electronics

MDPI

Article

ICEr: An Intermittent Computing Environment Based on a Run-Time Module for Energy-Harvesting IoT Devices with NVRAM

Junho Kwak, Hyeongrae Kim and Jeonghun Cho *

School of Electronic and Electrical Engineering, Kyungpook National University, Daegu 41566, Korea; junho7513@gmail.com (J.K.); hn02301@gmail.com (H.K.)
* Correspondence: jcho@knu.ac.kr; Tel.: +82-53-950-5516

Abstract: With the development of energy-harvesting technology, various applications have been developed that can be operated only with harvested energy, thereby making energy-harvesting technology suitable for edge devices in poor environments where battery replacement is difficult. However, devices with energy-harvesting technology have limitations: an application can operate intermittently in an energy-harvesting device, and the device's energy is greatly affected by the environment and the state of the device. Intermittent computing causes abnormal progress or affords incorrect results. The factors affecting the energy of the device can change the operation of the device. To solve these problems, we propose the "Intermittent Computing Environment based on a run-time module" (ICEr), which dynamically controls and manages an application for normal operation in intermittent computing. ICEr comprises an energy checker and a controller. The energy checker measures the energy state of a device at run-time, and the controller controls and manages an application through Backup, Restore, Sleep, and Wakeup. The controller optimizes those operations by considering the energy state and memory state together to minimize time and energy overhead. In this study, two kinds of experiments were conducted. In the first experiment, Embench was selected as the target application to validate ICEr and measure its performance. This experiment validated that ICEr behaves dynamically in various environments. Moreover, it showed a reduction in relative execution time overhead of up to 50% and a reduction in energy overhead of up to 49.5% against Hibernus, depending on the environment. In the second experiment, ICEr was applied to the *Temperature Measurement Application*, and the improvement of the energy efficiency for the real Internet-of-Things (IoT) application was confirmed.

Keywords: energy harvesting; intermittent computing; run-time; low-power; Internet-of-Things (IoT); non-volatile random access memory (NVRAM)

Citation: Kwak, J.; Kim, H.; Cho, J. ICEr: An Intermittent Computing Environment Based on a Run-Time Module for Energy-Harvesting IoT Devices with NVRAM. *Electronics* **2021**, *10*, 879. https://doi.org/10.3390/electronics10080879

Academic Editor: Kris Campbell

Received: 16 March 2021
Accepted: 03 April 2021
Published: 7 April 2021

1. Introduction

Energy-harvesting technology is advancing [1,2]. These advances enable the development of various applications that operate only with harvested energy [3–5]; thus, energy-harvesting technology is suitable for devices in environments where battery replacement is difficult. Consequently, energy-harvesting technology has been widely applied to edge devices operating in such environments [6,7].

However, these devices have limitations. The first limitation is that an application operates intermittently when the energy stored in the device is insufficient. In intermittent computing where an application operates intermittently, an application cannot proceed normally if forward progress is not guaranteed. Additionally, an application can lead to incorrect results if data consistency is not guaranteed. These problems cause the application to behave abnormally, preventing the device from performing its role. Therefore, it is important to ensure forward progress and data consistency in intermittent computing [8]. The second limitation is that the energy stored in the device is greatly affected by the

environment and the state of the device. The amounts of energy harvested and consumed depend on the environment and the state of the device, which in turn affect the energy stored in the device. Since the operation of the device is affected by the energy stored in the device, these factors can change the operation of the device. Therefore, it is also important to consider these factors when controlling the operation of the device.

Previous studies have investigated the problems that occur in intermittent computing using the programming model [9,10], the compiler-based model [11–13], and the just-in-time (JIT) model [14–16]. The programming model affords programming tools that can design an application operating in intermittent computing. While designing an application using the programming tools, developers consider the operational environment of the energy-harvesting device. The compiler-based model automatically inserts checkpoints into an application at compile time. Then, the checkpoint stores the application's operating state in the non-volatile memory to preserve the state even during power failure. The JIT model measures the energy state of an energy-harvesting device and executes an application based on the energy state. When the energy state is insufficient, the application's operating state is stored in the non-volatile memory. When the energy state is sufficient, the application's operating state is restored and the application is executed again.

Previous studies ensured forward progress and data consistency. However, they all have limitations. The programming model requires additional effort from developers and has low portability. To develop an application by using the programming model, developers must know how to use the programming tools and predict the application's energy consumption. This is a difficult and complicated process. Furthermore, the application needs to be redesigned based on the environment. The compiler-based model has large time and memory overheads. Numerous checkpoints need to be inserted in it to ensure forward progress because the model does not know the environments in which an application will run. The insertion of numerous checkpoints increases the necessary amount of code and the overheads related to checkpoints. The JIT model determines the behavior only by the energy state. This model does not consider the environment and the state of the device except for the energy state. It is the same for other models.

In this paper, the "Intermittent Computing Environment based on a run-time module" (ICEr) is proposed. Since ICEr is based on the JIT model, it dynamically controls and manages an application in intermittent computing. ICEr comprises an energy checker that measures the energy state and a controller that controls and manages an application. When ICEr is operational, the controller dynamically controls and manages an application by considering not only the energy state, but also the memory state, which is the most important state of the device. ICEr's dynamic operation minimizes unnecessary operations and optimizes time and energy overheads. To validate ICEr, an intermittent computing environment was built with the MSP-EXP430FR5594 LaunchPad Development Kit [17] and function generator. The benchmarks of Embench [18] were selected as target applications, and ICEr ran the target applications in the built environment. In addition, a wireless network environment was built with the XBee module to measure the ICEr's performance in a real Internet-of-Things (IoT) environment. In this environment, the *Temperature Measurement Application* was selected as the target application. A series of experiments showed that ICEr executes the application normally in intermittent computing and that it can increase energy efficiency in a real IoT environment.

Section 2 provides background on intermittent computing and related studies. Section 3 describes the proposed ICEr, and Section 4 describes experiments to validate the ICEr's operation and to measure its performance in the IoT application. Finally, Section 5 concludes the paper.

2. Background

2.1. Energy Harvesting and Intermittent Computing

Energy-harvesting technology harvests energy from the environment and uses it as computing power [19]. Energy-harvesting technology is advancing with the advances

in device technology [1,2], enabling various applications (e.g., IoT devices; wearable, implantable, and ingestible medical sensors; infrastructure monitors; and small satellites) to operate with harvested energy alone [3–5]. Therefore, energy-harvesting technology is suitable for devices in poor environments where battery replacement is difficult. It allows devices in hard-to-reach locations to operate without battery replacement. It also allows wireless sensor networks, for which it is expensive to change the batteries due to the number of devices, to reduce battery replacement costs. Therefore, energy-harvesting technology is widely applied to edge devices operating in poor environments [6,7].

An application is intermittently executed when sufficient energy is not harvested and the energy stored in the device is insufficient to operate the application. In this intermittent computing, energy charging and discharging are repeated. Figure 1 depicts an energy-harvesting device's behavior based on its energy state. The energy state is determined by the harvested and consumed energy. When the device is not operational, it consumes little energy. If the consumed energy is significantly less than the harvested energy, the energy of the device is charged. When the energy reaches a sufficient energy level, the system is turned on and the device is operational. Otherwise, the device would consume a lot of energy if it stayed continuously operational. If the consumed energy is greater than the harvested energy, the energy of the device is discharged. If the energy falls to an insufficient energy level, the device becomes non-operational. Then, the system is turned off and the application does not run. Through this process, the energy-harvesting device repeats the charging and discharging process, and the application repeats the running and stopping—intermittent computing.

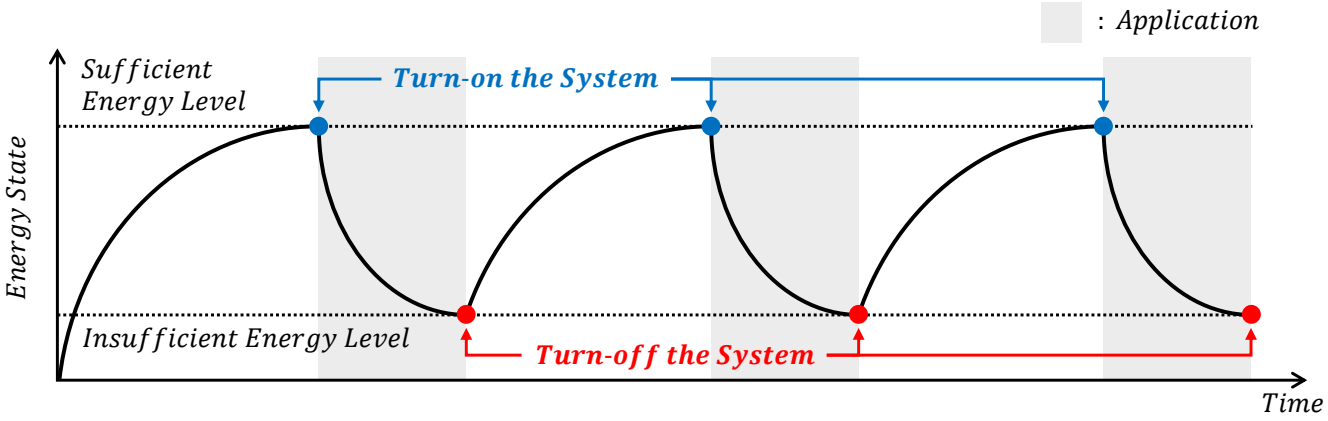

Figure 1. Intermittent computing by charging and discharging energy.

When an application runs intermittently, forward progress and data consistency must be ensured for the normal operation of the application [8]. If they are not ensured, the application can behave abnormally and cause some problems, as shown in Figure 2. In Figure 2a, power failure occurs after the execution of the first backup function that backs up the application's operating state. If the energy reaches the sufficient energy level, the system is turned on and the application is executed from the first backup function. However, power failure occurs again before the second backup function is executed because the sensing function and processing function consume a lot of energy. Consequently, whenever the system is turned on, only the same part of code is repeatedly executed and the application does not proceed. This is a problem that occurs when forward progress is not ensured. In Figure 2b, the "result" variable is allocated to the non-volatile memory to preserve its value even in case of power failure. The `for` loop accumulates the values of the `a` array in the "result" variable. After the `for` loop, power failure occurs and the second backup function is not executed. When the system is turned on again, the application is executed from the first backup function; thus, the `for` loop is executed again. Since the "result"

variable maintains its previous value, the wrong value is stored in the "result" variable. This problem occurs when data consistency is not ensured.

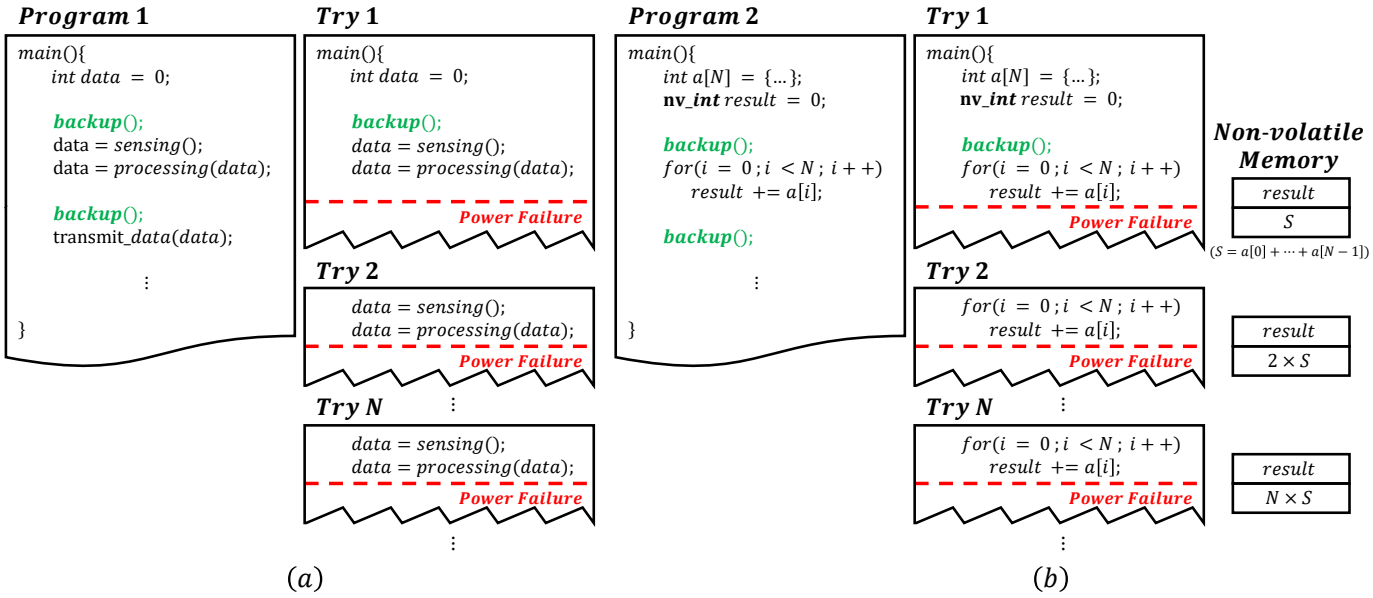

Figure 2. Problems that can occur in intermittent computing: (**a**) A problem that occurs when forward progress is not ensured. (**b**) A problem that occurs when data consistency is not ensured.

In addition, energy efficiency is also an important issue in intermittent computing. Energy-harvesting devices are mainly deployed in poor environments, and they use only harvested energy as the computing power. Thus, energy efficiency needs to be increased. Furthermore, wireless communication is usually essential in energy-harvesting devices that are used as edge devices [20,21]. However, wireless communication consumes a lot of energy, especially when connecting to networks. Therefore, in order to increase energy efficiency, energy consumed in wireless communication needs to be reduced [22].

2.2. Non-Volatile Random Access Memory

In intermittent computing, data in volatile memory should be stored in non-volatile memory to be retained when the energy-harvesting device is turned off [23]. The non-volatile memory includes non-volatile random access memory (NVRAM) and the commonly used electrically erasable programmable read-only memory (EEPROM) and flash memory. NVRAM is a random access memory (RAM) that maintains stored information even when power is not supplied, and it includes magnetoresistive random access memory (MRAM), ferroelectric random access memory (FRAM), etc. Among them, FRAM, a representative NVRAM, has a limit of a low degree of integration compared to EEPROM and flash memory, so it has a large area per unit capacity and a high cost per bit. However, FRAM has lower data read/write time, lower power consumption, and higher durability. Additionally, it can access data in 1-byte units and overwrite data without erasing [24,25]. These features of FRAM are advantageous for intermittent computing where data backup is frequently performed.

2.3. Related Works

2.3.1. Programming Model

The programming models support the design of an application that operates in intermittent computing [9,10]. Developers can design the entire operation by using the programming models by considering the environments where the application will operate. Representative programming models are Chain [9] and Alpaca [10].

Both Chain and Alpaca afford a programming method based on a task that is an execution unit of an application. The entire operation is divided into small operations, and each small operation is designed as a task. Then, the programming model builds an application by connecting tasks. If power failure occurs while the application is running, the task that is being executed stops. When the energy is sufficiently charged and the system is turned on again, the execution of application begins from the last task it was running before stopping. Chain and Alpaca also provide non-volatile memory access in their own way to maintain data values. When developers set the data to be stored in the non-volatile memory, the data are updated in non-volatile memory whenever the task is terminated.

The programming models can optimize an application in intermittent computing. If developers design a task to consume the appropriate energy for the target energy-harvesting device and environments, the re-execution of the task can be reduced. Moreover, only necessary data can be set to be stored in the non-volatile memory. However, since predicting the energy consumption of tasks is difficult, developers need to expend significant amounts of effort and development time on the programming models. In addition, if the energy-harvesting device or environment changes, the developers must redesign the application to reflect it. Therefore, the programming models have low portability.

2.3.2. Compiler-Based Model

The compiler-based models add checkpoints that store the application's operating state in the non-volatile memory [11–13]. Checkpoints are automatically inserted in the appropriate position of the application by the compiler at compile-time. Ratchet [11] and Chinchilla [12] are representative compiler-based models.

Ratchet inserts checkpoints between idempotent sections. Idempotent sections are separated by instructions that perform Write-After-Read (WAR) on the same memory address. Therefore, Ratchet analyzes WAR dependencies at compile-time and divides an application into idempotent sections. Meanwhile, Chinchilla inserts checkpoints between the basic blocks of an application. Since the operation of a basic block is generally simple and consumes little energy, inserting checkpoints between basic blocks reliably ensures forward progress. However, Chinchilla inserts too many checkpoints. Therefore, the checkpoints are selectively activated using a timer. Both Ratchet and Chinchilla store the application's operating state in the non-volatile memory at checkpoints. When the system is turned on again after power failure, the application's operating state stored by the last checkpoint is restored and the application is executed after the last checkpoint.

The compiler-based models of the related studies do not require developers to expend additional effort for intermittent computing, as the compiler automatically inserts checkpoints. Developers only need to consider the behavior of an application while designing it; hence, the development time is reduced. However, these models do not know information about the application, hardware platform, and environments where the application will be executed. Therefore, many checkpoints are inserted for safe operation. This increases not only the code size, but also the time and energy overhead due to checkpoint execution. If additional information is provided to the compiler-based models, they can be extended to insert optimal checkpoints. However, this is limited because the compiler cannot reflect the dynamic characteristics of execution.

2.3.3. JIT Model

The JIT models measure the energy state of an energy-harvesting device and execute an application based on the energy state [14–16]. These models determine whether an application is executable through the energy state and perform appropriate operations to control the application. Representative JIT models are Mementos [14] and Hibernus [15].

Mementos has characteristics of both the compiler-based model and the JIT model. Mementos inserts trigger points, which are similar to checkpoints, into an application at compile-time. Then, it measures the energy state at run-time and selectively activates the

trigger point based on the energy state. If enough energy is present, the trigger point is not activated. Hibernus controls an application by using a hardware interruption. It generates an interruption signal when the measured energy reaches the defined energy levels (e.g., sufficient and insufficient energy levels). If an interruption signal occurs, the interrupt handler controls the application according to the signal. Mementos and Hibernus store the application's operating state in the non-volatile memory in their own way when the energy becomes insufficient. Thereafter, when sufficient energy is present, the application's operating state is restored and the application is executed.

Unnecessary data backup and restore behaviors are reduced in JIT models, as these models operate according to the energy state. In addition, similarly to the compiler-based models, in these models, developers do not need to expend additional effort related to intermittent computing. However, JIT models require additional hardware for measuring the energy state, consequently consuming additional energy to run the hardware. Moreover, these models do not consider states of an energy-harvesting device other than the energy state (e.g., the memory state).

3. Our Proposed Approach: ICEr

ICEr is a JIT-based model that supports the normal operation of an application with low-power via intermittent computing. In an energy-harvesting device, ICEr measures the energy state, and then it controls and manages an application based on the measured energy state and the memory state. The following subsections describe ICEr in detail. Section 3.1 provides ICEr's system structure and ICEr's components, and Section 3.2 provides ICEr's operational flow. Section 3.3 describes how ICEr ensures forward progress and data consistency. Finally, Section 3.4 describes ICEr's energy efficiency, and Section 3.5 describes ICEr's programmability and portability.

3.1. System Structure

Figure 3 depicts the system structure of an energy-harvesting device to which the proposed ICEr is applied. The system structure comprises hardware, firmware, ICEr, and application layers. Since the system is an energy-harvesting device, the hardware layer contains the energy buffer and energy harvester. The energy buffer is a central energy buffer that stores the harvested energy and supplies the stored energy to the device. A supercapacitor is used as the energy buffer because a supercapacitor has a fast charge/discharge time. Additionally, since the voltage of a supercapacitor varies according to the amount of energy stored, the energy state can be easily measured through the voltage. The energy harvester can be a general energy harvester (Solar, RF radio, etc.) that can harvest energy depending on the environment. Unlike a general embedded system structure, an ICEr layer exists below an application layer for controlling and managing an application.

ICEr comprises an energy checker and a controller. The energy checker is a hardware unit that measures the energy state of the device. For this, the energy checker directly accesses the energy buffer. After measuring the energy state, the energy checker transmits a signal that indicates the energy state to the controller. Then, the application is controlled and managed by the controller that is a software unit. The controller first recognizes the energy state through the signal transmitted from the energy checker. Then, the controller properly performs Backup, Restore, Sleep, and Wakeup according to the energy state and memory state. The energy checker and controller operate independently, thereby allowing ICEr to control and manage the application in run-time.

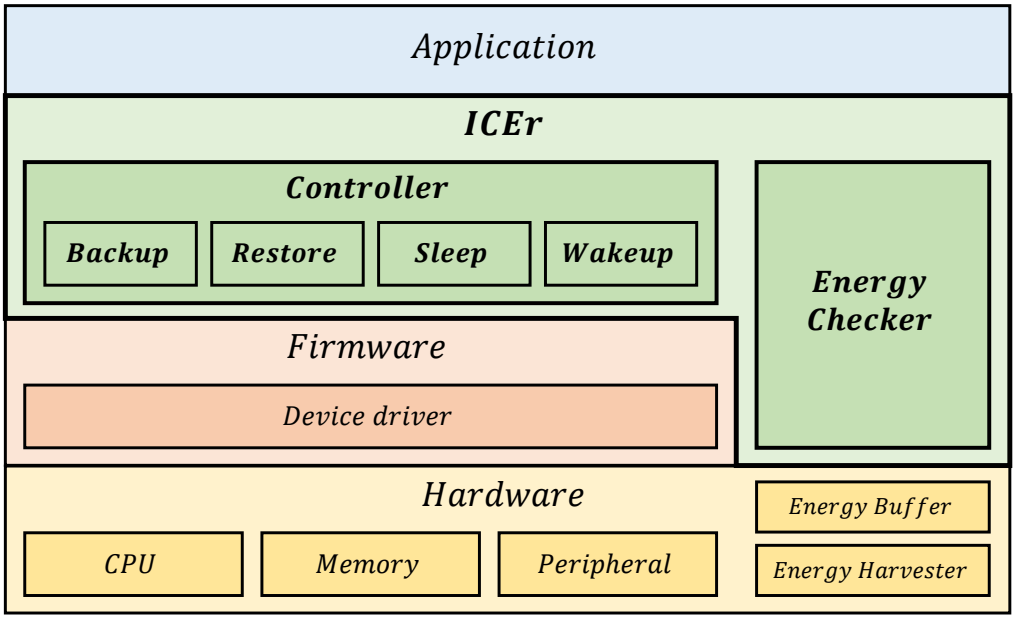

Figure 3. Structure of the system with the "Intermittent Computing Environment based on a run-time module" (ICEr).

3.1.1. Energy Checker

The energy state of a device is measured by the energy checker by directly accessing the energy buffer that stores the energy harvested from the energy harvester. Then, the energy checker generates a signal indicating the energy state and transmits it to the controller. The signal can either be a sufficient energy interrupt signal or an insufficient energy interrupt signal.

The energy checker compares the energy state with predefined energy levels to generate a signal. The predefined energy levels include the sufficient energy level and the insufficient energy level. The sufficient energy level indicates enough energy for the device to execute the application. It must be set to an appropriate level within the operating voltage range of the device. If the sufficient energy level is set too high, it takes a long time to charge, and if it is set too low, there is not enough energy available to execute the application. Since it is difficult to accurately calculate the amount of energy to operate the application normally, an experimental method of running the application in advance is used to find the appropriate energy level that satisfies the requirements (e.g., normal operation of the application; charging time). The insufficient energy level indicates the energy level at which the application cannot be executed. If the energy state becomes insufficient to execute the application, the application's operating state must be stored in non-volatile memory through data backup. Therefore, the insufficient energy level must be sufficiently higher than the minimum operating voltage to safely perform the data backup before the device is turned off. An experimental method is also used to find the insufficient energy level because the energy required for data backup is difficult to calculate.

On the one hand, a sufficient energy interrupt signal is generated when the energy state is greater than the sufficient energy level. It is a signal informing the system that enough energy is present to operate an application. On the other hand, an insufficient energy interrupt signal is generated when the energy state is less than the insufficient energy level. It is a signal informing the system that the energy is not enough to operate an application. These signals should be generated while the system is turned on, as the controller uses these signals to recognize the energy state. Therefore, the energy checker continues to run while the system is turned on.

The energy checker consists of a comparator and an interrupt. It additionally includes a regulator to generate a constant voltage representing the sufficient and insufficient energy levels. The comparator continuously compares the voltage of the supercapacitor, which is

the energy buffer, and the sufficient and insufficient energy levels to measure the energy state. To compare the two energy levels, the comparator uses the function to automatically switch the input energy levels as shown in Figure 4. This function automatically switches the input to another energy level when the energy state reaches the currently set input energy level. When the application is stopped because there is insufficient energy to run the application, the comparator sets a sufficient energy level as the input. Since the application is stopped, the energy state increases and the comparator compares it to the sufficient energy level. If the energy state increases and reaches the sufficient energy level, the sufficient energy interrupt signal is generated and the input of the comparator is switched from the sufficient energy level to the insufficient energy level. When the application runs according to the sufficient energy interrupt signal and the energy state decreases, the comparator compares it to the insufficient energy level. If the energy state reaches the insufficient energy level, the insufficient energy interrupt signal is generated and the input of the comparator is switched from the insufficient energy level to the sufficient energy level. By repeating this operation, the comparator can automatically switch the input energy level and compare the energy state to the two energy levels. The energy consumed by the comparator is supplied by the energy buffer.

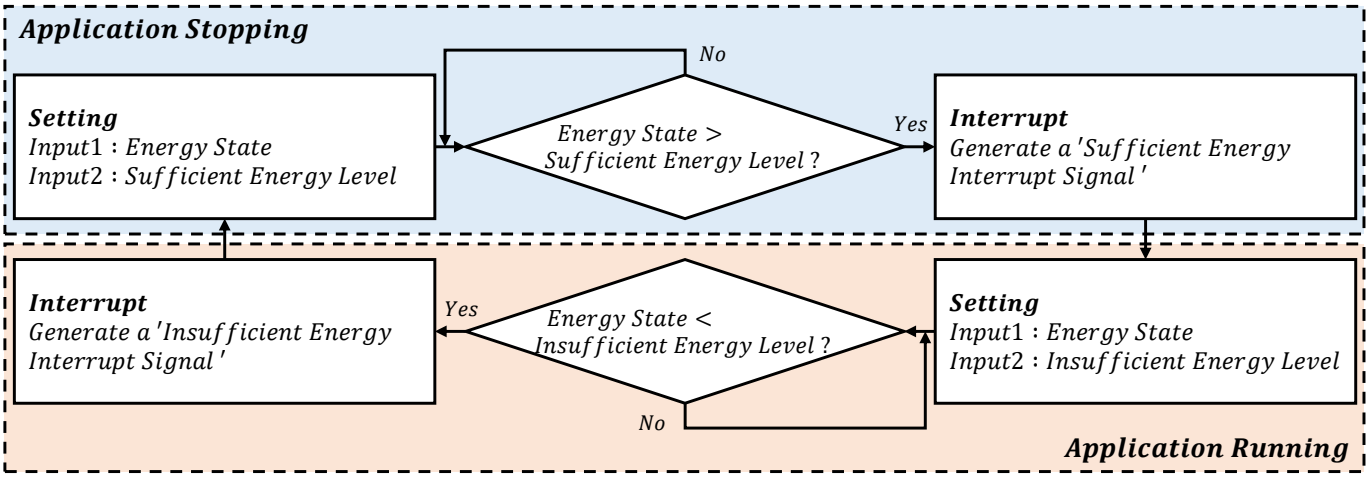

Figure 4. Comparator's function to switch the input energy level automatically.

3.1.2. Controller

The controller controls and manages an application to operate the application normally in intermittent computing. The controller recognizes the energy state through the signals transmitted by the energy checker. It also recognizes the memory state through a flag indicating whether power failure has occurred. The controller consists of operations that manage the application based on the energy state and the memory state: Backup, Restore, Sleep, and Wakeup.

Backup copies the application's operating state to the non-volatile memory. Data allocated to the registers and volatile memory are lost when power failure occurs. If intermediate data affecting the operation of the application is lost, the application cannot operate normally even when the system is turned on again. To prevent this, Backup copies the application's operating state, including all data in the registers and volatile memory, to the non-volatile memory. Figure 5 shows the memory layout of MSP430FR5994 [17], which is a chip used in the experiments. The chip has 256 KB of FRAM and 4 KB of static random access memory (SRAM). FRAM stores program code in some space, and data can be stored in the remaining space. Therefore, when copying the state, Backup saves the entire image of registers and SRAM in the remaining space of the FRAM. Although copying the entire image increases overheads by copying unnecessary memory areas, it is possible to stably store the application's operating state regardless of the application.

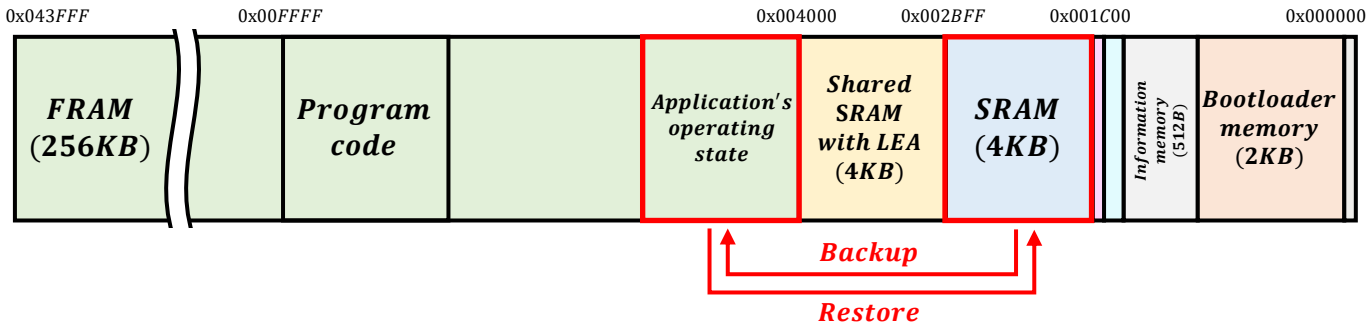

Figure 5. Memory layout of MSP430FR5994 used in the experiments and the controller's operations.

Restore restores the application's operating state. The application's operating state includes the results of the application that was executed before Backup. Therefore, if the application's operating state is restored, the application can continue to operate from the restored state. To restore the state, Restore copies the state stored in the non-volatile memory to the registers and volatile memory. As shown in Figure 5, since the entire image of the registers and SRAM was stored in the FRAM, Restore can restore the image as it is. Like Backup, Restore also increases the overhead by copying the entire image, but it can reliably restore the application's operating state.

Sleep minimizes energy consumption by stopping the application. It reduces the energy consumption to less than the energy that is provided by the energy harvester, thereby increasing the energy state of the device. However, since Sleep maintains the power supplied to the registers and volatile memory, data in the registers and volatile memory are retained. This allows the application to run immediately when the energy is sufficiently charged without power failure.

Wakeup awakens the system from Sleep and restarts the application. When the energy is sufficiently charged, Wakeup starts running the application. Since the data remain in the registers and volatile memory during Sleep, Wakeup does not need to restore the data.

3.2. State Diagram of ICEr

Figure 6 shows the state diagram of ICEr. When the system is turned on with minimal power supply, the energy checker and controller operate independently. The energy checker initializes settings to measure the energy state of the device and generate energy signals. The controller performs Sleep and waits for a signal from the energy checker.

In Sleep, the energy state increases; this is measured by the energy checker. If the energy state is greater than the sufficient energy level, the energy checker generates a sufficient energy interrupt signal and transmits it to the controller. When the controller receives the sufficient energy interrupt signal, it recognizes that there is sufficient energy to execute the application and prepares to execute the application. To execute the application, the application's operating state must exist in the registers and volatile memory. The controller performs the memory-state check by checking the flag to verify it. If the flag indicates that power failure has occurred, the data in the registers and volatile memory are lost. In this case, the controller performs Restore to restore the application's operating state. On the contrary, if the flag indicates that no power failure has occurred, data remain in the registers and volatile memory and the controller performs Wakeup. After the controller is ready to execute the application, it runs the application.

The energy state decreases when the application runs because the energy consumed to run the application is greater than the harvested energy. If the energy state is less than the insufficient energy level, the energy checker generates and transmits an insufficient energy interrupt signal. When the controller receives this signal, it recognizes that there is insufficient energy to execute the application. Therefore, the controller stops the application and performs Backup. After Backup is finished, the controller performs Sleep to minimize energy consumption. If the energy-harvesting environment is good, the energy harvester

harvests enough energy and the energy state of the device increases. In contrast, if the energy-harvesting environment is poor, the energy harvester does not harvest enough energy and the energy state decreases, even in Sleep. This can lead to energy depletion and cause a system shutdown due to power failure. When the system is turned off, data in the registers and volatile memory are lost and the system remains in the off-state until minimal energy is charged.

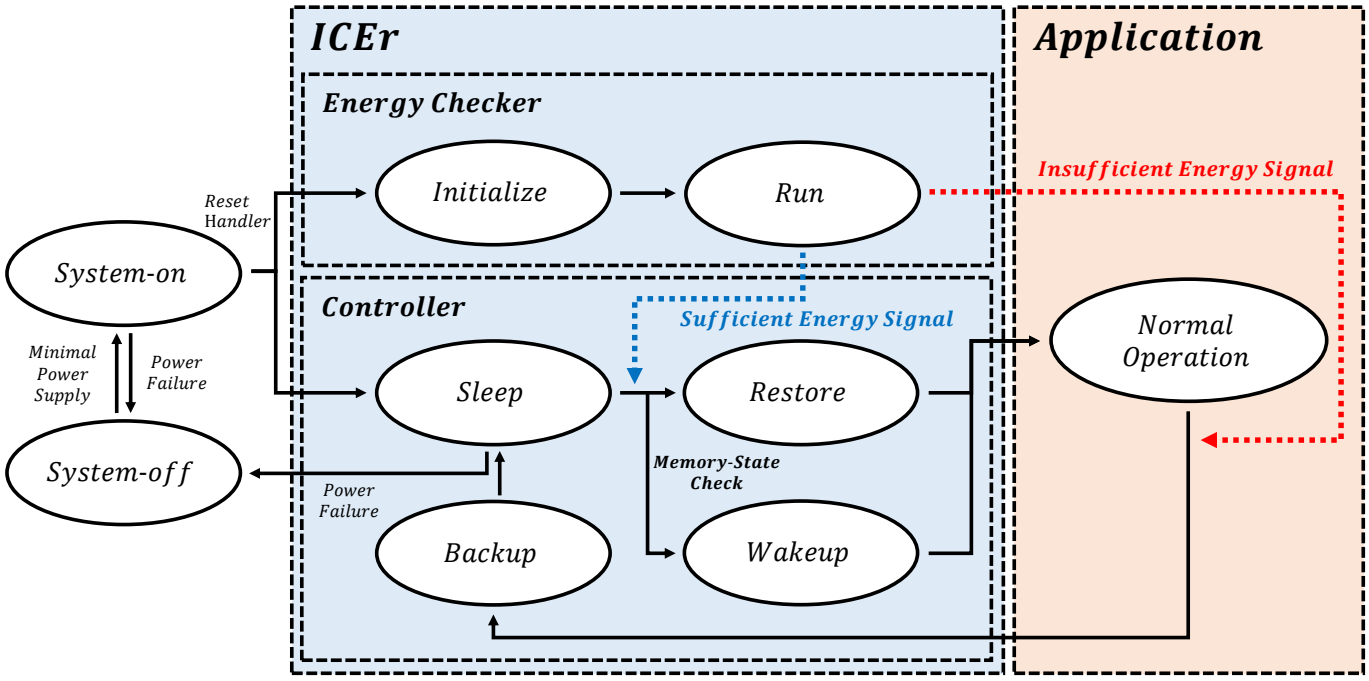

Figure 6. A state diagram of ICEr.

3.3. Forward Progress and Data Consistency

ICEr ensures forward progress of an application. When the energy state is insufficient, ICEr copies the application's operating state to the non-volatile memory and stops the application. Then, it waits until the energy state is sufficient to run the application. When the energy state becomes sufficient, ICEr performs the memory state check and restores the application's operating state if no data remain in the memory. Then, it executes the application again from the point where it stopped working. In this process, ICEr's energy checker operates continuously over time, so the energy state is always checked and sudden changes are not missed. Therefore, ICEr controls and manages the application in real-time according to the energy state and executes it in the forward direction without re-execution.

Data consistency is also ensured by ICEr. Data consistency can be broken in the process of overwriting some of the data stored in the non-volatile memory. However, ICEr always copies the entire image when the application's operating state is stored to the non-volatile memory. Therefore, all data stored in the non-volatile memory are simultaneously updated and data consistency is maintained.

3.4. Energy Efficiency

The total energy consumed by ICEr is the sum of the energy consumed by the energy checker and the controller. Since the energy checker continuously checks the energy state over time, it statically consumes energy. However, the controller operates dynamically according to the device and environment. It minimizes the number of Backup and Restore operations through the energy state check and avoids the execution of unnecessary Restore operations through the memory state check. As a result, the dynamic operation of the controller increases the energy efficiency of ICEr by reducing the energy consumption.

In intermittent computing, typical applications of IoT devices repeat the network connection for wireless communication every time the device is switched off and on again, consuming a lot of energy. However, since the application's operating state includes the network connection information, ICEr's Backup and Restore operations enable communication without a new network connection. Therefore, the number of network connections can be reduced. Since the energy consumed to restore the application's operating state is significantly less than the energy consumed for network connection, the overall energy consumption decreases and the energy efficiency increases.

3.5. Programmability and Portability

ICEr operates independently of an application. It controls the application according to the state of the device and copies the entire memory image to backup and restores the application's operating state. Therefore, ICEr can be applied to any application, regardless of the type of application. In this process, since ICEr considers the state of the device, the application does not need to consider this. It allows programmers to design the application with only operational considerations and does not require them to modify the application according to the hardware. This increases programmability and shortens the development time for the application.

Since the sufficient/insufficient energy levels of ICEr are determined by hardware, ICEr is hardware-dependent. Therefore, if the platform on which ICEr operates is changed, the energy levels of ICEr must be set again. The sufficient energy level should be determined at an appropriate level within the operating voltage range of the device to indicate the energy state that is sufficient for the device to run the application. The insufficient energy level should be higher than the device's minimum operating voltage enough to reliably perform data backup. Although ICEr needs to be modified depending on the platform, only the energy levels need to be reset according to the device's operating voltage range. This increases portability.

4. Experimental Results and Discussion

In this study, two kinds of experiments were conducted to validate the proposed ICEr. The first experiment validated the operation of ICEr and measured the performance, and the second experiment showed how ICEr affects energy efficiency for a typical application of an IoT device.

4.1. Experimental Environment

To validate the proposed ICEr, intermittent computing environments and an IoT device were constructed and ICEr was applied to it. The IoT device was composed of TI's MSP-EXP430FR5994 LaunchPad Development Kit [17]. The onboard Micro Controller Unit (MCU) has a 16-bit RISC architecture and operates and a clock speed of up to 16 MHz. It has a wide operating voltage range of 1.8–3.6 V and the memory layout as in Figure 5. It also has a comparator, which can continuously compare the energy state of the device with the reference voltage. The board has a 0.22F supercapacitor, which can store the harvested energy. The board has an active mode to run an application and a low power mode to stop the application and minimize energy consumption. The basic current consumption of the board in active mode is 0.233 mA, and the basic current consumption in low power mode is 0.089 mA. We experimentally set the sufficient energy level to 2.6 V and the insufficient energy level to 2.0 V within the operating voltage range.

The intermittent computing environments were built using a function generator. The harvested energy is difficult to control; hence, it is not suitable for validating ICEr. The function generator generated sine waves, which make abstract the energy states of the supercapacitor that is affected by the energy harvested from various energy harvesters and the energy consumed by the device. Since the voltage of the supercapacitor represents the energy stored in the supercapacitor, **E0**–**E6** controls the voltage to abstract the energy state

of the supercapacitor. Sine waves were not exactly the same as in the graph in Figure 1, but the overall shape was similar. The generated environments were as follows.

- **E0**: 3.3 V DC signal. It denotes an environment with a sufficient energy supply.
- **E1**: A sine wave signal with a period of 3 s. Sine waves ranging from 1.8 V to 2.6 V repeat. It denotes an environment where the charging and discharging speed is fast and power failure does not occur.
- **E2**: A sine wave signal with a period of 3 s. Four sine waves ranging from 1.8 V to 2.6 V and one sine wave ranging from 1.5 V to 2.6 V repeat. It denotes an environment where the charging and discharging speed is fast and power failure occasionally occurs.
- **E3**: A sine wave signal with a period of 3 s. Sine waves ranging from 1.5 V to 2.6 V repeat. It denotes an environment where the charging and discharging speed is fast and power failure continuously occurs.
- **E4**: A sine wave signal with a period of 5 s. Sine waves ranging from 1.8 V to 2.6 V repeat. It denotes an environment where the charging and discharging speed is slow and power failure does not occur.
- **E5**: A sine wave signal with a period of 5 s. Four sine waves ranging from 1.8 V to 2.6 V and one sine wave ranging from 1.5 V to 2.6 V repeat. It denotes an environment where the charging and discharging speed is slow and power failure occasionally occurs .
- **E6**: A sine wave signal with a period of 5 s. Sine waves ranging from 1.5 V to 2.6 V repeat. It denotes an environment where the charging and discharging speed is slow and power failure continuously occurs.

A wireless communication environment using Zigbee was also constructed [26]. Zigbee is a wireless communication technology that operates at low power; thus, it is suitable for communication between edge devices [27,28]. Digi's XBee S2C module for Zigbee was used. The module uses the Zigbee Mesh protocol and performs the network connections by setting parameters.

4.2. Operation Validation and Performance Measurement

4.2.1. Benchmarks

To validate ICEr and measure its performance, Embench's benchmarks were selected as target applications. Embench is a collection of benchmarks for embedded platforms, and its benchmarks are designed to measure performance [18]. Therefore, it is suitable for validating the operation of ICEr and measuring the performance. The following benchmarks were used in the experiment.

- crc32: A benchmark calculating 32-bit Cyclic Redundancy Check (CRC).
- edn: A benchmark performing vector multiplication.
- nbody: A benchmark solving multibody problems that find the future motion states based on mass, initial position, and initial velocity of multiple objects.
- picojpeg: A benchmark decompressing Joint Photographic Experts Group (JPEG) files.
- st: A benchmark calculating the mean, variance, standard deviation, and correlation coefficient of two arrays.
- ud: A benchmark testing the impact of the conditional flow.

4.2.2. Execution Time

Figure 7 illustrates the relative execution time for each benchmark. Each benchmark was executed in an environment with a sufficient energy supply (**E0**) and various intermittent computing environments (**E1**–**E6**). In each environment, the active execution time was measured when the benchmark was unchanged (Plain C), when Hibernus was applied, and when the proposed ICEr was applied. Then, the active execution time was normalized by the execution time of Plain C in **E0**.

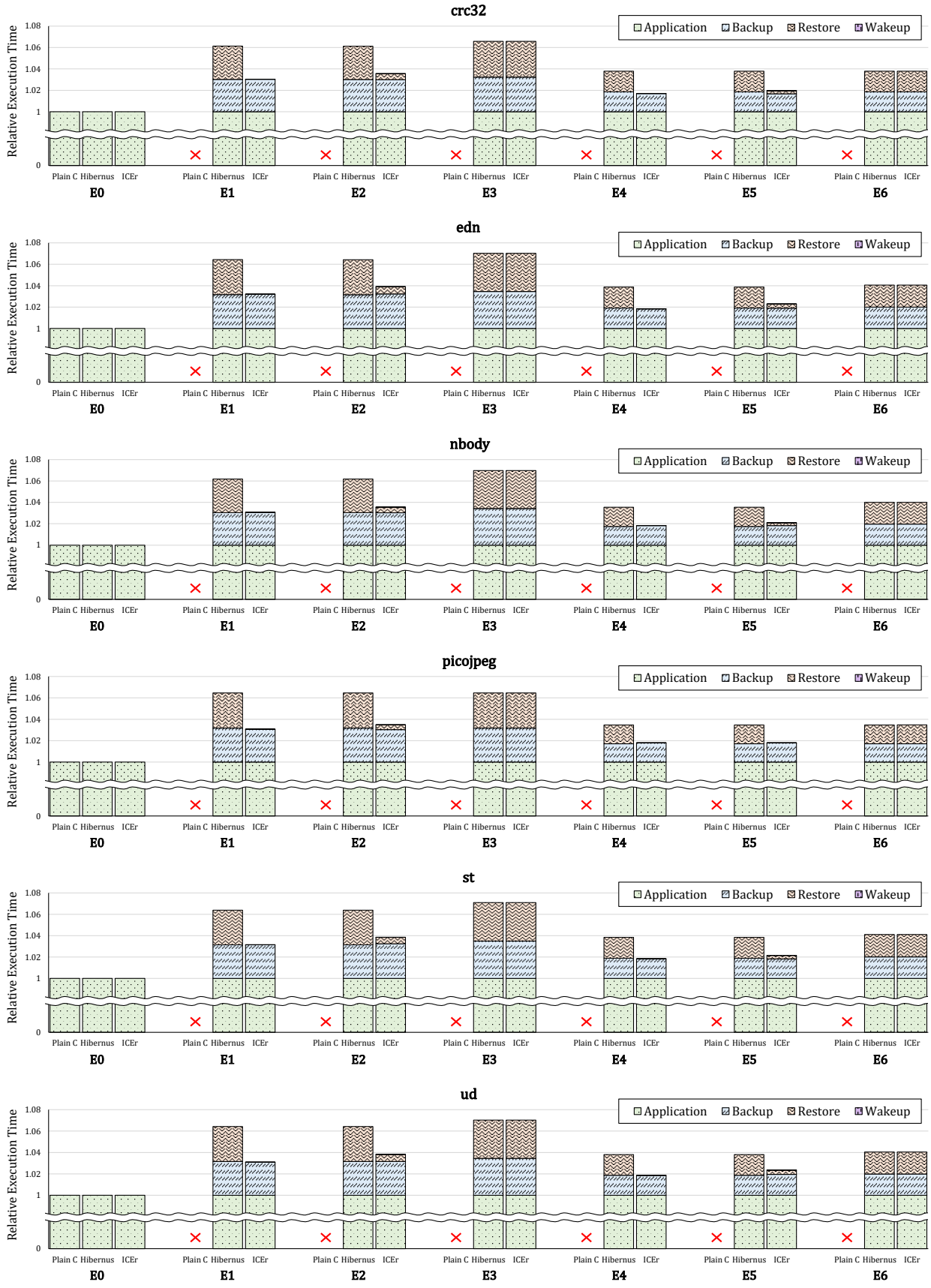

Figure 7. Relative execution time for each benchmark.

For all benchmarks, Plain C ran normally in **E0** because sufficient energy was supplied. However, in the intermittent computing environments (**E1**–**E6**), Plain C did not run normally due to power failure. Since Plain C does not consider the energy state, it does not handle power failure. Therefore, whenever power failure occurs, Plain C re-executes the benchmark from the beginning. Consequently, it did not finish due to repetitive power failures; thus, the execution time could not be measured.

When sufficient energy is supplied, Hibernus does not perform any additional operations. Therefore, for all benchmarks, the relative execution time of Hibernus was the same as that of Plain C in **E0**. When Hibernus runs in intermittent computing environments, the relative execution time for the application is 1, as it does not re-execute the application. However, it additionally performs Backup and Restore to handle power failure. Moreover, since it does not consider the memory state, Restore is executed whenever the energy state becomes sufficient. Therefore, the relative execution time based on the frequency of power failures varies minimally. For all benchmarks, Hibernus hasdthe relative execution time of about 1.06 in **E1**–**E3** and the relative execution time of about 1.04 in **E4**–**E6**.

For the same reason as Hibernus, ICEr had the same relative execution time as Plain C in **E0**. Furthermore, the relative execution time for the application is 1 in intermittent computing environments because ICEr does not re-execute the application. However, when performing Restore, ICEr considers not only the energy state but also the memory state. If no power failure occurs and data remain in the registers and volatile memory, Wakeup is performed with significantly less of a time overhead compared to Restore. As a result, ICEr had a different relative execution time depending on the intermittent computing environment. In **E1** and **E4**, wherein power failure did not occur, ICEr's relative execution time overhead was about 0.03, which is 0.03 shorter than that of Hibernus. This is a reduction of about 50% in the relative execution time overhead, which is the largest reduction in the intermittent computing environments. However, in **E3** and **E6**, wherein power failure continuously occurred, the relative execution time overheads of ICEr and Hibernus were almost the same.

4.2.3. Number of ICEr Operations

Table 1 affords the number of Backup, Restore, and Wakeup operations performed when running each benchmark in intermittent computing environments. Since the intermittent computing environments comprise repetitions of sine waves, the total number of operations is proportional to the execution time of the benchmark. In all intermittent computing environments, the numbers of Backup operations for Hibernus and ICEr were almost the same. However, the numbers of Restore and Wakeup operations were different. Hibernus did not perform Wakeup but performed Restore whenever the energy state became sufficient to operate the application. Therefore, regardless of the environment, Hibernus had the same number of Backup and Restore operations. On the other hand, ICEr properly performed Wakeup instead of Restore, depending on the environment. When power failure did not occur and data remained in the registers and volatile memory, Wakeup was performed. The fewer the number of power failures, the fewer the number of Restore operations. Therefore, depending on the environment, ICEr minimizes the number of Restore operations, which has higher time and energy overheads than Wakeup.

Table 1. Number of ICEr operations for each benchmark.

		E1		E2		E3		E4		E5		E6	
		H	**I**	**H**	**I**	**H**	**I**	**H**	**I**	**H**	**I**	**H**	**I**
crc32	# of Backup	12	11	12	11	13	13	7	6	7	6	7	7
	# of Restore	12	0	12	2	13	13	7	0	7	1	7	7
	# of Wakeup	·	11	·	9	·	0	·	6	·	5	·	0
edn	# of Backup	37	34	37	35	41	41	21	19	21	20	22	22
	# of Restore	37	0	37	7	41	41	21	0	21	4	22	22
	# of Wakeup	·	34	·	28	·	0	·	19	·	16	·	0
nbody	# of Backup	13	12	13	12	15	15	7	7	7	7	8	8
	# of Restore	13	0	13	2	15	15	7	0	7	1	8	8
	# of Wakeup	·	12	·	10	·	0	·	7	·	6	·	0
picojpeg	# of Backup	8	7	8	7	8	8	4	4	4	4	4	4
	# of Restore	8	0	8	1	8	8	4	0	4	0	4	4
	# of Wakeup	·	7	·	6	·	0	·	4	·	4	·	0
st	# of Backup	23	21	23	22	26	26	13	12	13	12	14	14
	# of Restore	23	0	23	4	26	26	13	0	13	2	14	14
	# of Wakeup	·	21	·	18	·	0	·	12	·	10	·	0
ud	# of Backup	27	24	27	25	30	30	15	14	15	15	16	16
	# of Restore	27	0	27	5	30	30	15	0	15	3	16	16
	# of Wakeup	·	24	·	20	·	0	·	14	·	12	·	0

H: Hibernus, **I**: ICEr.

4.2.4. Energy Consumption

In intermittent computing environments, the system's energy state is important. However, it is difficult to measure the system's energy state in run-time without affecting it [29]. Therefore, we measured the average current consumption for each operation. Then, we calculated the total energy consumption by multiplying the execution time and the average current consumption of each operation.

Table 2 illustrates the average current consumption for each operation, measured using EnergyTrace, a power analyzer tool that measures and displays the energy profile of an application. The reason for measuring the current consumption is that the current consumption for each operation is almost constant even if the energy state of the device is changed. The energy checker consumes current to generate the sufficient/insufficient energy levels through a reference voltage generator and runs a comparator. Backup consumes current to copy the application's operating state to the non-volatile memory, whereas Restore consumes current to restore the application's operating state from the non-volatile memory. Wakeup consumes little current because it only changes the operation mode without moving data. The memory state check is not displayed in Table 2 because it consumes too little energy by simply checking the flag.

Table 2. Average current consumption of each ICEr operation.

	Energy Checker	**Backup**	**Restore**	**Wakeup**
Average current consumption (mA)	0.039	0.13	0.088	0.001

Figure 8 depicts the energy overhead consumed by each operation for the benchmarks. In **E0**, since both Hibernus and ICEr did not perform any additional operations, no energy overhead was present. On the other hand, in intermittent computing environments, an energy overhead was present as they both performed additional operations. Hibernus and ICEr consumed similar energy for Backup for all benchmarks due to the similar number of Backup operations. However, the energy consumed for Restore was different for each intermittent computing environment. Since Hibernus performed Restore regardless of the environment, a lot of energy was consumed for Restore. On the other hand, based on the environment, ICEr minimizes the number of Restore operations and performs Wakeup with little energy consumption. Therefore, ICEr reduces the energy consumed for Restore,

thereby reducing the overall energy overhead. Compared to Hibernus, ICEr's maximum energy overhead reduction was about 49.5% in **E4** for crc32.

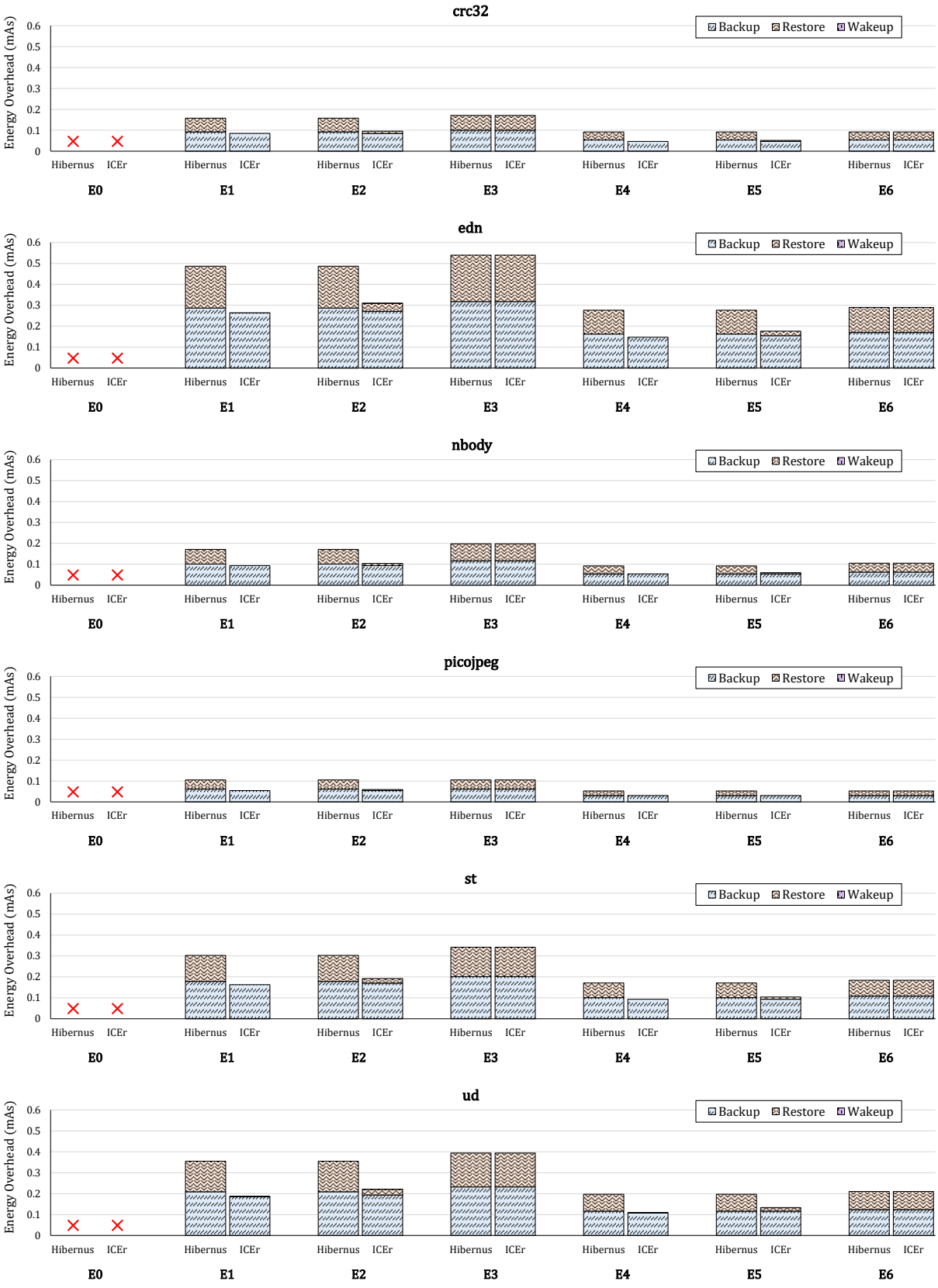

Figure 8. The energy overhead for each benchmark.

4.2.5. Memory Usage

Table 3 shows the memory usage of Hibernus, ICEr, and the benchmarks. FRAM usage indicates the code size because the code is stored in it. The code size and SRAM usage of ICEr are almost identical to those of Hibernus. However, ICEr's code size was 64% of crc32's code size, which had the smallest code size, and 11% of picojpeg's code size, which had the largest code size. Moreover, ICEr's SRAM usage was almost identical to crc32's SRAM usage, which was the smallest SRAM usage, and 7% of picojpeg's SRAM usage, which was the largest SRAM usage. Since ICEr is independent of the application, ICEr's code size and SRAM usage are also independent of the application. Therefore, ICEr's memory usage remains unchanged even for large-sized applications.

Table 3. Memory usage of Hibernus, ICEr, and the benchmarks.

	Hibernus	**ICEr**	**crc32**	**edn**	**nbody**	**picojpeg**	**st**	**ud**
FRAM (byte)	1046	1088	1700	3940	7642	9514	7634	1778
SRAM (byte)	168	170	174	1768	482	2552	1788	1922

4.3. Utilization in a Real IoT Application

4.3.1. Application

To validate the effectiveness of ICEr for a real IoT application, the *Temperature Measurement Application* was selected as the target application. The *Temperature Measurement Application* measures the temperature of the surrounding environment and transmits it to a server. The application comprises network connections, temperature measurements, and data transmissions. Network connections for wireless communications consume a lot of energy. However, connection is performed only once at the beginning of the application. Thereafter, the application repeats the temperature measurement and data transmission. These operations are periodically performed every second.

4.3.2. Energy Consumption

Figure 9 shows the energy overhead consumed to operate the *Temperature Measurement Application* in various environments. The average current consumption and execution time for the operations were multiplied to calculate the energy overheads. In each environment, the average current consumption was measured by EnergyTrace when the target application was unchanged (Plain C), when Hibernus was applied, and when ICEr was applied. Furthermore, to quantitatively measure the execution time, the temperature measurement and data transmission were executed 30 times.

Plain C operated without power failure due to the sufficient energy supply in **E0**. Since the system was not turned off, the network connection for wireless communication was performed only once at the beginning of the application, and the energy of 0.762 mAs was consumed for this operation. Then, the temperature measurement and data transmission were periodically performed 30 times, and the energies of 0.146 mAs and 0.457 mAs were consumed, respectively. On the other hand, in intermittent computing environments, power failures continuously occurred because Plain C operates without considering the energy state. When the system is turned on again after power failure, Plain C executes the application from the beginning. Consequently, the network connection was performed multiple times. In **E1**–**E3**, where power failure frequently occurred due to fast charging and discharging, the energy consumption for the network connection was about 12.954 mAs. In **E4**–**E6**, where power failure occurred less frequently due to slow charging and discharging, the energy consumption for the network connection was about 7.620 mAs. However, the energy consumptions for the temperature measurement and data transmission were almost identical to those in **E0** because the number of these operations was the same in all environments.

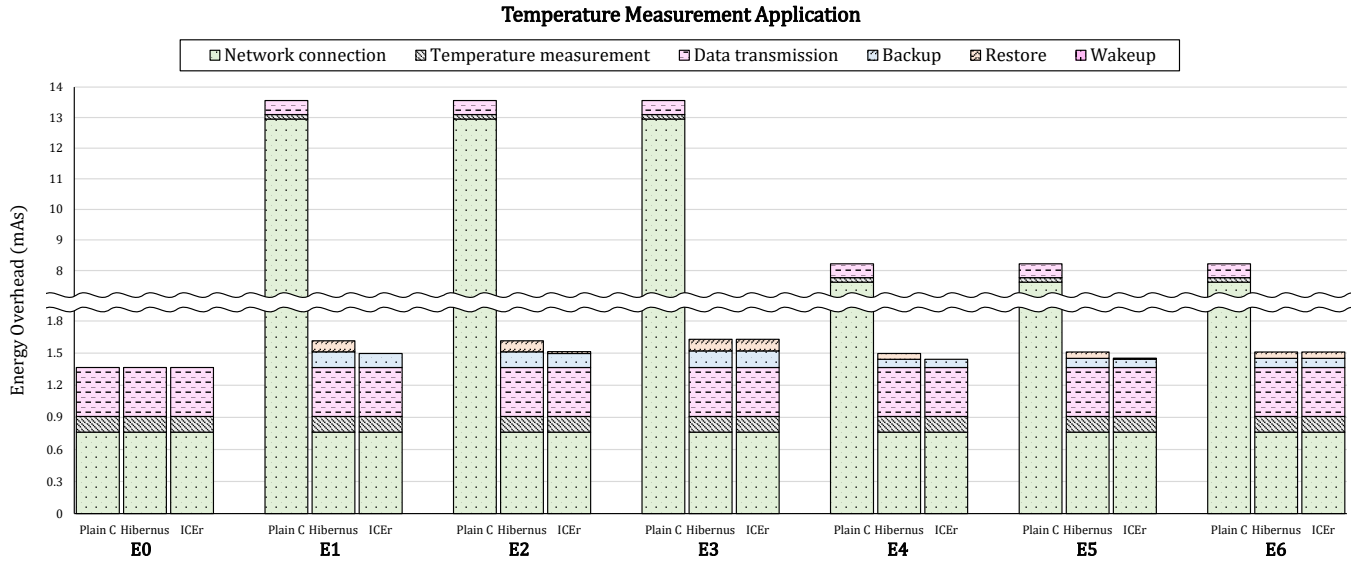

Figure 9. Energy overhead for the *Temperature Measurement Application*.

In **E0**, Hibernus and ICEr ran the application without any additional operations. Therefore, their energy consumptions were similar to that of Plain C. However, in intermittent computing environments, they copied the application's operating state to the non-volatile memory when the energy state was insufficient to execute the application. Since the application's operating state includes network connection information, its restoration enables wireless communication without a new network connection. Consequently, Hibernus and ICEr performed the network connection only once, and the energy of 0.762 mAs was consumed for this operation, which is significantly less than the energy consumed by Plain C in the same environment. On the other hand, the energy they consumed for the temperature measurement and data transmission was almost identical to used by Plain C due to the same number of executions. The additional operations performed to control and manage the application have little impact on the overall energy overhead because the energy consumed by these operations is small. Therefore, in intermittent computing environments, Hibernus and ICEr have similar energy overheads, and they significantly reduce the energy consumption for the network connection compared to Plain C. In particular, they showed an 89% reduction in the energy overhead in **E1**. ICEr also reduced the energy overhead by up to 7.3% over Hibernus through the memory state check in **E1**.

5. Conclusions

In this paper, we proposed ICEr, a low-power intermittent computing environment that dynamically controls and manages an application. It comprises an energy checker and a controller. The energy checker measures the energy state, while the controller controls and manages the application by performing Backup, Restore, Sleep, or Wakeup based on the energy state and the memory state. Two kinds of experiments validated ICEr's operation and performance. The first experiment showed that ICEr executes Embench's benchmarks normally in various intermittent computing environments. Moreover, it also showed that ICEr avoids unnecessary restore operations through the memory state check, reducing the relative execution time overhead by up to 50% and the energy overhead by up to 49.5% compared to Hibernus. The second experiment with *Temperature Measurement Application* confirmed that ICEr improves the energy efficiency for the real IoT application. ICEr reduced the energy overhead by up to 89% compared to Plain C by reducing the number of new network connections, and by up to 7.3% compared to Hibernus through the memory state check.

The proposed ICEr systematically supports dynamic execution to operate the application normally in intermittent computing. Moreover, it increases the energy efficiency by

controlling the application in consideration of the memory state that is the major state of the device. Our study showed that ICEr executes the monitoring application energy-efficiently on the energy-harvesting based IoT device. Therefore, we expect our achievements to be applied to energy-harvesting based IoT systems to build a battery-free monitoring system.

Author Contributions: J.K. wrote the entire manuscript and performed model design, hardware and software implementation, and formal analysis; H.K. assisted in related model research and analysis; J.C. was the principal investigator and is the corresponding author. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2020R1A2C1013836).

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

JIT	Just-In-Time
ICEr	Intermittent Computing Environment based on a run-time module
IoT	Internet of Things
NVRAM	non-volatile random access memory
EEPROM	electrically erasable programmable read-only memory
RAM	random access memory
MRAM	magnetoresistive random access memory
FRAM	ferroelectric random access memory
WAR	Write-After-Read
SRAM	static random access memory
MCU	Micro Controller Unit
CRC	Cyclic Redundancy Check
JPEG	Joint Photographic Experts Group

References

1. Liang, Y.; Yu, L. Development of Semiconducting Polymers for Solar Energy Harvesting. *Polym. Rev.* **2010**, *50*, 454–473. [CrossRef]
2. Lallart, M.; Cottinet, P.J.; Guyomar, D.; Lebrun, L. Electrostrictive polymers for mechanical energy harvesting. *J. Polym. Sci. Part B Polym. Phys.* **2012**, *50*, 523–535. [CrossRef]
3. Sample, A.P.; Yeager, D.J.; Powledge, P.S.; Mamishev, A.V.; Smith, J.R. Design of an RFID-Based Battery-Free Programmable Sensing Platform. *IEEE Trans. Instrum. Meas.* **2008**, *57*, 2608–2615. [CrossRef]
4. Karagozler, M.E.; Poupyrev, I.; Fedder, G.K.; Suzuki, Y. Paper Generators: Harvesting Energy from Touching, Rubbing and Sliding. In Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology, St. Andrews, UK, 8–11 October 2013; Association for Computing Machinery: New York, NY, USA, 2013; pp. 23–30. [CrossRef]
5. Paradiso, J.A. Systems for human-powered mobile computing. In Proceedings of the 43rd ACM/IEEE Design Automation Conference, San Francisco, CA, USA, 24–28 July 2006; pp. 645–650. [CrossRef]
6. Juang, P.; Oki, H.; Wang, Y.; Martonosi, M.; Peh, L.S.; Rubenstein, D. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. *SIGOPS Oper. Syst. Rev.* **2002**, *36*, 96–107. [CrossRef]
7. Manchester, Z. KickSat. 2015. Available online: https://kicksat.github.io/ (accessed on 4 March 2021).
8. Lucia, B.; Balaji, V.; Colin, A.; Maeng, K.; Ruppel, E. Intermittent Computing: Challenges and Opportunities. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*; Lerner, B.S., Bodík, R., Krishnamurthi, S., Eds.; Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik: Dagstuhl, Germany, 2017; Leibniz International Proceedings in Informatics (LIPIcs); Volume 71, pp. 8:1–8:14. [CrossRef]
9. Colin, A.; Lucia, B. 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, The Netherlands, 30 October–4 November 2016; Association for Computing Machinery: New York, NY, USA, 2016; pp. 514–530. [CrossRef]
10. Maeng, K.; Colin, A.; Lucia, B. Alpaca: Intermittent Execution without Checkpoints. *arXiv* **2019**, arXiv:1909.06951.

11. Woude, J.V.D.; Hicks, M. Intermittent Computation without Hardware Support or Programmer Intervention. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, USA, 2–4 November 2016; USENIX Association: Savannah, GA, USA, 2016; pp. 17–32.
12. Maeng, K.; Lucia, B. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, USA, 8–10 October 2018; USENIX Association: Carlsbad, CA, USA, 2018; pp. 129–144.
13. Ahmed, S.; Alizai, M.H.; Siddiqui, J.H.; Bhatti, N.A.; Mottola, L. Poster Abstract: Towards Smaller Checkpoints for Better Intermittent Computing. In Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), Porto, Portugal, 11–13 April 2018; pp. 132–133. [CrossRef]
14. Ransford, B.; Sorber, J.; Fu, K. Mementos: System Support for Long-Running Computation 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, 5–11 March 2011; Association for Computing Machinery: New York, NY, USA, 2011; pp. 159–170. [CrossRef]
15. Balsamo, D.; Weddell, A.S.; Merrett, G.V.; Al-Hashimi, B.M.; Brunelli, D.; Benini, L. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *IEEE Embed. Syst. Lett.* **2015**, *7*, 15–18. [CrossRef]
16. Maeng, K.; Lucia, B. Supporting Peripherals in Intermittent Systems with Just-in-Time Checkpoints. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, 22–26 June 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 1101–1116. [CrossRef]
17. MSP-EXP430FR5994 LaunchPad Development Kit. Available online: https://www.ti.com/tool/MSP-EXP430FR5994 (accessed on 5 March 2021).
18. Foundation, F. Embench: A Modern Embedded Benchmark Suite. 2019. Available online: https://github.com/embench (accessed on 5 March 2021).
19. Harb, A. Energy harvesting: State-of-the-art. *Renew. Energy* **2011**, *36*, 2641–2654. [CrossRef]
20. Stankovic, J.A. Wireless Sensor Networks. *Computer* **2008**, *41*, 92–95. [CrossRef]
21. Estrin, D.; Girod, L.; Pottie, G.; Srivastava, M. Instrumenting the world with wireless sensor networks. In Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, Salt Lake City, UT, USA, 7–11 May 2001; Proceedings (Cat. No.01CH37221); IEEE: Piscataway, NJ, USA, 2001; Volume 4, pp. 2033–2036. [CrossRef]
22. de Meulenaer, G.; Gosset, F.; Standaert, F.; Pereira, O. On the Energy Cost of Communication and Cryptography in Wireless Sensor Networks. In Proceedings of the IEEE International Conference on Wireless and Mobile Computing, Networking and Communications, Avignon, France, 12–14 October 2008; pp. 580–585. [CrossRef]
23. Jayakumar, H.; Raha, A.; Raghunathan, V. Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in IoT Edge Devices. In Proceedings of the 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID), Kolkata, India, 4–8 January 2016; pp. 264–269. [CrossRef]
24. Kawashima, S.; Cross, J.S. FeRAM. In *Embedded Memories for Nano-Scale VLSIs*; Zhang, K., Ed.; Springer: Boston, MA, USA, 2009; pp. 279–328. [CrossRef]
25. Qazi, M.; Clinton, M.; Bartling, S.; Chandrakasan, A.P. A Low-Voltage 1 Mb FRAM in 0.13 µm CMOS Featuring Time-to-Digital Sensing for Expanded Operating Margin. *IEEE J. Solid-State Circ.* **2012**, *47*, 141–150. [CrossRef]
26. Safaric, S.; Malaric, K. ZigBee wireless standard. In Proceedings of the Proceedings ELMAR 2006, Zadar, Croatia, 7–10 June 2006; pp. 259–262. [CrossRef]
27. Gill, K.; Yang, S.; Yao, F.; Lu, X. A zigbee-based home automation system. *IEEE Trans. Consum. Electron.* **2009**, *55*, 422–430. [CrossRef]
28. Wheeler, A. Commercial Applications of Wireless Sensor Networks Using ZigBee. *IEEE Commun. Mag.* **2007**, *45*, 70–77. [CrossRef]
29. Colin, A.; Harvey, G.; Sample, A.P.; Lucia, B. An Energy-Aware Debugger for Intermittently Powered Systems. *IEEE Micro* **2017**, *37*, 116–125. [CrossRef]