Towards Sustainable IoT Applications: Unique Challenges for Programming the Batteryless Edge

Arda Goknil
SINTEF Digital

Kasım Sinan Yıldırım
University of Trento

***Abstract*—The research on sustainable sensing applications relying on ambient energy resources is receiving attention to reduce the carbon dioxide (CO2) emissions and greenhouse effects of IoT applications. The advent of energy-harvesting technology and ultra-low-power computer systems has enabled intermittently powered, battery-free devices to operate using harvested ambient energy. Program execution on battery-free devices progresses in short bursts interleaved by recharge periods. This type of program execution requires new techniques to analyze, develop, and verify programs running on these devices. We present a roadmap from today's continuously powered IoT devices to tomorrow's battery-free IoT devices that highlights software engineering challenges for intermittent programs running on battery-free devices.**

THE INTERNET OF THINGS (IOT) forms a network of physical devices that can sense the environment via their sensors, perform computation and communicate wirelessly to interact with each other and exchange information. IoT applications (e.g., smart homes and cities, autonomous vehicles, wearables) support various tasks in our daily lives *intelligently* to increase our comfort and efficiency. On the other hand, the global *energy consumption* of IoT edge devices (e.g., sensors, actuators, and gateways) is already gigantic, i.e., equal to Portugal's annual electricity consumption in 2015 [1]. The consumption will increase considerably since the number of IoT edge devices will exceed 75 billion soon. Moreover, future *intelligent* IoT applications will employ modern Artificial Intelligence (AI) techniques that demand more computing capabilities and, in turn, more energy. For instance, Deep Neural Networks (DNNs) require thousands of mathematical operations to enable inference applications such as computer vision. Millions of IoT edge devices executing these operations consume a total power on the order of gigawatts, which is equivalent to millions of tons of CO2 per year [2].

The majority of IoT edge devices are powered using *batteries* which can store only a finite amount of energy. These energy-constrained devices (e.g., sensor nodes, implants, wearables) sense raw data, process it, and communicate wirelessly to push the pre-processed field information to more powerful nodes in the hierarchy. It is not feasible for them to offload computationally intensive tasks (e.g., inference tasks) to the cloud by sending *large amounts* of raw sensor data

Published by the IEEE Computer Society

This is the author accepted version of an article published in
IEEE Software, volume 39, issue 5, Sept.-Oct. 2022
https://doi.org/10.1109/MS.2022.3184519

and waiting for the results since communication is energy-intensive and can drain batteries frequently. Contemporary IoT applications tend to execute computationally intense AI tasks on edge devices and use the energy stored in batteries more conservatively. However, as the energy requirements of these tasks increase, it is inevitable for edge devices to drain their batteries more and more often. Unfortunately, replacing millions of batteries (optimistically every year) introduces a significant maintenance cost, and recycling batteries pose a severe threat to our environment.

Researchers are continuously proposing several hardware (e.g., power-efficient DNN accelerators) and software solutions (e.g., approximate computing) to decrease the energy requirements of IoT applications and extend the battery life of edge devices. However, batteries are still the most significant obstacle against long-lived, stand-alone, and environmentally-friendly IoT. Fortunately, the progress in energy harvesting circuits and the decrease in power requirements of processing, sensing, and communication hardware promised the potential of freeing IoT devices from their batteries. On the other hand, operating, without batteries, by relying only on ambient energy changes the way we develop software significantly.

Ambient energy is unpredictable and subject to environmental conditions. Therefore, removing batteries and relying only on ambient energy introduce frequent *power failures* that interleave the software execution with intervals during which the batteryless device is off and harvest ambient energy into its *tiny energy reservoir* (e.g., a capacitor) to operate again. This phenomenon led to the emergence of a new computing paradigm, the so-called *intermittent computing*. A minuscule amount of energy is spent to perform a burst of tasks and save the computational state (e.g., the global variables, program stack, general-purpose registers, program counter) in *nonvolatile memory* to recover upon a power failure. Upon recovery, the computation progresses forward from the latest successfully saved computational state.

Intermittent computing requires custom recovery solutions and programming models (e.g., checkpoints [3] and task-based models [4]) to tolerate power failures that might keep programs in an *inconsistent* computational state. A program in an inconsistent state might never progress correctly, output meaningful results, and terminate. For instance, a batteryless IoT edge executing audio event detection (using an acoustic sensor and DNN-based features for event classification) might never infer the audio event (e.g., human detection) due to the power failures that hinder execution progress. Naturally, most research in the past decade focused on the design and development of new programming models [5], language constructs [3], and runtimes [4] to ensure the consistency of the computational state and forward progress during intermittent execution.

Despite the recent efforts on intermittent computing, several challenges and research opportunities are waiting for the attention of researchers and practitioners. As a practical example, intermittent programs may fail at any time, between any two lines of code, during unpredictable lengths of time spent to charge the capacitor using sporadic ambient energy. They might be functionally correct but not be beneficial since they might not satisfy their non-functional requirements (i.e., timing constraints) on the target deployment environment. It is hard to predict the execution time of an intermittent program and check how likely it satisfies its timing constraints on a given deployment area. Today's popular IoT application development techniques [6], including modeling approaches, verification methods, and test environments, overlook these challenges since they only target continuously-powered systems.

Here, we highlight the challenges of programming the batteryless edge that deserve more profound study and understanding beyond those topics focusing on developing continuously powered IoT solutions. We emphasize that we need new software engineering techniques and tools (e.g., for the verification of intermittent programs, testing) to enable beneficial and reliable intermittent IoT applications. With the rise of mobile devices, there was an emerging need for a new generation of software verification and validation techniques and tools solely addressing mobile applications, such as mobile device security and testing of mobile applications. With the emergence of intermittent computing, we expect a similar need to arise in software engineering practice targeting batteryless IoT devices.

This paper is structured as follows. We

first give the description of intermittent computing on the batteryless IoT edge. Then, we highlight the differences between programming continuously-powered and intermittently-powered devices. Unique challenges for programming the batteryless edge are introduced in the final part.

Intermittent Computing on the Batteryless IoT Edge

Batteryless edge devices harvest energy from ambient (e.g., via solar panels) or dedicated wireless energy sources (e.g., radiofrequency transmitters such as WiFi routers). As depicted in Figure 1, the main components of a batteryless edge are (a) an *energy harvester* converting incoming ambient energy into electric current, (b) an *energy buffer* (typically a capacitor) storing the harvested energy to power electronics, (c) an *ultra-low-power microcontroller* that orchestrates sensing, computation, and communication, and (d) nonvolatile memory that is used to capture the volatile program state. Typical examples of batteryless edge devices are Flicker [7] (which can be powered using several harvesters from solar to piezoelectric) and Camaroptera [8] (which contains an ultra-low-power camera sensor and a long-range wireless transmitter). The ultra-low-power micro-controllers in intermittent computing platforms (e.g., MSP430FR5969 from Texas Instruments) comprise a combination of volatile and nonvolatile memory.

Forward Progress and Memory Consistency

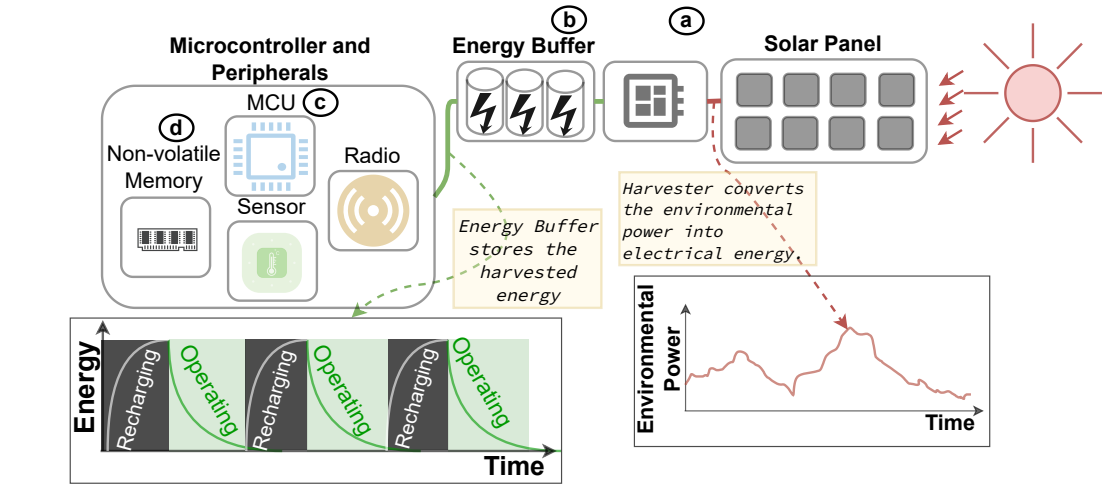
The frequent loss of the computation state is an inevitable phenomenon for the batteryless edge that operates using only harvested energy. Upon a power failure, the contents of the CPU registers and the volatile memory (i.e., the volatile computational state) are lost. Therefore, power failures hinder the *forward progress* of the computation: the computation starts from the beginning, and the intermediate, volatile results are lost at each reboot. Restarting a computation block after a power interrupt might also lead to catastrophic side-effects on *memory consistency*. If the program modifies the nonvolatile memory, *Write-After-Read (WAR)* dependencies on persistent variables (i.e., variables in nonvolatile memory) might keep these variables inconsistent [5] since repeated computation may produce different results (the violation of *idempotency*).

Figure 1 presents a code snippet (part of a sensing application that runs on a batteryless IoT edge) and an example intermittent execution scenario that demonstrates how WAR dependencies might lead to memory inconsistencies. `x` and `vector[]` are persistent variables maintained in nonvolatile memory. After executing `x++` (which sets `x=1`) in the scenario, a power failure occurs and leaves `x` modified. Upon recovery from the power failure, the device executes {`x++; vector[x]=0;`}, which increments `x` again and sets **vector[2]=0**. In a continuously powered execution (without power failures), the device would execute {`x++; vector[x]=0;`} only once, and we would observe **vector[1]=0**. Due to the power failure, `x++` is executed twice, and a different output is obtained.

If the program's control flow depends on external inputs such as sensor readings (e.g., checking persistent variables whose values are updated during I/O operations), power failures might lead to inconsistent program behavior [9]. In the scenario (adapted from [9]) shown in Figure 1, the temperature value (using `readTemp()`) is read, and the persistent variable `alarm` is set to true (the sensed temperature is more than a predefined limit) just before a power failure occurs. After recovery, the device re-executes the previously executed code lines and re-reads the temperature value. This time the temperature value is smaller than the predefined limit, and hence the persistent variable `tempOK` is set to true. At this point, both `alarm` and `tempOK` are *true*, which is logically incorrect.

What Makes Programming the Batteryless Edge Different?

Operating without batteries requires dealing with the forward progress of computation and memory consistency. These issues change significantly the way we develop software. Placing checkpoints or employing the task-based model are the two major approaches that ensure the forward progress of the computation and keep the nonvolatile memory consistent during intermittent execution. Figure 2 presents the checkpointed and task-based versions of a C program (i.e., 1-D convolution code for DNN inference) developed for continuously powered systems.



*A batteryless edge device operates **intermittently** by performing bursts of computation interleaved with time intervals during which the device is off and harvesting energy to fill its energy buffer.*

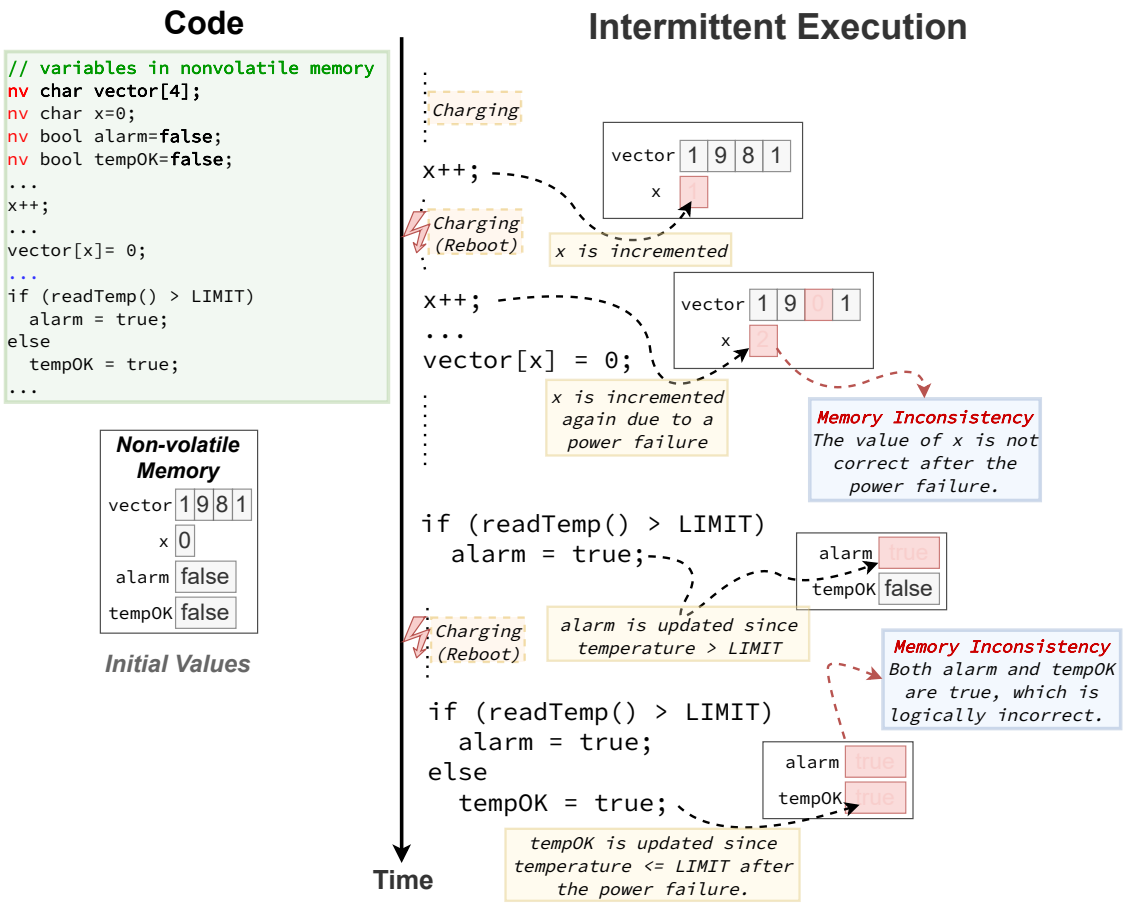


Figure 1. The main components of a batteryless IoT edge device are (a) an *energy harvester*, (b) an *energy buffer*, (c) an *ultra-low-power microcontroller*, and (d) *nonvolatile memory*. Intermittent execution due to the frequent loss of the computation state is an inevitable phenomenon for batteryless edge devices. If the program code modifies the nonvolatile memory, power failures might keep persistent variables in an inconsistent state.

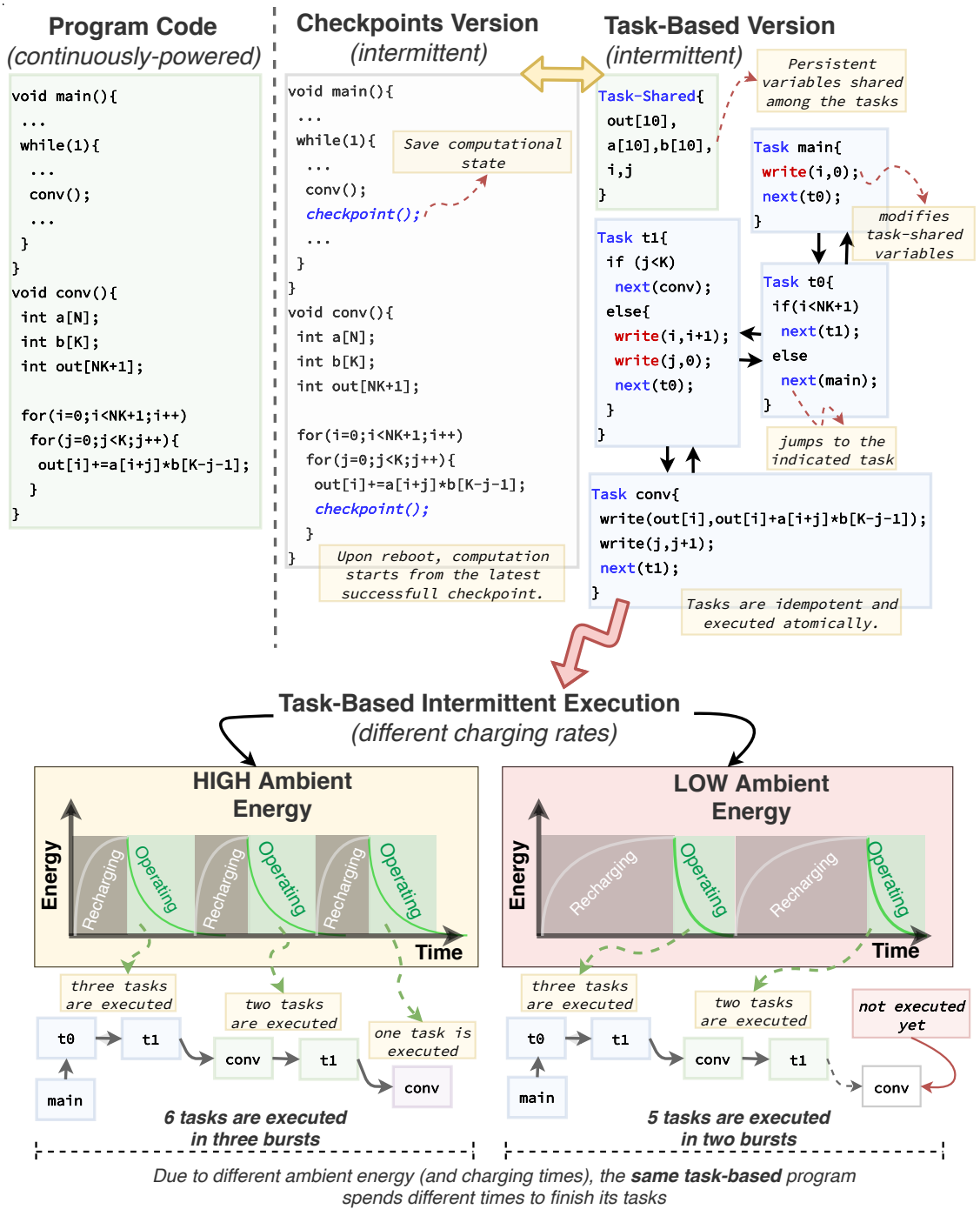


Figure 2. The top part of the figure presents the checkpointed and task-based versions of a 1-D convolution code for DNN inference developed for continuously powered systems. The batteryless device executes more tasks in a high-energy ambient environment during a fixed time interval since the capacitor is charged faster.

Checkpoints. In checkpointing, either a programmer or a compiler instruments the program to save the program state in nonvolatile memory [3]. In Figure 2, the `checkpoint()` interface (provided by the underlying runtime) inserts checkpoints to a C program. The runtime stores the checkpoint information by protecting it via *double buffering*. Thus, the new program state does not supersede the prior one immediately if the checkpoint data is partially updated due to a power failure. Moreover, the persistent variables modified by the program should also be *versioned* to keep nonvolatile memory consistent across reboots [10]. Compiler analysis is required to determine which persistent variables to be modified between two checkpoints. On crossing each checkpoint, the runtime saves the program state and versions the necessary persistent variables. After a reboot, the program state and versions are restored using the checkpoint data. The restore operation makes the code between two checkpoints naturally idempotent.

Task-based Model. This model requires the programmer to divide the computation into a set of tasks [5], as depicted in Figure 2. The task re-execution always produces the same results since the inputs are never modified (and the WAR dependencies are eliminated). Thus, tasks are idempotent (re-executable). Moreover, they are atomic in the sense that they have all-or-nothing semantics and cannot interrupt each other. The task-based model creates the local copies of the persistent variables shared among the tasks (the variables shown within the `Task-Shared` block in Figure 2). Each task manipulates its local copies (e.g., using `write` interface) and atomically commits them to original locations upon completion (e.g., using `next` interface). This operation prevents memory inconsistencies due to power failures.

Factors Affecting Software Development and Intermittent Program Behavior

Employing checkpoints or task-based programming models only ensures the forward progress of intermittent execution and memory consistency. These programming models introduce an unavoidable energy overhead. However, the energy overhead is reasonable, as confirmed by the state-of-the-art [10], [5], [3], [4]. In particular, the task-based programming model is pretty efficient since it logs the minimum amount of the data in non-volatile memory [5], [4]. Besides, other factors affect the intermittent execution of programs.

Energy Harvesting Environment. The energy availability of the deployment environment is stochastic and a significant factor for the rate of power failures, the execution time of intermittent programs, and in turn, program throughput. The execution time of programs running on continuously powered devices is more predictable since these programs are not affected by the stochastic nature of ambient energy. The harvested ambient energy depends on several factors, such as the energy source type (e.g., solar or radiofrequency), the distance to the energy source, and the efficiency of the energy harvesting circuit. As depicted in Figure 2, when incoming power is strong enough, the capacitor is charged rapidly, and the device becomes available quickly after a power failure. At low input power, the charging is slower and takes more time. Since the computing progresses slowly due to longer charging periods in a low-energy environment, the program might miss its deadlines, and the program throughput may not meet expectations (e.g., a batteryless long-range remote visual sensing system should take a picture every 5 minutes and transmit the relevant ones every 20 minutes).

Hardware Configuration. The energy consumption attribute of the target hardware leads to quantitative differences in program behavior (e.g., in program throughput and execution time). The power requirements of the target platform affect the end-to-end delay of program execution. The intermittent program might take a long time to finish on hardware platforms having high power requirements since the capacitor discharges faster. Hence, the program might drain the capacitor more frequently (since some instructions consume more energy in a shorter time). Therefore, the device is interrupted by frequent power failures, and it is unavailable and charging its capacitor for long periods. For instance, special instructions such as nonvolatile memory access instructions consume more energy than others, drain energy more frequently, and increase program execution time.

On the other hand, other hardware attributes

such as capacitor size and voltage threshold settings also affect the intermittent program behavior. The capacitor size determines the maximum length of program execution without a power failure. If the capacitor size is large, the device has more energy to spend until the power failure, but charging the capacitor takes more time. Another attribute is the size of the volatile program state (i.e., the volatile memory size and the number of registers on the target platform) that affects the checkpointing overhead, i.e., the execution time and energy consumption of the checkpoint operation that saves the program state in nonvolatile memory. The checkpointing overhead is architecture-dependent since the number of registers and the volatile memory size change from target to target.

Runtime Characteristics. Checkpoint-based recovery and task-based models are supported by runtime environments (e.g., [3], [4]). These runtimes provide programmers interfaces to develop intermittent programs and perform the necessary recovery/logging operations. Thus, today's intermittent programs are coupled tightly to the underlying runtime environments. Even the programs using the same programming model (e.g., task-based) are not portable across platforms and not compatible with other runtimes. Moreover, each runtime introduces different processing delays and energy overheads during intermittent execution and, in turn, changes the program behavior significantly.

Program Structure. Intermittent programming models require that source code be decomposed into code blocks (atomic tasks or code blocks divided by checkpoint instructions). These blocks need to be efficient and terminating. Their termination is guaranteed if they consume less energy than the capacity of the energy storage buffer. Therefore, while decomposing code into blocks, programmers consider only stored energy and not additional energy harvestable during execution. The termination of code blocks short enough is ensured, but having more code blocks than necessary may waste energy and impose an execution-time overhead, e.g., due to saving the program state in nonvolatile memory for each code block. For instance, checkpoint placement is crucial for programmers to ensure the desired timing behavior of their intermittent programs. The more frequent the checkpoints are, the more energy consumed, but less computation is lost upon a power failure. Code decomposition based on energy storage size, energy efficiency, and the forward progress of computation represents a new software design aspect unique to intermittent programs.

Unique Challenges for Programming the Batteryless Edge

Due to the differences and factors we presented, when implementing intermittent programs, programmers must consider several new challenges that are unfamiliar to most of the application developers that target continuously powered IoT systems.

Energy-aware Timing Analysis. Considerable research on intermittent computing has been devoted to compile-time analysis to find bugs and anomalies of intermittent programs [9] and structure them (via effective task splitting and checkpoint placement) based on worst-case energy consumption analysis [11], [12]. Despite these efforts, no attention has been paid to analyzing the timing behavior of intermittent programs. Without such an analysis, programmers will never know *at compile-time* if their intermittent programs execute as they intend to do in a real-world deployment (e.g., meeting throughput requirements). Worse still, it is extremely costly and time-consuming to analyze the timing behavior of intermittent programs on real deployments because programmers need to run the programs multiple times on the target hardware.

Design Space Exploration. The execution time and throughput of intermittent programs depend on multiple hardware and software design factors such as the capacitor size, the energy consumption of the target hardware, the efficiency of the energy harvester unit, and the program structure (e.g., checkpoint placement and the size and number of tasks in task-based models). As an example, consider a deployment environment with low ambient energy and frequent power failures. If the program execution time and throughput do not meet requirements in the low-energy environment, programmers might increase the capacitor size, change the target hardware, or remove some checkpoints. These changes may not always lead to what is intended (e.g., the bigger the capacitor

size is, the longer the charging takes). Therefore, programmers might have to do a what-if analysis by deploying several program versions into various hardware configurations, i.e., reconfiguring the hardware, restructuring the program, and checking if the restructured program has the desired execution time and throughput on the reconfigured hardware. This what-if analysis is currently manual and not guided. It can quickly become infeasible on target hardware deployments due to the size of design space, i.e., the number of possible hardware configurations and program versions.

Energy-aware Testing. The impact of harvesting ambient energy on the behavior of intermittent programs complicates their testing. Programmers need to test their programs with power failures under various energy conditions. They can expose some bugs only under distinct power failure timings or test cases across energy conditions. They also need to test energy-related program properties such as forward progress. The tools and techniques to test intermittent programs are mostly the same tools and techniques designed for testing programs running on continuously powered systems. They do not inherently support mimicking power failures and ambient energy conditions during intermittent program testing. Programmers need new testing tools with simulator support that can accurately emulate real-world energy harvesting conditions.

Runtime Independent Programming. Each intermittent runtime supports different language constructs and abstractions (e.g., Alpaca [13] supporting the privatization of data shared between tasks, and InK [4] enabling reacting to changes in available energy and variations in sensing data). When writing programs, programmers use these runtime-specific language constructs to support memory correctness, timely execution, etc. Intermittent programming is a fast-growing area, and thus, intermittent runtimes constantly evolve together with the language abstractions they support. New runtimes come with new constructs, or updates on the constructs are introduced for the existing runtimes. Programmers modify the program for the new/updated constructs. Or, they port it from an old runtime to a new one, which may require fundamental changes, e.g., new task structures replacing checkpoint instructions. It is a manual, time-consuming, and error-prone task. Therefore, programmers need techniques supporting runtime independent program models transformed (semi-) automatically into runtime dependent intermittent execution models.

Software Adaptation. Energy-aware adaptation of program execution (e.g., reducing sensor sampling rates or degrading computation) is a promising way to avert power failures, meet timing deadlines, and increase program throughput. There are different adaptation strategies that all depend on the characteristics of intermittent applications. For some applications, decreasing the number of sensor readings might be a better solution when the ambient energy is low. Some other applications might need to keep the sensing rate constant but can degrade the computation by skipping some computationally heavy code blocks. Due to constrained device capabilities and limited energy information, it is challenging to decide the best time and strategy for adapting execution. Estimating the available energy in an environment during runtime is hard. And, small changes in the ambient energy might have a significant impact on program execution. Therefore, we need flexible and configurable runtime adaptation frameworks [14] that provide automatic responses to changes in energy based on the adaptation heuristics programmers specify concerning environmental and physical phenomena (e.g., when off-time increases, degrade program execution to maintain throughput since the environment is experiencing energy scarcity).

Reusability of Libraries. The libraries implemented for continuously powered systems are not reusable for intermittent systems. Due to the rigid checkpoint and task-based programming models, programmers need to reimplement the new versions of open-source libraries, and programs are prevented from using closed-source libraries. As of now, intermittent programs can use closed-source libraries by employing checkpoints. Checkpointing the internal state of these libraries (e.g, when and what to checkpoint), and power failure recovery might be different for each library. Providing a generic solution for closed-source library management remains an open question for researchers and practitioners.

Secure Intermittent Execution. A significant software challenge to the widespread use of bat-

teryless devices is the secure execution of intermittent programs. Although cryptographic keys and algorithms, security certificates, protocols, and other security mechanisms used for continuously powered IoT devices still play a critical role in intermittent computing, several technical challenges remain. Ensuring secure intermittent computing is difficult due to the limited capabilities and energy budgets of batteryless devices. Intermittent execution models and runtimes do not provide inherent security support, but program recovery with charge-discharge cycles can pose high-security risks. For instance, by altering the checkpoint image, attackers can manipulate the state of the intermittent program and prevent the device from functioning correctly. A power failure might leave a cryptographic operation uncompleted and private data in an insecure state. Attackers can gain physical access to the device and obtain private data in the device's memory. Programmers should pay extra attention when implementing security functions in intermittent programs. Some security functions (e.g., stateful signature generation functions) may need to be executed without a power interruption.

Conclusion

Sustainable software is necessary for many reasons [15]: economic reasons, environmental reasons, and because society has sustainability awareness that has increased dramatically over the past decade. Intermittent computing paves the way for sustainable software in the batteryless edge. Due to the differences and factors we presented, programmers implementing software running on intermittent devices must consider various challenges unfamiliar to most programmers developing continuously powered systems. These challenges require new software engineering tools and techniques for the development and testing of intermittent programs.

Acknowledgments

This work was supported by the SINTEF projects G-IoT (Green Internet of Things) and Sustainable IoT (funded by the Research Council of Norway) and the Italian Ministry for University and Research (MUR) under the program Dipartimenti di Eccellenza (2018-2022).

REFERENCES

1. X. Liu and N. Ansari, "Toward green IoT: Energy solutions and key challenges," *IEEE Communications Magazine*, vol. 57, no. 3, pp. 104–110, 2019.
2. K. Hao, "Training a single ai model can emit as much carbon as five cars in their lifetimes," *MIT technology Review*, 2019.
3. V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawełczak, "Time-sensitive intermittent computing meets legacy software," in *ASPLOS'20*, 2020, pp. 85–99.
4. K. S. Yıldırım, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester, "InK: Reactive kernel for tiny batteryless sensors," in *SenSys'18*, 2018, pp. 41–53.
5. A. Colin and B. Lucia, "Chain: tasks and channels for reliable intermittent programs," in *OOPSLA'16*, 2016, pp. 514–530.
6. A. Taivalsaari and T. Mikkonen, "A roadmap to the programmable world: software challenges in the iot era," *IEEE Software*, vol. 34, no. 1, pp. 72–80, 2017.
7. J. Hester and J. Sorber, "Flicker: Rapid prototyping for the batteryless internet-of-things," in *SenSys'17*, 2017, pp. 19:1–19:13.
8. H. Desai, M. Nardello, D. Brunelli, and B. Lucia, "Camaroptera: A long-range image sensor with local inference for remote sensing applications," *ACM Transactions on Embedded Computing Systems (TECS)*, 2022.
9. M. Surbatovich, L. Jia, and B. Lucia, "I/o dependent idempotence bugs in intermittent systems," in *OOPSLA'19*, 2019, pp. 1–31.
10. B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," in *PLDI'15*, 2015, pp. 575–585.
11. A. Colin and B. Lucia, "Termination checking and task decomposition for task-based intermittent programs," in *CC'18*, 2018, pp. 116–127.
12. S. Ahmed, M. Nawaz, A. Bakar, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola, "Demystifying energy consumption dynamics in transiently powered computers," *ACM Transactions on Embedded Computing Systems*, vol. 19, no. 6, pp. 1–25, 2020.
13. K. Maeng, A. Colin, and B. Lucia, "Alpaca: intermittent execution without checkpoints," in *OOPSLA'17*, 2017, pp. 1–30.
14. 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, 2021.

15. A. Fonseca, R. Kazman, and P. Lago, “A manifesto for energy-aware software,” *IEEE Software*, vol. 36, no. 6, pp. 79–82, 2019.

Arda Goknil is a senior research scientist at SINTEF Digital in Oslo, Norway. His research interests include model-driven software engineering, software testing, requirements engineering and intermittent computing. Dr. Goknil received his doctorate in software engineering from University of Twente in the Netherlands. Contact him at arda.goknil@sintef.no.

Kasım Sinan Yıldırım is an assistant professor at the Department of Information Engineering and Computer Science, University of Trento, Italy. His research interests include embedded systems and IoT, wireless communication and protocols, self-organizing sensor networks and distributed algorithms, operating systems/run-times and architectural support for tiny embedded devices. Dr. Yildirim received his doctorate in computer science from Ege University in Turkey. Contact him at kasimsinan.yildirim@unitn.it.