A Survey of Intermittent Computing in Energy Harvesting Devices

Cennet Tugce Turan
University of California, Los Angeles
cennettugce@gmail.com

Abstract—**Batteryless solutions have been emerging as an alternative to heavy, costly and unsustainable battery usage in devices. Energy harvesting is used to power low energy devices; however, the implementation of energy harvesting requires special attention from the programmers because of the intermittently operational nature of batteryless devices. An energy buffer should be added, and the problems caused by power failure should be handled at the software level. This paper provides a survey on solutions brought to the design space to enable intermittent computing in energy harvesting devices. The problems with intermittent computing is explained, the solution methods are compared and presented, and the various application areas are surveyed.**

I. Introduction

Most of the devices in the market require the use of batteries, which add considerable size, weight and cost to the device. More importantly, they are not sustainable and require frequent charging. As an alternative to the high amount of battery usage in low-power operating devices, energy harvesting was introduced as an environmental, practical solution that eliminates the battery requirement. An energy harvesting device uses the energy from its environment without the necessity of a battery. The source of energy can be sunlight, radio waves, motion, thermal gradient, piezoelectric, etc. [1]. Energy harvesting can be used in IoT devices, wearables, medical sensors, implantable devices, monitors, and satellites.

Energy harvesting devices operate and buffer energy when the energy is available in the environment, and lose power until sufficient energy is loaded to the buffer. Therefore, energy harvesting devices operate intermittently, meaning that their volatile memory will get lost unpredictably. The disturbances in runtime leaves memory and peripherals in an undetermined state. The stock frame gets lost, which results in impossible outcomes for a conventional execution. If the continuous energy interval is never reached for the task, the system might not progress and may try to execute the same code repeatedly. To be able to overcome these issues, the system for intermittent computing should ensure progress - despite frequent losses in the volatile state - and follow the atomicity constraints to ensure memory consistency. At the same time, the applications should provide adaptation and minimize the program overhead and memory footprint [2].

Programmers designed intermittent computing systems that respond to these issues with different approaches. The two main programming models are *checkpointing* [2, 3] and *task-based programming* [4, 5, 6]. Recently, *interpret language route* [7, 8] has been introduced in order to enable intermittent computing on high-level languages such as Python.

This paper provides a description of intermittent computing, challenges in the field, and a summary of recent work done mostly on the software side of the energy-harvesting devices.

II. System requirements for Intermittent computing

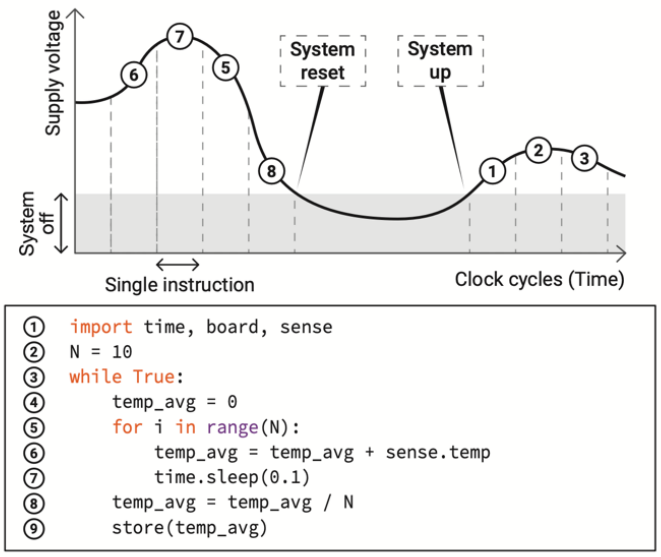
A. Hardware and Storage Requirements

There are different energy harvesters that can be used in different applications. Therefore, hardware adaptation is required such as an energy buffer (capacitor) between the processor and the energy source to keep the execution periodic although the energy input is non-periodic. Different types of applications require different sizes and capacitors, which should be governed by the designer for the application.

Energy distribution of the system should be defined such that the charging mechanism for the capacitor enables optimal behavior for the specific task. After a power failure, the device should wait until the energy buffer reaches to the turn on voltage. Most of the applications sets the turn on voltage to the minimum voltage required for the system such as WISP [9]. However, this might cause issues such that the buffer might never provide enough energy for high energy required tasks.

B. Software Requirements

Intermittent computing requires special management to be able to operate progressively. The execution is interrupted by the power failures and that prevents the program from making progress. Every time a power failure happens, the program starts executing from the *main()* after the power failure. If the interruptions are frequent enough, the program either never makes progress, or it takes a very long time to execute the code, as illustrated in Figure 1. Since the power failures are unpredictable, there is no guarantee of the complete execution of the program.



```
① import time, board, sense
② N = 10
③ while True:
④     temp_avg = 0
⑤     for i in range(N):
⑥         temp_avg = temp_avg + sense.temp
⑦         time.sleep(0.1)
⑧     temp_avg = temp_avg / N
⑨     store(temp_avg)
```

Figure 1. Execution model for an intermittently operating system. Program never reaches to line 9.

Another problem is memory consistency. Even if the volatile memory is recovered after the power failure, the change in non-volatile memory may cause unwanted results. If a variable is updated in non-volatile memory between checkpointing and a power failure, when the system starts from the last checkpoint location, the variable gets updates for the second time.

Atomicity must be satisfied for I/O operations on the code. Sensors requires to be read atomically and intermittency causes data collected to be outdated and useless for the application [1].

III. Programming Models

Various programming models have been introduced to meet the system requirements of intermittent computing and enable batteryless operations on the microcontrollers. WISP was one of the first building blocks of intermittent computing [9]. It enabled microprocessor operation and sensor integration for an RFID-based energy harvesting system. It used the MSP430 microcontroller, which is also used in many other intermittent computing applications, such as Mementos [2]. WISP has an event-driven system that sleeps between the events to preserve power. After WISP, checkpointing, task-based algorithms, and adaptation models enabled faster and more promising executions.

A. Checkpointing

Checkpointing is introduced to bring a solution to the control-flow problem by saving the program state to non-volatile memory before the power failure happens, and recovering the state after the reboot. *Checkpointing* comes with its own challenges, such as when to checkpoint. Mementos introduces an energy-aware checkpointing algorithm for RFID systems. It estimates the available energy during compile time and inserts checkpoints to the system during runtime. Mementos also implements a simulator to analyze the performance of the intermittent operating systems [2].

Checkpointing does not always guarantee memory consistency. It adds an overhead to the code, which increases the execution time and energy, leading the program to make slower progress. DICE comes with a plug-in system that reduces the memory cost of checkpointing [3].

DICE introduces a checkpointing mechanism that only saves the changes that has occurred in volatile data, instead of saving all the global variables repeatedly on every checkpoint. This way, the cost of checkpointing on the system memory and the overall time spent is reduced significantly [3]. DICE is a plug-in system that can be used in different checkpointing systems such as Mementos. The system reduces the data written on non-volatile memory as well as peak energy demand during checkpointing and checkpointing frequency, since it will happen only if there is a change in the volatile memory. Because there is less time to spend for checkpointing, the time to complete a workload is reduced as well.

There is progress for programs that use only volatile state. However, non-volatile memory accumulates errors across reboots if checkpointing happens before a variable in non-volatile memory is updated, and a reboot takes place after the update. When the program starts from the last checkpoint location, the variable will be updated twice, such as *r1* in Figure 2. This is not a possible scenario for continuous execution programs and it leads to various problems in the execution. Memento suffers from this problem: the programmer doesn't know where the last checkpointing happened before the power failure.

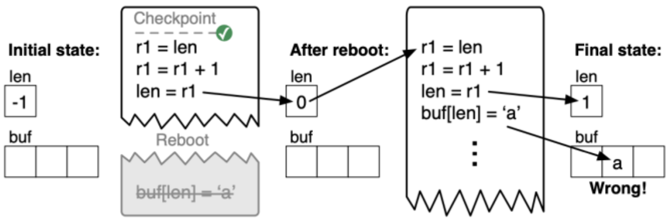


Figure 2. Impossible scenario in continuous computing.

B. Task-Based Programming

The task-based approach divides the program into atomic code blocks. Every time a reboot happens, the programs starts from the uncompleted task. The execution model tracks volatile and non-volatile memory states, which was a problem with *checkpointing*. The first system to ensure the consistency of memory is DINO [4], which promises the program execution is the same as continuous execution by keeping the non-volatile memory constant during an execution of a task, and only updating the non-volatile memory after the completion. This way, the atomicity of the task and the sensor stability is satisfied.

Only a year later, Chain [5] was introduced. This task-based model connects the tasks with channels and performs all the I/O operations through the channels. This way the program promises memory consistency, atomicity and sensor stability while increasing the performance two to seven times compared to DINO [4]. Chain can be implanted for machine learning applications, encryption, compression and sensing [5].

Alpaca being similar to previous applications for task-based models, enables programmer to control where the task begins and ends [6]. Alpaca ensures memory consistency and atomicity by using privatization, which keeps the update on a buffer and updates the main memory at the end of the execution of the task. Alpaca shows up to 15x improvement in runtime; and by avoiding static multi versioning, 5.5x improvement on memory usage [6].

Task-based models show state of art improvement in runtime, memory consistency, memory consumption compered to *checkpointing*. However, task-based models require special attention from the programmer for the code to be divided into tasks. This is a time-consuming process and requires expert knowledge, which prevents intermittent computing to be widely used and available for everyone. If the division of tasks are not well done and the energy is not sufficient for the task to complete, the task is never completed.

C. Adaptation

The efficient usage of energy requires task management and energy management related to each other. Different tasks require different energy capacity. There is a tradeoff between energy capacity constraint tasks, which require a minimum energy level on the capacitor to function continuously, and temporarily constrain tasks which require high energy when demanded. If one of the requirements is satisfied, then the other one will not function fully. Capybara is a hardware/software power system that specifies the energy requirements of the tasks and configures the energy storage mechanism [10]. The hardware mechanism can be used with different capacitors and harvesters.

Different approaches to adaptation is to change the program according to the available energy. As the energy in the environment changes, the energy provided to the energy harvesting device changes. The big question here is how and when to adapt to enable flexible and general adaptation for different applications. Flexibility comes with a tradeoff for the memory space. REHASH [11] introduces a runtime system, framework complementary tool to write intermittent computing applications that downgrades or upgrades the program. It informs the programmer about the impact of the adaptation on the system performance.

IV. User Friendly Implementations

Most of the platforms introduced in the previous chapters implement C/C++ code, which is not a widely used and user-friendly language. To enable the ease of access to intermittent computing and spread use of energy harvesting devices, there have been attempts to adapt intermittent computing to Python. BFree [7] is the first one to introduce an end-to-end system to enable hobbyists and students to write intermittent computing applications. The adaptation of intermittent computing on MicroPython is enabled, and none of the modifications are visible to the user, ensuring a smooth user experience. The system rewrites CircuitPython interpreter to checkpoint and to restore the state. This is different than previous software approaches in the literature. Moreover, the initialization of the system and the peripherals and system libraries are modified to enable fast rebooting. The paper also introduces a custom hardware shield based on Adafruit's CircuitPython device. The implementation is done on ARM Cortex M0 controller, which has better technical specifications than the widely-used microcontroller TI MSP430 FR [9, 10]. Although the system is not the quickest or most efficient compared to task-based approaches, it is user-friendly and opens doors for new areas of implementation of intermittent computing. The design includes a specific hardware, which limits the area of use.

Battery-free MakeCode develops a source code transformation for Microsoft MakeCode IDE and compiler [8]. It supports multiple languages. Because it runs on an online IDE, it requires no installations or custom boards. Code transformation occurs in the middle end and modifies the Intermediate Representation. MakeCode Iceberg is an open-source extension available for everyone.

MakeCode inserts checkpoints by modifying the IR, which brings generalization for the code and enables usage with different target devices [8]. Another approach would be checkpointing during runtime like BFree [7]. However, that limits the system to being target-specific. Checkpointing uses a timer approach and writes the entire data onto non-volatile memory once approximately every 100ms. Although there are some limitations for the libraries that can be used, MakeCode Iceberg introduces a sufficient, novice friendly programing for intermittent computing that can be used with different target devices.

V. Applications

Improvement in intermittent computing platforms opened new application areas for energy harvesting devices such as cellphones [17], machine learning [15], house monitoring [14], video streaming [12], and eye tracking [13].

Machine learning is implemented in every filed of the technology. Intermittent learning can open the door to privacy conscious, secure, autonomous, responsive, adaptive and forever-evolving energy harvesting devices. Intermittent learning proposed by Lee et al. [15] can perform training and inferring while ensuring atomicity, consistency, programmability, timeliness, and energy efficiency on certain classes of machine learning applications. Intermittent learning is performing in an energy scarce environment, and the algorithm should be modified to be able to train the model intermittently. To optimize the learning and inferring, the system decides when to train and when to infer. Data collected from the environment should be carefully evaluated since some of the data might be useless for the algorithm and needs to be eliminated. The system also changes the execution order of the task according to the energy scarcity. Their implementation reaches up to 100% energy efficiency and decreases learning examples by 50% compared to state-of-the-art intermittent computing systems.

One of the most recent and only interactive application for energy harvesting is battery-free Game Boy [16]. It introduces ENGAGE, a real Nintendo Game Boy playing system. The control operation of the system is in real time which is a difficult task because of responsiveness issues in intermittent computing. Game Boy [16] uses solar energy and button-press to power the device. The game is saved periodically by checkpointing the entire system so that the game can continue from where it ended. To decrease the checkpointing time and cost, just the difference between the states is checkpointed to the main memory, similar to DICE [3]. Game Boy can play different type of games. However, precise, time-based games are difficult to play because the game gets interrupted too frequently. If the power is not available, the screen turns off, creating an unpleasant gaming experience.

VI. Conclusion

This paper presented the hardware/software requirements for intermittent computing and reviewed the intermittence-enabling software models and energy harvesting applications. Batteryless devices have a promising future for the sustainable, environmentally friendly development of technology. There is still plenty of research space for new ideas and implementations. The area of energy harvesting is expanding rapidly.

References

[1] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent Computing: Challenges and Opportunities,", 2017, doi: 10.4230/LIPICS.SNAPL.2017.8.J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[2] Ben Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-Running Computation on RFID-Scale Devices. In Proc. 16th Int'l Conf. Archi- tectural Support for Programming Languages and Operating Systems (ASPLOS'11). ACM, Newport Beach, CA, USA, 159–170.

[3] Ahmed, Saad & Bhatti, Naveed Anwar & Alizai, Muhammad Hamad & Siddiqui, Junaid & Mottola, Luca. (2019). Efficient Intermittent Computing with Differential Checkpointing. 10.1145/3316482.3326357.

[4] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’ 15). ACM, New York, NY, USA, 575–585. DOI:https://doi.org/10.1145/2737924. 2737978

[5] Alexei Colin and Brandon Lucia. 2016. 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). ACM, New York, NY, USA, 514 ‐ 530. DOI:https://doi.org/10.1145/ 2983990.2983995

[6] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: intermittent execution without checkpoints. Proc. ACM Program. Lang. 1, OOPSLA, Article 96 (October 2017), 30 pages. https://doi.org/10.1145/3133920

[7] Vito Kortbeek, Abu Bakar, Stefany Cruz, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. 2020. BFree: Enabling Battery-free Sensor Prototyping with Python. Proc. ACM Interact. Mob. Wearable Ubiquitous Technol. 4, 4, Article 135 (December 2020), 39 pages. https://doi.org/10.1145/3432191

[8] Christopher Kraemer, Amy Guo, Saad Ahmed, and Josiah Hester. 2022. Battery-free MakeCode: Accessible Programming for Intermittent Computing. Proc. ACM Interact. Mob. Wearable Ubiquitous Technol. 6, 1, Article 18 (March 2022), 35 pages. https://doi.org/10.1145/3517236

[9] Alanson P Sample, Daniel J Yeager, Pauline S Powledge, Alexander V Mamishev, and Joshua R Smith. 2008. Design of an RFID-based battery-free programmable sensing platform. IEEE Transactions on Instrumentation and Measurement 57, 11 (2008), 2608–2615.

[10] Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices. SIGPLAN Not. 53, 2 (February 2018), 767–781. https://doi.org/10.1145/3296957.3173210

[11] Abu Bakar, Alexander G. Ross, Kasim Sinan Yildirim, and Josiah Hester. 2021. REHASH: A Flexible, Developer Focused, Heuristic Adaptation Platform for Intermittently Powered Computing. Proc. ACM Interact. Mob. Wearable Ubiquitous Technol. 5, 3, Article 87 (Sept 2021), 42 pages. https://doi.org/10.1145/3478077

[12] Saman Naderiparizi, Mehrdad Hessar, Vamsi Talla, Shyamnath Gollakota, and Joshua R. Smith. 2018. Towards Battery-Free HD Video Streaming. In Proc. NSDI (April 9–11). USENIX, Renton, WA, USA, 233–247.

[13] Tianxing Li and Xia Zhou. 2018. Battery-Free Eye Tracker on Glasses. In Proc. MobiCom (October 29 — November 2). ACM, New Delhi, India, 67‐82.

[14] Josiah Hester, Lanny Sitanayah, and Jacob Sorber. 2015. Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors. In Proc. SenSys (Nov. 1‐4). ACM, Seoul, South Korea, 5‐16.

[15] Y Seulki Lee, Bashima Islam, Yubo Luo, and Shahriar Nirjon. 2019. Intermittent Learning: On-Device Machine Learning on Intermittently Powered System. ACM Interact. Mob. Wearable Ubiquitous Technol. 3, 4 (Dec. 2019), 141:1–141:30

[16] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. 2021. Battery-Free Game Boy: Sustainable Interactive Devices. GetMobile: Mobile Comp. and Comm. 25, 2 (June 2021), 22–26. https://doi.org/10.1145/3486880.3486888

[17] Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua R Smith. 2017b. Battery-free Cellphone. ACM Interact. Mob. Wearable Ubiquitous Technol. 1, 2 (June 2017), 25:1–25:19.