Report

Software-based, Intelligent Energy Optimization Methods for Green IoT

Author(s):

Arda Göknil, Hui Song, Rustem Dautov, K. Sinan Yildirim

Report No:

2021:01423 - Unrestricted

Client(s) (pos partner):

Internal

SINTEF Digital
Postal address:
Postboks 124 Blindern
0314 Oslo, Norway
Switchboard: +47 40005100
info@sintef.no

Enterprise /VAT No:
NO 919 303 808 MVA

Report

Software-based, Intelligent Energy Optimization Methods for Green IoT

KEYWORDS
Intermittent Computing, Energy Harvesting, Software-defined IoT Systems, Green IoT

VERSION
1.0

DATE
2021-12-16

AUTHOR(S)
Arda Göknil, Hui Song, Rustem Dautov, K. Sinan Yildirim

CLIENT(S)
Internal

CLIENT'S REFERENCE

PROJECT NO.
102024558-1

NO. OF PAGES/APPENDICES
17+Appendices

SUMMARY
Internet of Things (IoT) has the potential to support a more sustainable world by digitalizing various daily-life tasks and industrial processes. On the other hand, the global energy consumption of IoT devices is already gigantic. The energy consumption and the e-waste of IoT devices will increase considerably since the number of devices will exceed 75 billion soon. Over the years, Green IoT (G-IoT) has emerged as a research area to reduce the energy consumption of IoT devices and increase the lifespan of these devices. Among many, we consider intermittent computing and software-defined IoT systems two important research directions within the context of G-IoT. In this paper, we investigate the intermittent computing techniques, categorize them, and present the open challenges and research opportunities that can assist the research community. We provide an overview of software-defined IoT systems, discuss their sustainability objectives, and present techniques to achieve these objectives. As a position paper, we focus on the opportunities and challenges rather than the results and findings.

PREPARED BY
Arda Göknil

SIGNATURE
Arda Goknil
Arda Goknil (Jan 10, 2022 14:10 GMT+1)

CHECKED BY
Ketil Stølen

SIGNATURE
Ketil Stølen
Ketil Stølen (Jan 10, 2022 21:10 GMT+1)

APPROVED BY
Odd Are Svensen

SIGNATURE
Odd Are Svensen
Odd Are Svensen (Jan 13, 2022 14:02 GMT+1)

REPORT NO.
2021:01423

ISBN
978-82-14-07702-5

CLASSIFICATION
Unrestricted

CLASSIFICATION THIS PAGE
Internal

Document history

VERSION	DATE	VERSION DESCRIPTION
1.0	2021-12-16	Final version

Table of contents

1 Introduction

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. Thus, there is a need for research to develop IoT systems that achieves lower energy consumption, optimized energy footprints, and longer lifespan of IoT devices.

Green IoT (G-IoT) has emerged as a research area to reduce the energy consumption of IoT devices and increase the lifespan of these devices. It is defined as "energy-efficient procedures (hardware or software) adopted by IoT technologies either to facilitate the reduction in the greenhouse effect of existing applications and services or to reduce the impact of the greenhouse effect of the IoT ecosystem itself" [18]. Among many, we identify two important research directions within the context of G-IoT:

Intermittent computing: The advancements in energy-harvesting circuits and ultra-low-power microelectronics enable battery-free sensing devices that operate by using harvested energy only. Intermittent computing refers to programming models where periods of program execution are separated by reboots. Intermittent systems are generally powered by energy-harvesting devices: they start executing a program when the accumulated energy reaches a threshold and stop when the energy buffer is exhausted.

Software-defined IoT systems: The term denotes systems where even the lower-end sensors are programmable and can be remotely updated/reflushed/redeployed with new software that will define how the devices work. This means longer lifespan for devices, and even repurposing – i.e., reusability in different scenarios. With the advances in wireless connectivity, software (re-)deployment can be performed in a remote, scalable and completely automated manner, typically through an IoT cloud platform.

This paper presents the overview of two research dimensions, i.e., intermittent computing and software-defined IoT systems, as part of software-based, intelligent energy optimization methods for Green IoT. We first investigate the intermittent computing techniques in the literature, categorize them, and present the open challenges and research opportunities that can assist the research community. We later provide an overview of software-defined IoT systems, discuss the potential sustainability objectives of these systems, and present techniques to achieve these objectives. We focus on the opportunities and challenges rather than the final results and findings.

The paper is structured as follows. Section 2 presents 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. In Section 3, we discuss software-defined IoT systems that adapt themselves based on changing conditions to achieve sustainability objectives. Section 4 concludes the paper.

2 Intermittent Computing and Green IoT

The majority of IoT edge devices are powered by 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 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 computational 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 non-volatile 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 [2] and task-based models [3]) 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 [4], language constructs [2], and runtimes [3] to ensure the consistency of the computational state and forward progress during intermittent execution.

Despite the recent efforts on intermittent computing, several unique 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 the 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 [5], including modeling approaches, languages, verification methods, test environments, and tools, overlook these challenges since they only target continuously powered systems. For instance, the state-of-the-art does not model intermittent computing and execution and does not consider the energy availability of the environment that leads to dynamic charging/discharging times during

energy harvesting. Therefore, existing software development techniques are poorly suited to the new generation of fully sustainable IoT applications running on batteryless edge devices.

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.

2.1 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 (i) an energy harvester converting incoming ambient energy into electric current, (ii) an energy buffer (typically a capacitor) storing the harvested energy to power electronics, (iii) nonvolatile memory that is used to capture the volatile program state, and (iv) an ultra-lowpower microcontroller that orchestrates sensing, computation, and communication. Typical examples of batteryless edge devices are Flicker [6] (which can be powered using several harvesters from solar to piezoelectric) and Camaroptera [7] (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.

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 [4] since repeated computation may produce different results (the violation of idempotency).

Figure 1 presents a code snippet and an example intermittent execution scenario that demonstrate 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 [8]. In the scenario 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.

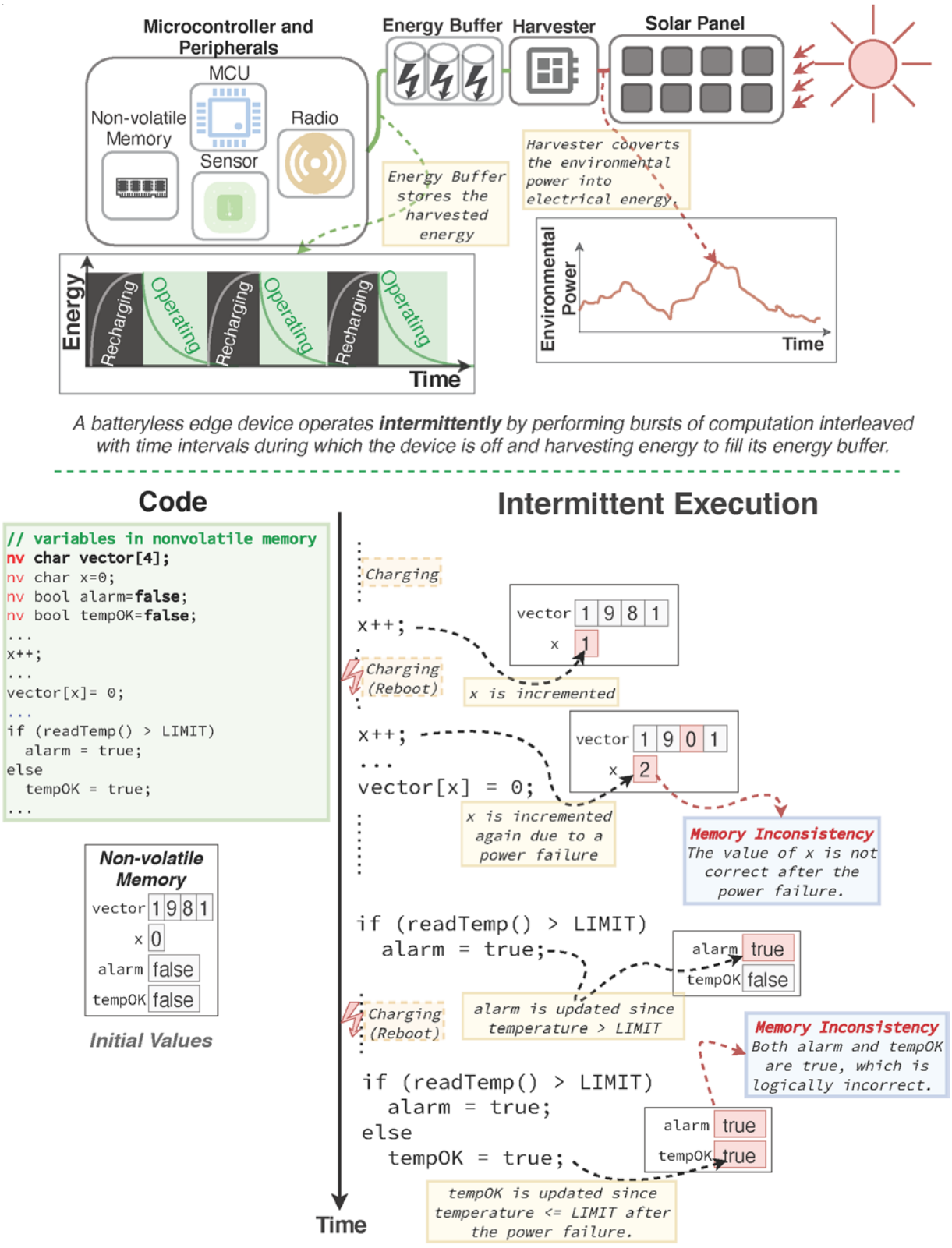

Figure 1. The main components of a batteryless IoT edge device are (i) an energy harvester, (ii) an energy buffer, (iii) an ultra-low-power microcontroller, and (iv) nonvolatile memory. Intermittent execution due to the frequent loss of the computation state is an inevitable phenomenon for batteryless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

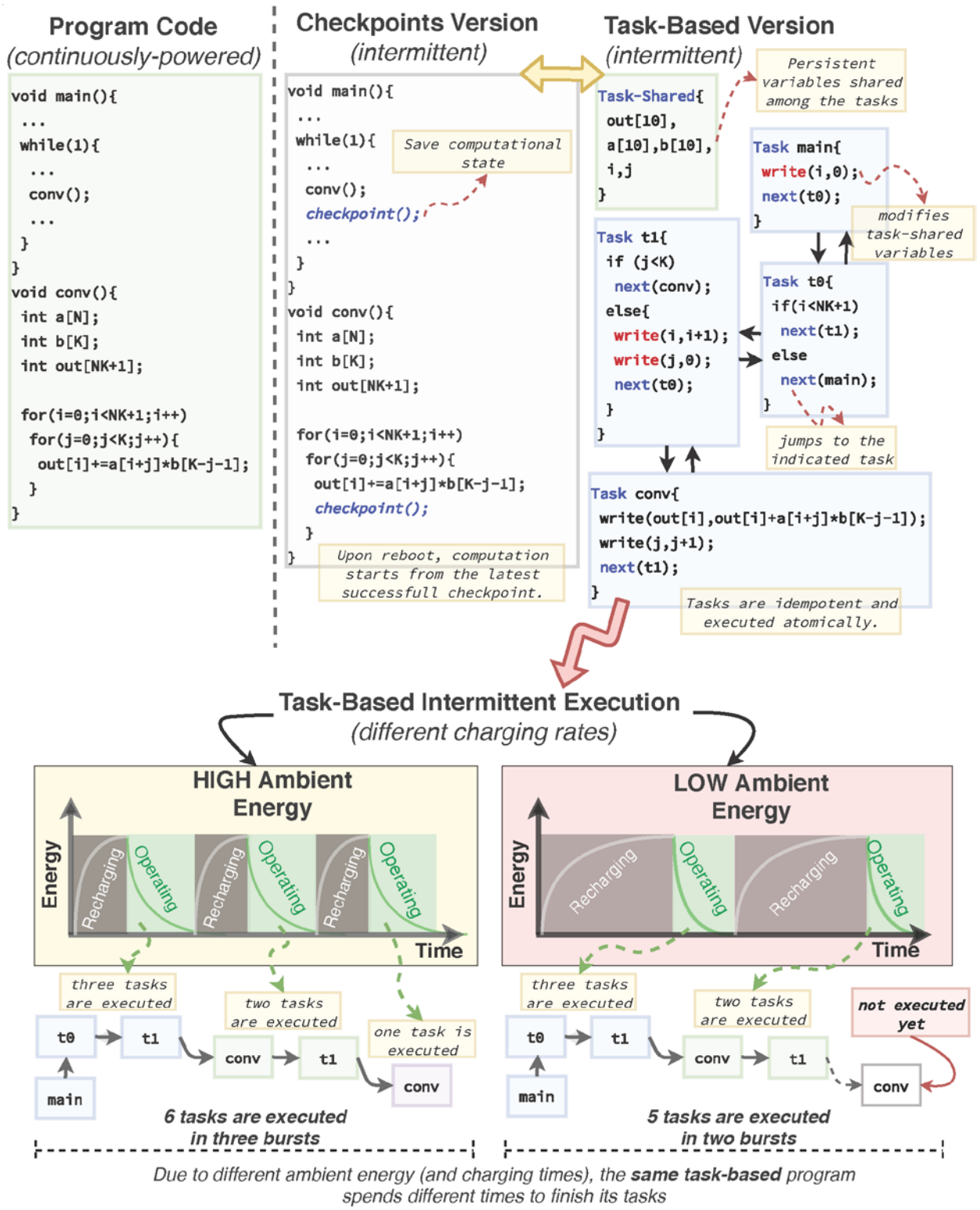

Figure 2. The top part of the figure presents the checkpointed and task-based versions of a 1-D convolution code for DNN inference developed for continuously powered systems. The batteryless device executes more tasks in a high-energy ambient environment during a fixed time interval since the capacitor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

2.2 What Makes Programming the Batteryless Edge Different?

Operating without batteries requires dealing with the forward progress of computation and memory consistency. These issues significantly change 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.

Checkpoints. In checkpointing, either a programmer or a compiler instruments the program to save the program state in nonvolatile memory [2]. 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 [9]. 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 [4], 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.

2.2.1 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. 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., [2], [3]). 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., taskbased) 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.

2.3 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 [8] and structure them (via effective task splitting and checkpoint placement) based on worst-case energy consumption analysis [10], [11]. 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 [12] supporting the privatization of data shared between tasks, and InK [3] 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 [13] 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 batteryless 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.

2.4 Summary and Conclusion for Intermittent Computing

Sustainable software is necessary for many reasons [14]: 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.

3 Software-Defined IoT Systems

The IoT technology started from development and applying many types of specialized devices for sensing and actuation, to connect the physical things into the internet. For the sake of cost, energy consumption, and physical maintenance, these devices are normally designed for the task, or the things that they are attached to. They undertake the minimal tasks, normally just sensing in a pre-defined interval or simple actuation actions. The business logics such as data aggregation, analytics, decision making, etc., are hosted in the cloud.

The recent years have seen an overturn of this trend. As semiconductor technology improves, the unit cost for the computation power keeps decreasing, and the more and more logics are pushed from the cloud down to the devices, leading to the emerging of edge computing. The logics are running as software hosted by the different devices, and concentrating on edge devices, such as gateways, microcontrollers, etc. Such a trend has the following major impact to the green aspect of IoT systems:

- The flexibility of software defined IoT logics makes it possible for dynamic adaptation to optimize the resource and energy usage.
- The capability of redeploying software for new tasks and usages allows the reuse of devices for different purposes, reducing the potential electronic waste.

3.1 Dynamic adaptive IoT software for reducing energy consumption

Self-adaptation is a widely expected feature for both hardware and software systems. The idea is that the systems are able to automatically adapt their status and behaviours when their context has changed, to achieve better performance, tolerate failures and reduce the maintenance cost. Over decades of research and innovation in this direction, there are still huge potential that is worth exploring and exploiting for self-

adaptive systems, and to have green and sustainable aspects into the consideration is an important direction for the future work on self-adaptive systems.

We briefly discuss the potential sustainability aspects that can be considered for self-adaptive systems, and the possible adaptation points to achieve these aspects. The report will be focused on the opportunities and challenges, rather than initial results and findings.

3.1.1 Green objectives for self-adaptive systems

We foresee the following aspects, or objectives, for self-adaptive IoT systems, which may have an impact on the sustainability of the target systems.

- **Lower energy consumption of the whole system**. IoT systems are energy consuming. Sensors consume electricity to stay alive and to collect data. After that, the whole data processing pipeline from sensors, edge devices, local servers, to the central cloud, requires electricity from different powers sources. At the same time, data communication is also energy consuming. Along the pipeline, different devices and communication channels have different efficiency on energy consumption, and an optimized pipeline may significantly reduce the total energy consumption of the whole system.
- **Optimized energy footprints**. In an IoT system, different devices and resources may use different power sources, and therefore the same unit of energy consumption may have different environment footprint. For example, an edge device on batteries may have a higher energy footprint than a server in the datacentre powered by renewable electricity. Even for the same datacentre, there is possibility to switch between different power sources. If possible, optimization of the total footprint may be a more relevant may be a more relevant objective for green self-adaptation.
- **Longer lifespan of IoT devices**. The production of IoT and edge devices also have significant environment footprints, both on the manufacturing side and on the recycling side. Increasing the lifespans of these devices will contribute to the reduction of such footprints. In the next subsection, we will talk about re-deployment of software on IoT devices to reuse the same device for different purposes, as a way to extend the lifespan. However, more generally, for devices that are used for the same scenario, self-adaptation for an optimized usage and better maintenance activities will also increase the lifespan.
- **Reduced maintenance cost**. Self-healing and self-recovery without human interaction by system operators is an important direction of self-adaptive systems. This will reduce the cost of human maintenance, which also have environmental impacts, such as the logistic of system operators or the transportation of the devices.

3.1.2 Potential adaptation points

These green objectives can be achieved through different types of system adaptations, also known as the management capabilities of the IoT systems.

- **Elastic resource allocation**: A system can dynamically adjust the computation, storage or network resources to some of its services according to the computation load, in order to reduce the energy consumption without harming the performance
- **Dynamic service placement**: In a Cloud-edge-IoT continuum, different parts have different energy efficiency. Placing services on the right parts (e.g., cloud vs. edge) will optimize the total energy consumption of the whole system and help to increase the lifespan of the devices or their batteries. Moving services closer or father from the data sources can also balance the energy consumption on data processing and communication.
- **Switching between alternative service implementations**: The same service can be implemented in different technologies. Depending on the contexts, some alternatives may have better energy efficiency than others. Switching the alternatives based on the contexts is also a way to optimize the total energy consumption.

- **Dynamic management of device lifecycle**. Edge devices are normally designed to tolerant frequent switching on and off. On the application side, the use of more stateless microservices also allows migrating services from inactive devices. Exploiting these capabilities can lead to an active control of devices lifecycles to reduce unnecessary energy consumption, and also extend device lifespan.
- **Self-healing and self-recovery**. Modern devices and application services have the capability to tolerate unexcepted situations and can recover from failures. Self-adaptation based on these capabilities in a global perspective will also increase the system performance and significantly reduce the need for manual maintenance.

The adaptation points listed above are not meant to be an exhaustive list, but rather some examples of possible mechanisms that a future sus self-adaptive system can utilize.

In summary, the sustainability concern is emerging as an import aspect for designing and implementing self-adaptative systems, and on the other hand, self-adaptive systems have many potential mechanisms to manipulate system to achieve these sustainability-oriented aspects.

3.2 Software redeployment on IoT devices for reducing e-waste

3.2.1 Motivation

Billions of sensors are already embedded in a vast array of networked physical objects, and soon it will be difficult to buy a new product without connected smart features. Controlling an appliance from a mobile phone is already a standard feature and ubiquitous sensors and actuators will transform industrial automation, buildings, and our interaction with the surrounding world.

This global digital transformation is driven by a new class of microcontrollers (MCUs), i.e. stand-alone miniature computing chips equipped with flash memory for code storage, and a small amount of RAM in which to execute the code. They are also able to communicate with hardware peripherals via General Purpose Input/Output (GPIO). This includes both digital I/O and associated protocols such as I2C, Serial, SPI, and CAN, as well as analog I/O for reading data from environmental sensors and other analog sources. The power efficiency means that MCUs, even with sensors, can be placed virtually anywhere and run for years on a small battery, or indefinitely by adding an energy harvesting unit such as a solar cell. By using a fraction of the power that single-board computers do, their total cost of ownership is significantly lower.

With the rapid development of the Internet of Things (IoT), MCU-enabled tiny sensor devices have penetrated almost every aspect of people's everyday life, enabling smart and interactive environments, such as homes, buildings and even whole cities. This rising sensor-based economy, however, produces myriads of devices with short lifespans and offers no way of properly disposing them once they become obsolete [15]. The United Nations estimated 74.7 million tonnes of e-waste to be generated annually by 2030, thus almost doubling in 16 years, with less than a third of it being recycled [16].

While MCUs have been embedded in everything from toys to cars for several decades now, it is only been in the last few years that they have become truly useful for the IoT. They are not only sufficiently powerful to run complex applications, but many have also received built-in support for gateway connectivity such as Ethernet, WiFi, Bluetooth, and others. Many of them have also opened up their memory buses to be extended with external flash and RAM. As we further argue in this report, these increased connectivity capabilities can and should be exploited in order to enable hardware re-use and re-purposing, and thus to extend the lifespan of IoT devices and to slow down the e-waste generation rate. To enable this, we present in this report a scalable hierarchical agent-based approach to deploying code to terminal IoT devices via edge gateways. The proposed architecture enables re-programming and re-purposing of devices, not immediately connected to the Internet, thus facilitating hardware re-use and re-purposing, reducing the device obsolescence, and, eventually, contributing to the creation of a more sustainable IoT.

3.2.2 The proposed architecture

In a simplified IoT hierarchy, we distinguish between three main elements, i.e., a centralised cloud platform, Internet-connected edge gateways, and downstream IoT devices not connected to the Internet, but only to a local edge gateway. Accordingly, data communication takes place via two links between (i) the cloud platform and edge gateways and (ii) edge gateways and IoT devices. While both types of network interaction are a common state of practice, it is still a challenge to propagate data (e.g., firmware updates) from the cloud down to terminal IoT devices via edge gateways in a generic, scalable, and automated manner.

The cloud counterpart is usually able to directly communicate with edge gateways, but not with the billions of terminal sensor devices placed in the field. In these circumstances, it is typically not feasible to target a firmware update to a specific single IoT device (using, for example, its ID). Instead, it is required to annotate firmware updates with target conditions and push them down to the edge layer, which, in its turn, will evaluate the conditions and route the updates to matching IoT devices in its subnet. Thus, edge gateways become the key element in this software management hierarchy. Accordingly, we tackle the challenge of last-mile software deployment by introducing a context-aware deployment agent to be installed on edge gateways in order to bridge the communication gap between the cloud and IoT devices by 'routing' software updates to target IoT devices taking into account the current cyber-physical context. The agent is expected to be operating in a daemon-like manner, implementing the following four-fold functionality, as depicted in Figure 3.

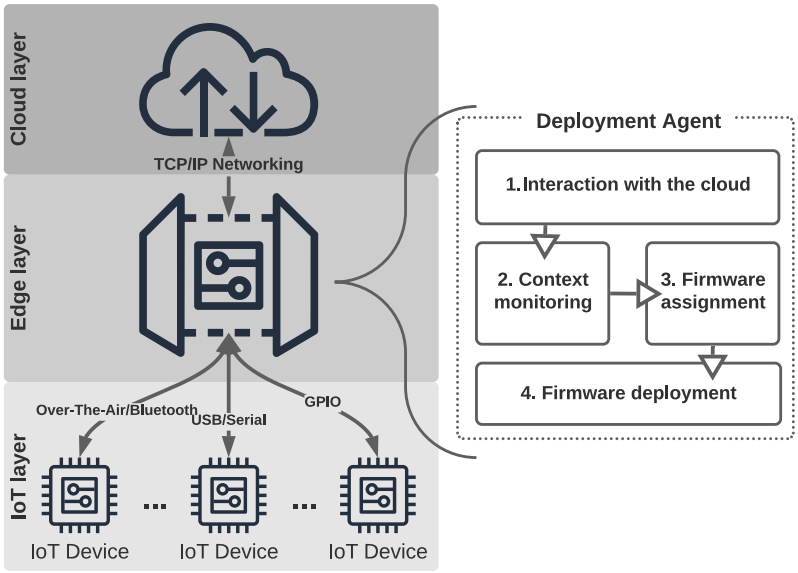

Figure 3. Architecture for last-mile deployment

- **Interaction with the cloud**: edge gateways sit in the middle of the IoT hierarchy and are able to receive updates from the cloud. It is expected that the cloud platform maintains a list of registered edge gateways and communicates with them asynchronously using standard pub-sub mechanisms or -- less usual -- synchronously using direct method invocations. In both cases, edge gateways should be able to receive update commands (annotated with target conditions) along with the actual code to be installed on IoT devices.
- **Context monitoring**: edge gateways are expected to perform continuous context monitoring by keeping track of associated downstream devices. These are closely located IoT devices using the gateway for pushing collected sensor data via one of the available communication interfaces, such as Bluetooth, ZigBee, serial USB connection, or even direct physical wiring via GPIO pins. The collected context information about downstream IoT devices, among other things, may include the current firmware version, manufacturer brand and model, MCU architecture, available physical

interfaces, installed hardware sensors/actuators, etc. It is important to collect the relevant context information, since it will be taken into account at the next step to resolve to which devices firmware updates should be applied.

- **Context-aware firmware assignment**: at this step, edge gateways will map target conditions of received firmware updates with the collected device context information, thus assigning new firmware to matching devices. In its simplest form, this can be implemented as a collection of AND and OR logical operators, where each IoT device is evaluated independently from the rest. More sophisticated scenarios might take into account the whole fleet of IoT devices, so that updates are assigned following some global policies. For example, it might be required to evenly distribute several firmware variants among the devices, or follow an A/B testing strategy, where some new code is tried only on a limited subset of devices. This step should also consider situations when an IoT device satisfies conditions of multiple updates, as well as when none suitable devices are found. As an outcome of this step, the deployment agent generates a map of firmware updates and matching IoT devices (assuming that a device can accept at most one update at time).
- **Interaction with downstream IoT devices**: the firmware update pipeline concludes with the actual flashing of new code onto matching IoT devices. Once again, the collected context information about IoT devices will help the deployment agent to determine which underlying physical interface should be invoked in each individual case.

3.2.3 Summary

The proposed last-mile deployment agent, provisioned on edge gateways via the centralised cloud platform in the form of containerised micro-services, is able to receive firmware updates from the cloud and install them on connected IoT devices. The major benefit of this solution is that IoT software engineers are not required to know about target IoT devices beforehand when releasing code updates (which would be simply infeasible given the growing numbers). Instead, the task of assignment of firmware updates is shifted to the edge layer and takes place on gateways, which are able to evaluate target conditions against the collected cyber-physical context in order to selectively install new code to terminal IoT devices. By offloading the software assignment task to the edge layer, the proposed solution implements a hierarchical approach to software deployment in the IoT-Edge-Cloud continuum, and also contributes to the increased scalability by parallelising the software assignment task across multiple edge gateways, rather than centralising it on the cloud. By integrating with the already available cloud-based container management at the edge, it is possible to build an end-to-end software management pipeline, so that even terminal IoT devices without Internet connectivity can be re-flashed in an automated scalable manner. This is a step towards a sustainable IoT where all the elements, even dormant and outdated, are not wasted, but are rather re-programmed and re-used.

A potentially promising direction for future work is to implement dynamic discovery of IoT devices in a local network of an edge gateway. The currently implemented prototype assumes that IoT devices are connected to the host gateway via a USB port; in this case, implementing a polling mechanism for continuously updating the list of connected devices and their cyber-physical contexts is relatively straight-forward using the standard Linux tools. A more challenging task would be to implement similar dynamic discovery in a local TCP/IP network or even via a wireless interface by detecting idle devices within the range of the edge gateway. Some initial attempts in this direction are reported in [17].

Another direction yet to be explored is the continuous self-adaptive behaviour of the system, as well as the intelligent resolution of infinite loops and potential conflicts between the changed context (caused by the applied firmware updates) and the target conditions. While the current proof of concept is implemented as a finite four-step pipeline triggered by the initial update command received from the cloud, some advanced scenarios would require a continuous operation where firmware assignment and installation would be triggered by the changing cyber-physical context of IoT devices.

In this report we focused on demonstrating only the technical feasibility of designing and implementing sustainable IoT systems for smart environments. There are, however, many other aspects to be considered and addressed in this respect. We intentionally omitted the discussion on the device ownership, access management, and security issues, which are major hindering factors. The proposed solution (and any other similar approaches relying on re-programming and re-purposing existing IoT infrastructures) is not yet ready to be immediately put into practice and adopted by the society. As any other `green' initiative, it would require involving not just ICT researchers and IoT companies, but rather multiple stakeholders and policy makers from several adjacent domains, including governmental, environmental, and legal organisations, who should join their efforts to define common strategies for building sustainable IoT systems.

4 Conclusion

We presented the overview of intermittent computing and software-defined IoT systems as part of software-based, intelligent energy optimization methods for Green IoT. We first introduced the open challenges and research opportunities for intermittent computing. We then discussed the potential sustainability objectives of software-defined IoT systems and the techniques to achieve these objectives.

Program execution without batteries requires techniques ensuring the forward progress of computation and memory consistency. Placing checkpoints or employing the task-based model are the two major approaches that guarantee the forward progress of the computation and keep the non-volatile memory consistent during intermittent execution. Besides, some other factors affect the intermittent program execution. One of the significant factors is the energy availability of the deployment environment. It is stochastic and affects the rate of power failures, the execution time of intermittent programs, and, in turn, program throughput. Therefore, 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.

The differences and factors we presented for intermittent programming force programmers to consider various challenges unfamiliar to most programmers developing continuously powered systems. Existing IoT application development techniques, including modeling approaches, languages, verification methods, test environments, and tools, overlook these challenges since they only target continuously powered systems. These challenges require new software engineering tools and techniques to develop and test intermittent programs.

Software-defined IoT systems enable dynamic adaptation to optimize resource and energy usage. They have the capability of redeploying software for new tasks and usages, which allows the reuse of IoT devices for different purposes and reduces the potential electronic waste. We identified various research directions for software-defined IoT systems, e.g., the dynamic discovery of IoT devices in a local network of an edge gateway, the continuous self-adaptive behavior, and the intelligent resolution of infinite loops and conflicts between the changing context and the target conditions.

5 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] 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.

[3] 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.

[4] A. Colin and B. Lucia, “Chain: tasks and channels for reliable intermittent programs,” in OOPSLA’16, 2016, pp. 514–530.

[5] 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.

[6] J. Hester and J. Sorber, “Flicker: Rapid prototyping for the batteryless internet-of-things,” in SenSys’17, 2017, pp. 19:1–19:13.

[7] M. Nardello, H. Desai, D. Brunelli, and B. Lucia, “Camaroptera: A batteryless long-range remote visual sensing system,” in ENSsys’19, 2019, pp. 8–14.

[8] M. Surbatovich, L. Jia, and B. Lucia, “I/o dependent idempotence bugs in intermittent systems,” in OOPSLA’19, 2019, pp. 1–31.

[9] B. Lucia and B. Ransford, “A simpler, safer programming and execution model for intermittent systems,” in PLDI’15, 2015, pp. 575–585.

[10] A. Colin and B. Lucia, “Termination checking and task decomposition for task-based intermittent programs,” in CC’18, 2018, pp. 116–127

[11] 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.

[12] K. Maeng, A. Colin, and B. Lucia, “Alpaca: intermittent execution without checkpoints,” in OOPSLA’17, 2017, pp. 1–30.

[13] 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.

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

[15] S. Higginbotham, “The internet of trash: IoT has a looming e-waste problem,” IEEE Spectrum: Technology, Engineering, and Science News, vol. 17, 2018.

[16] V. Forti, C. P. Balde, R. Kuehr, and G. Bel, “The Global E-waste Monitor2020: Quantities, flows and the circular economy potential,” tech. rep., United Nations University/United Nations Institute for Training and Research, International Telecommunication Union, and International Solid Waste Association, 2020

[17] R. Dautov and S. Distefano, “Targeted content delivery to IoT devices using Bloom filters,” International Conference on Ad-Hoc Networks and Wireless, pp. 39–52, Springer, 2017

[18] C. Zhu, V. CM Leung, L. Shu, and E. C-H. Ngai, "Green internet of things for smart world," IEEE access, vol. 3, pp. 2151-2162, 2015.