable func. Then, func is applied to the flow-in object p, and an object of %b type is returned (Line 12).

C. Putting Things Together: A Sample Sensing Application

Figure 3 presents an event-driven air monitoring application, which includes an application source code and a configuration file. PureLANG compiler (implemented using Xtext [27]) 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 non-volatile memory size, 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 19–21) 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 9–11), 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 10).

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 non-volatile 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 non-volatile 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: An 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 is 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. 4). RewindingVM runtime engine stores the continuations in a stack in object memory (Continuation Stack in Fig. 4). 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.

Undolog 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. The runtime atomically clears the undo-log memory as the last 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. This strategy already ensures the atomic execution of I/O operations. Therefore, the RewindingVM runtime engine does not treat I/O operations in a specific way.

RewindingVM Compiler Optimizations. We implemented two compiler optimizations for RewindingVM: (i) executing
several primitive functions as a block, and (ii) some loop optimizations to reduce PureVM overheads such as repetitive undo logging (i.e., page copy) operations. Programmers can modify PureVM application configuration file to enable optimizations.

1) An Example RewindingVM Execution.

Fig. 4 shows how RewindingVM handles the control event described in Fig. 3. 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). In the lower right corner of Fig. 4, 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. 3) 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. 4) and runs it. Since this process changes the runtime data and the stack of the non-volatile memory, it copies the pages to undo memory before any update. 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 non-volatile object memory to the last consistent state. Then, in step 6, the method on the stack runs, as described in step three. In the seventh step, the undo memory is cleared by committing it. 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.

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 non-volatile 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 ([5, 12, 6, 4]). CF stores and reads an input data stream using a cuckoo filter 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. BC counts the number of 1s in a bitstream with seven different methods that are executed 100 times each.

We compiled these applications using the RewindingVM and JustInTimeVM compilers and executed them on the MSP430FR5994 evaluation board [28]. To power MSP430 evaluation boards intermittently, we used Powercast TX91501-3W transmitter [29], 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 60cm from the RF transmitter.

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 [30]. 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 re-calibrate the size of the tasks, the programmer must recompose all tasks for the new device. This way of programming limits the code portability.

Fig. 5 and Fig. 6 show the normalized execution times of the benchmarks with InK, RewindingVM and JustInTimeVM under continuous power and RF energy harvesting scenario.
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 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 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.

B. PureVM Point-to-Point overheads

Fig. 7 and Fig 8 show the useful work and overheads (i.e., undo-logging and stack operations) of corresponding runtimes on continuous power. RewindingVM and JustInTimeVM uses 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 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.

C. PureVM Memory overheads

Table II 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 than PureVM. The programmer configures 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.

D. Case Study: Heating, ventilation, and air conditioning (HVAC) controller

As a case study to demonstrate the applicability of PureVM, we developed an air condition controller for home automation (Figure 9). 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-second 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 [31] (which supports BLE) using interrupts over the serial peripheral interface. We ran this application for 3 hours. We observed that the temperature of the environment was measured 294 times, and 22 BLE advertisement messages were sent to the HVAC system. During the entire run, there were 18 power failures and recovery.
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 for virtualizing intermittent computing. As follow-up 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], [22]. We also plan to port PureVM to different ultra-low-power micro-controllers 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 [23], to PureVM as future work.

References

[1] J. Hester and J. Sorber, “Flicker: Rapid prototyping for the batteryless internet-of-things,” in Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems, 2017, pp. 1–13.
[2] K. S. Yildirim, R. Carli, and L. Schenato, “Safe distributed control of wireless power transfer networks,” IEEE Internet of Things Journal, vol. 6, no. 1, pp. 1267–1275, 2018.
[3] M. Nardello, H. Desai, D. Brunelli, and B. Lucia, “Camaroptera: A batteryless long-range remote visual sensing system,” in Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems, 2019, pp. 8–14.
[4] K. S. Yildirim, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester, “Ink: Reactive kernel for tiny batteryless sensors,” in Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems. ACM, 2018, pp. 41–53.
[5] V. Korbek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawelczak, “Time-sensitive intermittent computing meets legacy software,” in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 85–99.
[6] A. Colin and B. Lucia, “Chain: Tasks and channels for reliable intermittent programs,” in Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016). Amsterdam, Netherlands: ACM, Oct. 30 – Nov. 4, 2016, pp. 514–530.
[7] C. Durmaz, K. S. Yildirim, and G. Kardas, “Puremem: a structured programming model for transiently powered computers,” in Proceedings of the 34th ACM SIGAPP Symposium on Applied Computing, 2019, pp. 1544–1551.
[8] A. Colin, E. Ruppel, and B. Lucia, “A reconfigurable energy storage architecture for energy-harvesting devices,” in Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, 2018, pp. 767–781.
[9] E. Ruppel and B. Lucia, “Transactional concurrency control for intermittent, energy-harvesting computing systems,” in Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2019, pp. 1085–1100.
[10] K. Maeng and B. Lucia, “Adaptive dynamic checkpointing for safe efficient intermittent computing,” in 13th {USENIX} Symposium on Operating Systems Design and Implementation (OSDI’18), 2018, pp. 129–144.
[11] B. Ransford and B. Lucia, “Nonvolatile memory is a broken time machine,” in Proceedings of the workshop on Memory Systems Performance and Correctness, 2014, pp. 1–3.
[12] K. Maeng, A. Colin, and B. Lucia, “Alpaca: Intermittent execution without checkpoints,” Proc. ACM Program. Lang., vol. 1, no. OOPSLA, pp. 96:1–96:30, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133920.
[13] M. Surbatovich, L. Jia, and B. Lucia, “I/O dependent idempotence bugs in intermittent systems,” Proceedings of the ACM on Programming Languages, vol. 3, no. OOPSLA, pp. 1–31, 2019.
[14] B. Ransford, J. Sorber, and K. Fu, “Mementos: System support for long-running computations on RFID-scale devices,” in Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS XVI). Newport Beach, CA, USA: ACM, Mar. 5–11, 2011, pp. 159–170.
[15] B. Lucia and B. Ransford, “A simpler, safer programming and execution model for intermittent systems,” in Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’15). Portland, OR, USA: ACM, Aug. 13–17, 2015, pp. 575–585.
[16] 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. 8:1–8:19, Jul. 2015.
[17] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-hashimi, G. V. Merrett, and L. Benini, “Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices,” vol. 35, no. 12, pp. 1968–1980, 2016.
[18] J. Van Der Woude and M. Hicks, “Intermittent computation without hardware support or programmer intervention,” in Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI’16). Savannah, GA, USA: ACM, Nov. 2–4, 2016, pp. 17–32.
[19] N. A. Bhatti and L. Mottola, “Harvos: Efficient code instrumentation for transiently-powered embedded sensing,” in 2017 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN). ACM/IEEE, April 2017, pp. 209–220.
[20] K. Maeng and B. Lucia, “Adaptive low-overhead scheduling for periodic and reactive intermittent execution,” in Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020, pp. 1005–1021.
[21] ——, “Supporting peripherals in intermittent systems with just-in-time checkpoints,” in Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2019, pp. 1101–1116.
[22] J. Hester, K. Storer, and J. Sorber, “Timely execution on intermittently powered batteryless sensors,” in Proceedings of the 15th ACM Conference on Embedded Networked Sensor Systems, 2017, pp. 1–13.
[23] A. Y. Majid, C. D. Donne, K. Maeng, A. Colin, K. S. Yildirim, B. Lucia, and P. Pawelczak, “Dynamic task-based intermittent execution for energy-harvesting devices,” ACM Transactions on Sensor Networks (TOSN), vol. 16, no. 1, pp. 1–24, 2020.
[24] E. W. Dijkstra, “Letters to the editor: go to statement considered harmful,” Communications of the ACM, vol. 11, no. 3, pp. 147–148, 1968.
[25] G. J. Sussman and G. L. Steele, “Scheme: A interpreter for extended lambda calculus,” Higher-Order and Symbolic Computation, vol. 11, no. 4, pp. 405–439, 1998.
[26] A. Bakar, A. G. Ross, K. S. Yildirim, and J. Hester, “Rehash: A flexible, developer focused, heuristic adaptation platform for intermittently powered computing,” Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, vol. 5, no. 3, pp. 1–42, 2021.
[27] M. Eysholdt and B. Behrens, “Xtest: implement your language faster than the quick and dirty way,” in Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, 2010, pp. 307–309.
[28] “Texas instruments. msp430fr5994 launchpad development kit.” 2021. [Online]. Available: http://www.ti.com/tool/MSP-EXP430FR5994
[29] “Powercast corp.” 2014. [Online]. Available: https://www.powercastco.com/products/development-kits/
[30] K. S. Yildirim, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester, “Tudsslink,” 2019. [Online]. Available: https://github.com/ndusslink
[31] “Nordic semiconductor. nrf52832 system-on-chip.” 2021. [Online]. Available: https://www.nordicsemi.com/Products/Low-power-short-range-wireless/NRF52832
[32] M. Karimi, H. Choi, Y. Wang, Y. Xiang, and H. Kim, “Real-time task scheduling on intermittently-powered batteryless devices,” IEEE Internet of Things Journal, 2021.