Seminar Paper: Debugging Intermittent Systems

Kevin Kollenda
kevin.kollenda@fau.de

ABSTRACT

Intermittent systems can be found in anything from payment cards to home-automation devices. Debugging embedded systems has already proven to be a difficult task, which only gets harder when considering the unreliable and fast-changing energy environments that characterize intermittent systems. Implementing a debugger that can not only handle common operations, such as setting breakpoints or memory manipulation, but also cope with the unique energy requirements requires novel techniques and research. Well-integrated intermittent system debuggers accelerate the development cycle and are invaluable in the creation of error-resilient devices. This paper outlines and discusses approaches and challenges that need to be considered when designing and implementing an intermittent system debugger.

KEYWORDS

debugging, intermittent systems, embedded systems, energy emulation

1 INTRODUCTION

The past few years have seen a steady increase in the number of embedded systems [15]. They can be found in an ever-growing number of devices, ranging from simple consumer electronics to complex industrial appliances. As energy harvesting technologies are gaining traction, there is a growing interest in reducing the amount of batteries required to power such devices and even replace them outright [17].

These developments are partially caused by the severe disadvantages of batteries, such as the high environmental impact of their production and disposal, the limited capacity and the resulting need for periodic replacement [6, 16]. Thus, there is an increasing number of battery-less embedded systems, which harvest their energy from external sources, including solar, thermal, kinetic and radio [8]. These systems are called *intermittent systems*, due to their unique energy characteristics. As they fully rely on external energy sources, they are subject to frequent power failures leading to intermittent execution of their software. While they are often equipped with (super-)capacitors to stabilize the power supply, these are only able to keep the device running for a brief amount of time (i.e. few seconds).

Existing debugging techniques and tools for embedded systems are not equipped to handle the unique challenges of intermittent systems. Typically, these debuggers either assume or provide a continuous power supply to the device under test (DUT) [19], which masks all intermittency related issues outright [2, 3]. Trying to circumvent these issues by keeping the original power supply circuitry in-place, can cause problems during debugging. Tracing calls, like `printf(. . . )` and `assert(. . . )`, skew the energy characteristics of the DUT and change the behaviour encountered during regular execution. Breakpoints pose additional challenges, as they can halt execution indefinitely while the developer inspects the halted program state, draining the available energy. Finding a solution to these problems is crucial, as a proper debugging environment is essential for developing robust and error-resilient software [6].

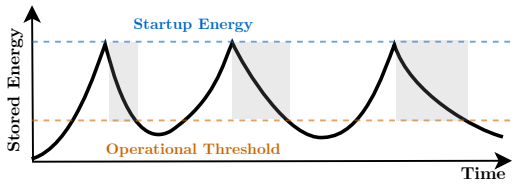

Figure 1: Energy characteristics of intermittent systems. Execution begins after reaching the required start-up energy of the device. While working on computational tasks and interacting with peripheral devices, the energy storage slowly drains. Upon reaching a set operational threshold, the system loses its power. Work can only be done in the short intervals between system start-up and power failure [6].

2 INTERMITTENT SYSTEMS

Intermittent systems are characterized by their lack of batteries and their dependence on energy harvesting. Harvested energy is collected to a (super-)capacitor that powers the device upon reaching a voltage threshold and operating until it is depleted again. Capacitors are only able to store small amounts of energy and external energy sources are often unreliable (e.g. solar panels) leading to frequent power failures. Depending on the type of device, this charge-drain-recharge cycle happens many times per second, triggering frequent restarts of the entire program [6]. Figure 1 displays the energy characteristics typical of intermittent systems. After harvesting enough energy to reach the system's start-up energy, the system will drain the stored energy during its computational tasks. A power failure occurs when energy levels drop below the operational threshold.

There has been extensive research on executing programs reliably on intermittent systems without risking data loss or corruption. Some of these techniques rely on transforming the program into a state-machine, which only executes tasks that can be completed inside the currently available energy budget. Others try to save and restore the volatile state in face of power loss to non-volatile memory (NVM).

Task-Based Programming aims to split up programs into tasks, that are guaranteed to execute completely given a certain energy budget. Tasks can be thought of as atomic units of execution, as they either complete successfully or their effects and results are discarded [21]. Each task must only require as much energy as is available between two power failures, or be split up into subtasks that do. System integrity is ensured, as no task can corrupt the system's state. *Alpaca* [12] is a C extension featuring such a task-based programming model.

Checkpointing relies on saving the system's state to non-volatile memory at certain points during execution. When power becomes available after a power failure, the last checkpoint is restored and program execution resumes. Multiple trade-offs have to be made when deciding on the checkpointing algorithm to use, such as the frequency of checkpoints or how much state is stored. For example, delta compression can lead to significant reductions in the transferred data, but requires additional energy for computing the changes. *Mementos*[18] provides energy aware checkpoints that heuristically decide when to checkpoint at run time.

Non-volatile systems abandon volatile state outright and replace all required components with non-volatile alternatives. Non-volatile random-access memory is already closing the gap to regular random-access memory and there is research exploring the micro-architectural changes required to decrease the volatile state used by processors [10, 11]. Increasing the amount of state kept in non-volatile memory lowers the impact of intermittency significantly. If everything is stored in persistent mediums, power failures are a non-issue making such intermittent systems behave like continuous ones — albeit execution halting indefinitely until the system has harvested enough energy. As the only change required to support debuggers for these kinds of systems is working towards lowering the attachment latency, they will not be mentioned in the following sections.

Restoring program state is not enough for many embedded devices, which often rely on connected hardware and sensors. These input and output (IO) devices are often stateful themselves (such as a display showing a certain image) and might need to be reinitialized following a power loss.

As the energy dependence of intermittent systems plays a major role in the development process, it is important to be able to gain insights of the system's energy usage and storage over time. *Energy traces* are generated by measuring the power circuitry of connected devices and the DUT [7]. Analyzing and replaying these traces can prove very valuable in the reproduction of intermittency related failures.

3 DEBUGGING INTERMITTENT SYSTEMS

Intermittent Systems, with their unique set of behaviours and characteristics, require capable debugging tools that allow developers to fully control and monitor the DUT. The following section will outline errors only encountered during non-continuous execution, describe the challenges associated with debugging such systems and pose a list of requirements a fully-featured intermittent system debugger should fulfill.

3.1 Errors in Intermittent Systems

As the running program can be interrupted at any time due to sudden power loss, it is essential to employ the services of a state management framework as described in section 2. These frameworks must be able to mitigate the issues described in the following to ensure error-free execution.

3.1.1 Volatile State Restoration. Checkpoint-based software relies on restoring the previously saved state to continue execution. Depending on the complexity of the mechanism, this includes everything from processor registers (e.g. program counter, stack pointer, general purpose registers, …) to the entire volatile random-access-memory (RAM) of the chip [18]. Intricate checkpointing frameworks can contain programming errors themselves, as their advanced design must be fully resilient to power failures. Many issues stem from the large amounts of data to be saved to non-volatile memory before depleting the stored energy. Atomicity guarantees must be in place to prevent unfinished checkpoints from being written to persistent storage [5]. Otherwise, a combination of a previous program counter and new register contents will stop the program from resuming correctly [14]. Figure 2 depicts such a partial state restoration, where the processor's state was successfully saved and restored from NVM, but the system's memory did not get stored completely. Thus, the program will continue from the same location with corrupted memory, leading to undefined behaviour.

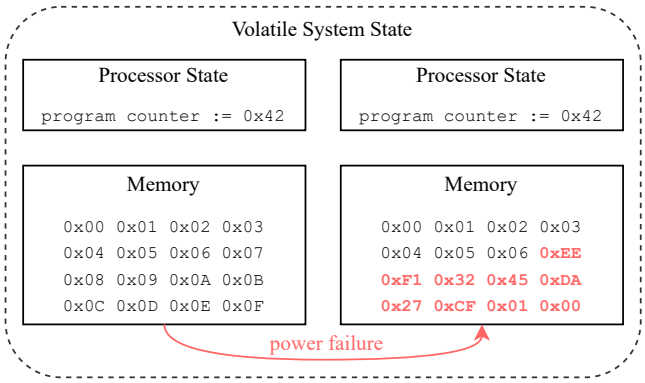

Figure 2: Simplified view on the state found in an intermittent system. If the state restoration algorithm is erroneous it can corrupt the memory. In this example, the processor's program counter and parts of the system memory were restored correctly, but left the remaining memory in an invalid shape. [5]

Common counter measures, such as multiple buffering, can still be susceptible to these issues. They need to ensure that updating the flag indicating which buffer to use happens without interruptions, otherwise the problem will persist in other forms.

3.1.2 Periphery State Management. Many embedded systems depend upon external devices to function, like temperature sensors, displays or real-time clocks (RTC). The state of these peripheral devices must be considered as well, especially if they are powered separately from the main circuit or hold persistent state internally. Sensors often need to be initialized or calibrated before they can be used properly, which needs to be accounted for while designing the persistency framework — be it task- or checkpoint based [13, 21].

Figure 3 showcases an intermittency problem due to missing periphery state restoration. After a power failure occurs in line 5, the checkpointing system will restore the intermittent systems volatile state as described in subsection 3.1.1. This includes the previously read `data` sent in line 6. However, the next time the while loop's condition is evaluated in line 3, `SensorRead(...)` will try to read from an uninitialized and uncalibrated sensor. Depending on the hard- and software implementation, this can lead to the function call blocking indefinitely or the transmission of invalid data.

```
1       sensor = InitializeSensor();
2       Calibrate(sensor);
3       while(data = Read(sensor)) {
4           Checkpoint();
5           // <Power failure occurs>
6           Transmit(data);
7       }
```

Figure 3: Code snippet showcasing a read-transmit loop, commonly found in sensor-based intermittent devices. When a power failure occurs in line 5, the checkpointing framework will restore the systems memory. However, the sensor is not re-initialized and re-calibrated, leading to undefined behaviour upon reaching line 3.

Restoring from power loss only works properly, when considering the interactions between the system and its attached peripheral devices. Loading the state of one without the other will inevitably lead to issues, as their progress diverges [3].

3.2 Challenges in Debugging Intermittent Systems

Common debugging approaches for embedded devices, like toggling a light-emitting diode when reaching certain lines of code, are not applicable due to intermittent systems' high energy sensitivity. While the usage of oscilloscopes can provide a deep look into the DUT's energy behaviour and its attached devices, the prohibitive costs and inability to relate energy consumption to the running code make them unsuitable for most applications [2]. Inserting tracing calls like `printf` or assertions to verify that the program is executing correctly does not only alter the energy characteristics of the device, but may also need to allocate non-volatile memory to save the results. As there is only limited NVM available, this technique can only be used after careful consideration. It may be tempting to directly stream tracing output to connected devices, but as these IO devices need to be powered and clocked themselves this can further skew program behaviour as this consumes additional energy.

Software breakpoints are typically implemented as regular function calls, which might impact checkpointing algorithms. Often times, these algorithms try to heuristically determine where to put checkpoints by analyzing control flow and data dependencies [18]. Function calls pose a good target, leading to different state restoration behaviour when software breakpoints are used. Figure 4 showcases how volatile state can get saved pre-maturely when implicit checkpoints are inserted by software breakpoints. In the example, a `for` loop repeats `N` times and stores the newly computed total to non-volatile memory. As the counter variable `i` resides in volatile memory, it does not survive a power loss. Depending on `N` and the speed of the deployed sensor, the loop could take a long time to complete. Thus, there is a high probability of a power failure occurring during iteration. If `i` is not restored, the loop will be executed `N` times again instead of the remaining `N - i` iterations, which skews the total value. As `DBG_Breakpoint()` causes a checkpoint

Snippet (a)

```
1 Checkpoint();
2 total = NVM_Load();
3 for i < N {
4
5
6   total += Sense();
7   NVM_Store(total);
8 }
9 // i gets saved
10 Checkpoint();
```

Snippet (b)

```
1 Checkpoint();
2 total = NVM_Load();
3 for i < N {
4   // i gets saved
5   DBG_Breakpoint();
6   total += Sense();
7   NVM_Store(total);
8 }
9
10 Checkpoint();
```

Figure 4: Code listing showing how software breakpoints can hide erroneous checkpoint placement. As the loop's counter `i` resides in volatile memory, it is only stored to NVM upon reaching a checkpoint. This leads to the loop starting all over again, when a power failure occurs before the last checkpoint. `DBG_Breakpoint()` introduces an implicit checkpoint, as function calls are used as a heuristic to determine checkpoint placement. Here, this leads to `i` being saved to NVM, eliminating the issue found in the original code.

to be generated inside the loop, `i` will be saved when running in debug mode — masking the original error.

Intermittent Systems debuggers must not only implement standard debugging procedures that integrate with the aforementioned state management, but also be able to control the power supply of the DUT. Only the ability to provide the DUT with any kind of energy input, be it continuously powered or replaying a pre-recorded energy trace, enables full reproduction of previously encountered issues [3]. *Ekho* is a stand-alone energy emulator capable of recording and replaying detailed energy harvesting traces. Mocking the device's power supply is handled by an *energy emulator*, hooking into the DUTs circuitry. Minimizing interference is of utmost importance, as not to change the behaviour of the intermittent system.

System emulators can help in debugging embedded devices, as they are able to execute the given software directly on the host machine, increasing the development speed significantly. However, even with fine-grained instruction level emulation, it is difficult to replicate the timing behaviour of real chips accurately. As intermittent systems can fail between two clock cycles, the micro-architecture of the chip must be implemented precisely. Additionally, the energy impact of each instruction must be simulated to track energy usage during execution. *SIREN* [4] is an intermittent system emulator, which relies on real-world energy traces recorded by *Ekho*. Many emulators focus solely on CPU emulation and simply disregard or completely mock external devices, making them unsuitable for debugging complex circuits. Even when instructions are timed clock-accurately and peripheral devices are fully emulated, every produced chip has subtle variations in its power consumption and startup behaviour. Thus, emulators can support the development process but not replace inspecting the real hardware directly [3].

3.3 Required Features

Tight integration of the debugger and the energy emulator enable energy-neutrality during debugging. As developers want to debug the device in its whole while considering the behaviour caused by intermittent execution, an intermittent system debugger should implement certain features to accommodate these requirements:

- **Energy emulation** allows the developer to replay recorded energy traces or supply synthetic power to the DUT. Furthermore, monitoring of the system's power consumption is required to closely observe the impact of running code on the given energy budget.
- **Energy-neutral debugging** minimizes the impact of breakpoints and tracing calls. Single-stepping can only work when energy neutrality is given, as otherwise there may simply not be enough energy stored to support time expensive debugging. *Energy-guards* enable the exclusion of code from energy emulation, thus masking the impact on the device's stored power [2].

These features must not significantly introduce computational overhead or change the program's behaviour, as to provide a realistic execution environment.

4 IMPLEMENTATIONS OF INTERMITTENT SYSTEMS DEBUGGERS

An intermittent system debugger must satisfy these requirements while minimizing the impact on the DUT. Multiple approaches on how to design a debugger for intermittent systems will be discussed.

4.1 Energy Management

Simulating the device's energy can either be performed by replacing the original power supply outright or by hooking into the original circuit. If the power supply is fully replaced by the energy emulator, it can provide the system with virtually any type of energy input [3]. Real-world testing can be done by replaying a pre-recorded energy trace containing the measured capacitor voltage and the net power consumption of the device. Full energy control also allows to simulate any kind of capacitor and energy harvesting device, massively simplifying the part selection process. Stress- and limit-testing can be performed by providing configurable synthetic signals, e.g. configurable sawtooth and square waves, to the DUT. Partial energy management leaves the DUTs power supply in place and hooks to the existing power lines [2]. Keeping energy interference to a minimum is essential to enable a realistic test environment. Figure 5 showcases how full energy management takes over the entire power supply of the device, allowing for the capabilities mentioned above.

Energy emulation can either be performed *passively*, where the system powers itself using energy harvesting, or *actively* with the debugger providing the system with power [2, 3]. Active mode is helpful in early development stages, as it enables the developer to fully manipulate the energy input characteristics of the DUT. However, it is possible to closely mimic an energy environment with passive mode that closely matches the one targeted for final deployment. Measurements acquired during passive mode can also provide deep insights into the system's behaviour and be used to determine statistics, such as average power usage and the amount of energy harvested during regular operation.

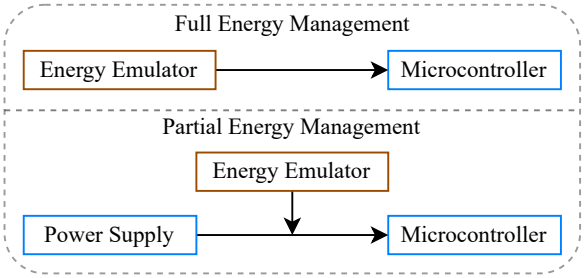

Figure 5: Energy management strategies typically employed in intermittent systems debugging. Replacing the power supply entirely allows the attached energy emulator to fully control the energy input of the device. A less invasive method is to hook into the existing power lines of the DUT and manipulate the existing energy levels.

4.2 Debugger Design

Many considerations and trade-offs need to be made when creating an intermittent system debugger. This section aims to shed some light on the advantages and disadvantages of choosing to support one capability or design over another.

4.2.1 Software-Based Debuggers. Debugging features, like setting break- or watchpoints, can be implemented using a software library. This requires the application developer to set breakpoints or declare energy-guarded sections in the existing code. When the MCU encounters such a call during execution, the debugger springs into action and handles the request [2].

As discussed in section 3.2, software-based debuggers can mask existing errors in the code. Every additional line of code inserted for debugging can cause the compiler to optimize differently [3]. Function calls to different modules, i.e. the debugger's software library, completely prevents function inlining in languages like C. Loop unrolling also gets more difficult the more code resides in a loop body. Even though programs are often compiled without optimizations during debugging, it does not help towards the final stages of the development cycle where programs are run in release mode.

Software-based debugging abstracts away the underlying hardware, which allows it to be used when developing for chips without integrated debugging hardware. However, this comes at the cost of having to recompile the entire program when modifying breakpoints or assertions, slowing down the development cycle. Additionally, hardware debuggers can be prohibitively expensive making software debuggers attractive for individuals.

4.2.2 Hardware-Based Debuggers. Many microcontrollers (MCU) have integrated debugging circuitry, which can be accessed by hardware debuggers attached to the MCU. Common debugging features, like breakpoints or memory manipulation, are performed by the MCU itself. While this alleviates the need to instrument the running code to make use of the debugger, e.g. inserting `DBG_Breakpoint()`

calls, it is not without issues. As intermittent systems suffer power failures, all volatile MCU state is lost — including essential debugging information. If it is not restored fast enough, the program will resume execution disregarding previously set breakpoints. Most MCUs cannot alter their own debug registers due to security concerns, worsening the issue. Even when the debugger keeps track of all necessary state and tries to restore it as fast as possible, breakpoints occurring at the start of execution might still be missed [3]. Thus, hardware debuggers need to minimize the time required for (re-)connection to alleviate these problems. Fully preventing this issue is only possible by requiring the debugger to be fully attached at program start, i.e. calling `DBG_Attach()` before anything else, increasing start-up latency. Furthermore, the on-chip debugging support requires energy, which needs to be accounted for in the energy emulator.

4.2.3 Development Interface. Interactions with debuggers should be as smooth as possible to enable rapid development cycles. Building and expanding upon existing debuggers like GDB, easens the migration process and offers integration with existing tools and plugins. As software debuggers provide a completely new library interface, it is more difficult to incorporate them to existing software and developer teams. Apart from offering an interface for coding-related debugging tasks, there also needs to be a well-designed way to communicate with the energy emulator. Graphical user interfaces are incredibly useful for visualizing and manipulating energy traces and monitoring the device's energy level over time.

4.3 Energy-Neutral Debugging

One of the major challenges encountered while fixing issues in intermittent systems, is the increased power consumption caused by debugging code that traces the control flow or performs constraint checks on large data structures. Even worse, conventional embedded system debuggers are not resilient to power failures of the tested device and further strain the limited energy budget when handling breakpoint invocations or slowly single-stepping through code. Thus, a debugger designed for intermittent system has to offer energy-neutrality during those moments.

4.3.1 Breakpoints. As the control flow of intermittent systems is not only dependent on conventional conditions found in program code, the current energy level also plays a significant role in the behaviour of the system. Apart from code breakpoints that activate when reaching a certain line of code, *energy breakpoints* trigger when the amount of stored energy is lower or equal than a set value [2]. Combining these already useful primitives enables *combined breakpoints*, which test the energy level at a line of code. These combined breakpoints can reveal unexpected energy loss caused by unforeseen code paths and trigger exactly at the right time when the device's energy deviates from the expected amount. Tight integration with the energy management unit, allows the debugger to record and restore the energy level encountered when pausing execution at a breakpoint. From the perspective of the developer, this allows to debug the device like an ordinary embedded system, while the debugger powers the DUT in the background [2, 3].

4.3.2 Energy-Guards. During development, it might be helpful to include constraint checking for important data structures across the entire program. This can catch issues introduced by faulty memory restoration, as shown in section 3.1.1, and common coding errors alike. However, iterating through all elements of a container, such as lists or trees, and verifying that pointers are set correctly, or that certain checksums hold up is computationally and energetically expensive [2]. In intermittent systems this leads to the depletion of stored energy before reaching the erroneous parts of code, preventing extensive constraint checks outright.

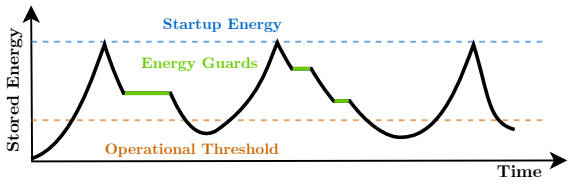

Figure 6: Energy-guards bridge the time between regular program execution and debugging tasks. Green sections mark the energy emulator springing into action and supplying the DUT with power, hiding the effects of debugging.

Intermittent system debuggers provide *energy-guards*. These energy-guards enable intervals of code to run with no impact on the device's energy storage, by employing a similar energy restoration mechanism to breakpoints [2, 3]. Code inside a guarded section does not affect the energy level of the system, allowing not only for expensive constraint checks, but also the verification and reporting of data produced by external devices. Even ad hoc debugging mechanisms, like toggling LEDs when certain conditions are met, can be used inside an energy-guard. This behaviour can be observed in figure 6, where the green sections indicate guarded code. Similar energy traces are the result of other debugging actions, such as breakpoints and manual single-stepping. Only the tight integration of an energy emulator and a debugger enables the full functionalities required to properly debug intermittent systems.

Energy-guards allow the integration of intermittency-unaware library code into the rest of the program, as uninterrupted execution is guaranteed. Developers migrating embedded device code to intermittent systems get the ability to start with a fully energy-guarded program and slowly increase the power failure resiliency by shrinking the guarded section and rewriting the affected code.

`DBG_printf()` and `DBG_assert()` functions further improve the developer experience by providing energy-neutrality for common debugging tools. Failing an asserted condition behaves similarly to entering an energy-guarded section or a breakpoint, until the programmer resumes execution [3].

4.4 Automated Testing

Provided the aforementioned features and capabilities, it can be feasible to perform automated testing of intermittent systems. Debuggers can either extend existing debuggers, like GDB, or offer a scriptable interface to allow other software to interact with the DUT. When parameterized with the functions used for checkpointing and the affected memory ranges, it is possible to verify that the memory was not corrupted during state restoration. While this already offers a significantly improved debugging experience for

	EDB	DIPS
Debugger Design	Software	Hardware
Energy Management	Partial	Full
GDB-Based	No	Yes
Energy-neutral Debugging	Yes	Yes
Breakpoints	Software	Software & Hardware
Automated Testing	No	Yes
Single Stepping	No	Yes
Supported Architectures	MSP430	ARM

Table 1: Design and capability comparison of EDB [2] and DIPS [3].

testing a single device, it can also be used during the development of new checkpointing frameworks. Similar instrumentation can be performed for task-based frameworks, where energy breakpoints can be set inside running tasks to ensure that sufficient energy is available to finish the current task [3]. If this invariant does not hold true, the atomicity guarantees of the framework were violated and need to be checked. Hardware debuggers can observe the full memory range available, which additionally allows to verify that peripheral devices' state was successfully restored.

5 EXISTING DEBUGGERS

The rising popularity of intermittent systems has lead to the first generation of energy-aware debuggers.

To the best of my knowledge, the *energy-interference-free debugger* (EDB) [2] was the first publicly available debugger that was designed from the ground up for intermittent systems. It features a software-based debugger with partial energy management capabilities, allowing to measure and manipulate the energy levels of the DUT on the fly. The newly introduced concept of energy-neutral debugging, showed its worth as multiple bugs were found in existing code using EDB. Minimizing back-feeding from EDB to the tested device needs to be considered, as EDB hooks into the existing power rails of the device. Cumulating the current generated by EDB across all attached devices and communication lanes, yields a total worst-case current flow below $1\mu A$. Even though intermittent systems are known for their low power usage, this is still a negligible amount. Additionally, there are little changes when comparing the energy level at the start of an energy-guard and after resuming execution. The mean difference of stored energy is around $4 - 5\%$, being suitable for most use-cases [2].

DIPS [3], the *debugger for intermittently-powered systems*, follows in EDB's footsteps and expands upon its features. As a hardware-based debugger, it is not affected by the issues regarding masked errors described in section 3.2 and missed optimizations as discussed in section 4.2.1. DIPS accomplishes this by utilizing the MCU's debugging circuitry instead of relying on a software-library for common debugging tasks. Furthermore, it enables automatic testing of the DUT, which proved itself to be a major factor in the development of error-resilient systems. While support for the MSP430 architecture, which can be found in many intermittent systems due to its support for non-volatile FRAM [9], is still pending DIPS has already proven useful as shown in numerous case studies. DIPS's energy emulator can replay energy traces to an accuracy of one millisecond and sample the device's voltage and current level at a frequency of $50kHz$. As hardware debuggers need to reconnect quickly to the MCU's inbuilt hardware, it is important to offer low attachment latencies. DIPS takes less than $100ms$ to reconnect on most devices, with the initial connection delay staying under $400ms$ across all devices. If breakpoints could occur during these $100ms$, one can instrument the tested program to call `DBG_Attach()` on start-up, which will ensure that the debugging environment is fully setup before running user code [3].

6 RELATED WORK AND FURTHER RESEARCH

Foundational work on testing continuously powered embedded systems was summarized in [1]. *Mementos* [18] introduced checkpoint based state handling, bridging the gap between software written for battery powered and intermittent systems. *Alpaca* [12] foregoes the potential issues caused by checkpointing and uses energy-aware task scheduling to prevent inconsistencies across power loss. As non-volatile memory performance approaches that of conventional memory [20], the idea of running the entire system from NVM and thereby minimizing volatile state is gaining traction. While replacing conventional random-access memory is already possible, micro-architectural changes are required to enable intermittency-safe processors. [10] and [11] shed some light on how such intermittent systems could be implemented. Future work can leverage the findings presented by *EDB* [2] and *DIPS* [3] to improve the debugging landscape. As the technical foundations are already available, the focus can shift to the incorporation of additional hardware architectures and improving the energy emulation.

7 CONCLUSION

As intermittent systems gain popularity, the need for feature-rich debuggers grows larger. Only tight integration between an advanced energy emulator and complex debuggers can fulfill the unique requirements posed by intermittent systems. Automated testing builds upon the functionalities provided by debuggers and allows the integration of existing fuzzing frameworks, which have proven useful many times. Error-resiliency needs to be one of the foundational priorities when designing and building intermittent systems, especially when used in health or security-related contexts. *EDB* and *DIPS* showcase how energy-aware debuggers can significantly improve the development process and are essential in reducing issues caused by the special energy characteristics of such systems. Future work can expand upon these ideas, which allows intermittent systems to be used in even more places and fields as they are already established in.

REFERENCES

[1] Abhijeet Banerjee, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2016. Chapter Three - On Testing Embedded Software. Advances in Computers, Vol. 101. Elsevier, 121–153. https://doi.org/10.1016/bs.adcom.2015.11.005

[2] Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. 2016. An Energy-Interference-Free Hardware-Software Debugger for Intermittent Energy-Harvesting Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 577–589. https://doi.org/10.1145/2872362.2872409

[3] Jasper de Winkel, Tom Hoefnagel, Boris Blokland, and Przemysław Pawełczak. 2023. DIPS: Debug Intermittently-Powered Systems Like Any Embedded System. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems* (Boston, Massachusetts) *(SenSys '22)*. Association for Computing Machinery, New York, NY, USA, 222–235. https://doi.org/10.1145/3560905.3568543

[4] Matthew Furlong, Josiah Hester, Kevin Storer, and Jacob Sorber. 2016. Realistic Simulation for Tiny Batteryless Sensors. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems* (<conf-loc>, <city>Stanford</city>, <state>CA</state>, <country>USA</country>, </conf-loc>) *(ENSsys '16)*. Association for Computing Machinery, New York, NY, USA, 23–26. https://doi.org/10.1145/2996884.2996889

[5] Michele Grisafi, Mahmoud Ammar, Kasim Sinan Yildirim, and Bruno Crispo. 2022. MPI: Memory Protection for Intermittent Computing. *IEEE Transactions on Information Forensics and Security* 17 (2022), 3597–3610. https://doi.org/10.1109/TIFS.2022.3210866

[6] Josiah Hester and Jacob Sorber. 2017. The Future of Sensing is Batteryless, Intermittent, and Awesome. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems* (Delft, Netherlands) *(SenSys '17)*. Association for Computing Machinery, New York, NY, USA, Article 21, 6 pages. https://doi.org/10.1145/3131672.3131699

[7] Bashima Islam and Shahriar Nirjon. 2020. Scheduling Computational and Energy Harvesting Tasks in Deadline-Aware Intermittent Systems. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 95–109. https://doi.org/10.1109/RTAS48715.2020.00-14

[8] Meng-Lin Ku, Wei Li, Yan Chen, and K. J. Ray Liu. 2016. Advances in Energy Harvesting Communications: Past, Present, and Future Challenges. *IEEE Communications Surveys & Tutorials* 18, 2 (2016), 1384–1412. https://doi.org/10.1109/COMST.2015.2497324

[9] Brock J. LaMeres. 2023. *The MSP430*. Springer International Publishing, Cham. 135–152 pages. https://doi.org/10.1007/978-3-031-20888-1_4

[10] Yongpan Liu, Zewei Li, Hehe Li, Yiqun Wang, Xueqing Li, Kaisheng Ma, Shuangchen Li, Meng-Fan Chang, Sampson John, Yuan Xie, Jiwu Shu, and Huazhong Yang. 2015. Ambient energy harvesting nonvolatile processors: From circuit to system. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2744769.2747910

[11] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015. Architecture exploration for ambient energy harvesting nonvolatile processors. *2015 IEEE 21st International Symposium on High Performance Computer Architecture, HPCA 2015*. https://doi.org/10.1109/HPCA.2015.7056060

[12] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution without Checkpoints. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 96 (oct 2017), 30 pages. https://doi.org/10.1145/3133920

[13] Kiwan Maeng and Brandon Lucia. 2019. Supporting Peripherals in Intermittent Systems with Just-in-Time Checkpoints. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1101–1116. https://doi.org/10.1145/3314221.3314613

[14] Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. 2021. Discovering the Hidden Anomalies of Intermittent Computing. In *Proceedings of the 18th ACM International Conference on Embedded Wireless Systems and Networks (EWSN), Delft (The Netherlands), February 2021*. :.

[15] Peter Marwedel. 2021. *Embedded system design: embedded systems foundations of cyber-physical systems, and the internet of things*. Springer Nature.

[16] Loreto Mateu and Francesc Moll. 2005. Review of energy harvesting techniques and applications for microelectronics. In *VLSI Circuits and Systems II*, Jose Fco. Lopez, Francisco V. Fernandez, Jose Maria Lopez-Villegas, and Jose M. de la Rosa (Eds.), Vol. 5837. International Society for Optics and Photonics, SPIE, 359 – 373. https://doi.org/10.1117/12.613046

[17] J W Matiko, N J Grabham, S P Beeby, and M J Tudor. 2013. Review of the application of energy harvesting in buildings. *Measurement Science and Technology* 25, 1 (nov 2013), 012002. https://doi.org/10.1088/0957-0233/25/1/012002

[18] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. 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* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 159–170. https://doi.org/10.1145/1950365.1950386

[19] Segger. 2023. J-Link EDU Debug Probe. Retrieved December 20, 2023 from https://www.segger.com/products/debug-probes/j-link/models/j-link-edu

[20] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 496–508.

[21] Eren Yildiz, Saad Ahmed, Bashima Islam, Josiah Hester, and Kasim Sinan Yildirim. 2023. Efficient and Safe I/O Operations for Intermittent Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 63–78. https://doi.org/10.1145/3552326.3587435

Received 17. January 2024