# Neverlast: An NVM-centric Operating System for Persistent Edge Systems


Christian Eichler
christian.eichler@rub.de
Ruhr University Bochum (RUB)
Germany

Henriette Hofmeier
henriette.hofmeier@rub.de
Ruhr University Bochum (RUB)
Germany

Stefan Reif
reif@cs.fau.de
Friedrich-Alexander University Erlangen-Nürnberg (FAU)
Germany

Timo Hönig
timo.hoenig@rub.de
Ruhr University Bochum (RUB)
Germany

Jörg Nolte
joerg.nolte@b-tu.de
Brandenburg University of Technology (BTU) Cottbus-Senftenberg
Germany

Wolfgang Schröder-Preikschat
wosch@cs.fau.de
Friedrich-Alexander University Erlangen-Nürnberg (FAU)
Germany



## ABSTRACT

The development of novel non-volatile memories (NVM) with low latencies presents a turning point in the design of computational systems, especially the design of efficient intermittently-powered edge computing systems. Persistent edge computing systems operate under the imminent risk of high-frequent power-supply failures and therefore require adjustments to all parts of the system: The operating system must persist the whole system state prior to power failures to guarantee consistent progress.

In this paper, we present Neverlast, a comprehensive approach focusing on the persistence of computational progress (i.e., register and memory contents) and the persistence of hardware configuration (e.g., configuration of external sensors). To ensure persistence, Neverlast implements an interrupt-driven power-failure manager that preserves the system's state on a fine-grained instruction level with minimal overhead. For peripheral devices, our configurable device and energy manager tracks and replays operations to ensure that devices restart in the correct state. Following this approach, persistence is provided as a service to the application, abstracting from power management and state-preserving techniques.




*APSys '21, August 24–25, 2021, Hong Kong, China*



ACM ISBN 978-1-4503-8698-2/21/08.
https://doi.org/10.1145/3476886.3477513

**ACM Reference Format:**
Christian Eichler, Henriette Hofmeier, Stefan Reif, Timo Hönig, Jörg Nolte, and Wolfgang Schröder-Preikschat. 2021. Neverlast: An NVM-centric Operating System for Persistent Edge Systems. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '21), August 24–25, 2021, Hong Kong, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3476886.3477513


## 1 INTRODUCTION

Edge systems [9] and smart dust [15] operate under hard environmental conditions where power-supply failures are the norm rather than the exception. As no reliable power-supply infrastructure is available, computing nodes apply energy-harvesting techniques and operate *intermittently* [2, 12, 13, 16, 17, 19]. The recent development of fast and energy-efficient non-volatile memory hardware allows edge systems to preserve their internal states across power outages, enabling entirely novel application scenarios as the entire internal state can be made persistent with only minimal overhead.

Guaranteeing *whole-system persistence* in edge scenarios where power-supply failures are the norm, rather than an exception, requires adjustments to all parts of the system—including both the hardware and the software [1]. The system must be appropriately prepared for spontaneous power loss, yet the mechanism to preserve progress itself induces a significant overhead, as it has to guarantee that the system's state is recoverable. Related work has proposed a plethora of mechanisms to ensure that the software state is recoverable [2, 5, 12, 13, 16, 17, 19, 28]. Edge systems, however, comprise more than just a processor and memory—they make use of sensors and actuators and form ad-hoc communication networks. In this paper, we, therefore, integrate *peripheral devices* into persistent operating systems. In particular, we

extend Neverlast, a persistent operating system, for device support. Devices pose two challenges for persistent systems. First, they draw a significant amount of power, which requires consideration in the system-level power management. Second, their internal state is *volatile* and therefore needs persistence mechanisms.
This paper makes the following contributions:
1) The adaptation and extension of the Neverlast OS to a more capable hardware platform, the MSP430FR5994.
2) The design and implementation of a whole-system–state-retentive platform that includes the processor as well as peripheral devices.
3) An extensive evaluation demonstrating that the power draw of external devices affects the safety and progress granularity of persistent edge systems and illustrates the need for system-level power management.

The rest of the paper is structured as follows. Section 2 briefly introduces approaches for persistent systems in general, and Section 3 summarizes our non-volatile system, Neverlast. Design considerations of our support for peripheral devices are presented in Section 4. Section 5 evaluates our prototypical implementation of Neverlast. The final Section 6 presents future work and concludes.

## 2 BACKGROUND AND RELATED WORK

NVM technologies enable systems that preserve their computation progress at fine granularity. Therefore, all store operations to non-volatile RAM are persistent [1]. This property makes them particularly suitable for intermittently-powered systems where power failures occur frequently. However, the combination of volatile and non-volatile memory is considered a "broken time machine" [21] because it is possible that only a part of data is persistent, while other parts of the data reside in volatile memory (e.g., registers or caches) only, which may lead to an inconsistent system state. In consequence, a system with non-volatile storage must consider which states are *recoverable* to ensure that, when resuming execution, each observable state is logically consistent, despite the temporary power loss. Ensuring recoverability, however, causes a trade-off because the energy spent on data-consistency measures reduces the energy available for application progress. In particular, in energy-harvesting scenarios where power failures occur frequently, the system has to ensure that the granularity of recoverable states is dense enough to enable progress with just a tiny amount of available energy. Energy-efficient data-consistency measures are therefore essential in intermittently-powered systems.

The state-of-the-art techniques in programming intermittently-powered systems ensuring data consistency can be broadly classified into one of two categories. First, specific programming models separate computation from non-volatile storage [8, 17]. Similarly, dedicated programming languages, such as Mayfly [13], abstract away from the persistence of storage. Second, transaction-based systems [5, 14, 18, 19, 28] use compiler extensions and runtime support systems that automatically insert checkpoints at suitable locations to persist volatile progress for the processor or peripheral devices. This approach, however, does not consider the actual battery charge and, thus, induces significant overhead in time and energy due to unnecessary checkpoint executions. To account for this problem, approaches with *conditional* checkpoints have been introduced, whereby the system polls the energy-supply state at high frequency and checkpoints are only executed in case of low power supply. The frequent battery-check operations, however, lead to considerable time and energy overhead.

Our approach, in comparison, migrates the decision when to create a checkpoint to the operating-system level. The operating system passively monitors the power-supply state and uses an *interrupt*-based mechanism to handle the threat of power loss. Thus, our design decouples the power-monitoring functionality from the application-checkpoint logic. Similar approaches utilizing interrupts on power loss are described by Berthou et al. [4] and by Narayanan et al. [20], though the latter's system targets server-size computers with completely different performance and power characteristics and much less frequent power failures. Berthou et al. [4], in contrast to our system, designed the *interrupt*-based checkpoint mechanism for a system with volatile main memory.

Persisting not only the system's computational progress but also the state of peripheral devices poses several challenges to the design of system software. Our system proposes to utilize the operating system's syscall interface as a basis for persistent devices. With the help of additional system calls that mark state-relevant sequences, our system keeps a log of calls that lead to the current device state. This log can be replayed once the system resumes after a power outage. In contrast, Arreola et al. [22] suggest keeping a history of configuration instruction on the level of the serial communication protocol. Similarly, FSL [24] traces the device initialization of a cold-boot process once and later replays device-register write operations to reduce system warm-reboot times. Branco et al. [7] also propose a log of state-relevant instructions, though their approach introduces an additional layer between the application and device drivers. Which instructions are logged is determined via device-specific state machines that are provided by driver developers. A different approach is presented by [3], who utilize their *interrupt*-based checkpoint mechanism for persisting external devices. Therefore, devices keep track of their state in specific data structures that can be saved to NVM in case of low power.

# 3 NEVERLAST

In this Section, we give a short insight into the previously published version of Neverlast [10], in particular the core idea to ensuring non-volatility, including the power-failure detection and the basic operating-system services.

Neverlast implicitly provides persistence such that all computational progress is preserved across power losses by default. Instead of transforming application code into transaction patterns to maintain data consistency across power losses, Neverlast provides a virtually persistent processor that keeps its state with the granularity of progress being a single CPU instruction, rather than "tasks" [8, 18] or application-level transaction checkpoints. In doing so, Neverlast provides persistence for the applications along with all operating-system–level resources the applications use.

From the application perspective, this mechanism is equivalent to an interrupt that suspends the execution temporarily (i.e., while insufficient energy is available) and transparently continues execution later.

*Power-failure Event Handling.* Neverlast relies on a hardware component that is commonly present on MCUs used for embedded devices: An on-chip analog voltage comparator, named *Comparator_D* on the MSP430FR5739, the MCU Neverlast was initially designed for. The analog voltage comparator is used to monitor the system's supply voltage by comparing its input channels against a built-in reference voltage and issues an interrupt once the input-voltage level drops below the configured level. A resistor-based voltage divider is employed to achieve interrupts at voltages above 2.5 V, such as the 2.67 V used by the original Neverlast.

Once the supply voltage falls below a particular threshold value, the voltage comparator triggers the interrupt, the corresponding interrupt service routine pushes the processor registers to the stack and subsequently writes the current stack pointer to a non-volatile memory area. The time and energy required to execute the power-failure event handler depend on the hardware, particularly on the amount of volatile state that requires a write-back operation. As the last step, the just stored system state is marked as valid by writing an additional bit to detect aborted executions of the ISR, for instance, caused by premature power outages. This valid bit is checked prior to restoring the system state to prevent the loading of inconsistent states.

For this paper, we replace Neverlast's power-failure ISR with a more sophisticated component handling power failures and external devices: The device and energy manager.

*Fine-grained Data Persistence.* To maximize the forward progress of processes running on Neverlast, the amount of data moved during backup- and restore-routines is kept to a minimum as each data-copy operation requires time, and if power-supply failures occur often, the copy-related overhead harms the application progress. Therefore, Neverlast keeps most data permanently in the FRAM, including the stacks for all processes, leaving the processor registers as the only system-relevant data in volatile memory[1]. Since our target embedded platform has no volatile caches, only the CPU registers need to be saved on power loss.

In conjunction with the power-failure detection described above, this approach allows the system to restore at the exact instruction on which it was interrupted while keeping the backup routine minimal, only storing general-purpose registers and the stack pointer along with the state's valid flag.

In this paper, we extend Neverlast's approach to data persistence by persisting device configurations.

*Process Management.* Neverlast provides, in addition to the detection of imminent power failures, basic operating-system functionalities such as process management, process switching via cooperative scheduling, and synchronization of shared resources using semaphores. Our design takes advantage of the already available operating-system functionalities, as presented in the following Section 4.

# 4 DESIGN

This Section presents Neverlast's design concepts for persisting peripheral devices. In general, persisting devices is highly dependent on the device's behavior and thus requires the interplay of all parts of the system (i.e., applications, operating system, drivers, and if applicable remote communication partners). Neverlast's design addresses the persistence of the devices' internal state to allow the system to resume its execution after a power failure occurred, without the need for the application to handle possible device initialization routines and reconfigurations. To ensure this functionality, Neverlast poses some requirements on the devices to be persisted, as presented in the following. The subsequent sections detail the syscall interface, which allows persisting devices, and Neverlast's device and energy manager, which powers and manages accesses to external devices.

*System Model.* Devices to be persisted, by design, must satisfy the following requirements: The devices exhibit a well-defined initial state from which the target state can be restored by log replaying. Further, the devices must not accumulate inaccessible internal state that cannot be recovered deterministically. Most peripheral devices typically used in edge systems, such as sensors and wireless transceivers, satisfy these requirements.

[1] On the MSP430FR5739, only the FRAM controller contains an internal cache, but it guarantees write-back on power failure.

## 4.1 Syscall Interface

First, we introduce the operating-system interface that lays the foundation for saving and restoring the external devices' states. We argue that the operating system's syscall interface[2] is a balanced tradeoff between overhead and programmatic effort, even though it requires the drivers themselves to adhere to some form of standardized interface (as it is, for instance, common in the Linux kernel[26]).

Neverlast's fundamental idea is introducing two additional system calls – `replay_start` and `replay_end` – that enclose the operations to be replayed when recovering from a power outage. `replay_start` takes, in addition to the device's file descriptor, an id indicating whether and in which order the enclosed system calls should be replayed. These temporary replay blocks are used to guarantee proper continuation in case a power-failure interrupt occurs amid a sequence of associated syscalls (e.g., a write-read sequence).

Calls to the syscalls `read`, `write`, and `ioctl` (if enclosed by `replay_start`/`replay_end`) as well as the corresponding parameters and the enclosing block id are written to the log stored in the non-volatile memory. The log is implemented as a singly linked list of unions of different struct types, one type of struct for each system call's different set of parameters. To allow for fast list operations, we store a pointer to the head and tail of each list for every replay id and every device. The list's nodes are stored in a statically allocated array; unused nodes are stored in a dedicated free list, allowing for constant-time allocation of new list nodes. If two blocks are assigned the same id, the latest block replaces the previous log entries to reduce the number of system calls that have to be replayed.

When recovering from a power outage, the entire log is replayed following an approach similar to Neverlast's fine-grained data persistence (see Section 3). This fine-grained persistence is beneficial for two reasons: Firstly, the amount of data that needs to be persisted on and restored after a power failure remains low, allowing for fast computation continuation after power restoration. Secondly, as every system call might be interrupted by a power-failure interrupt, we can ensure that the system call can be repeated once the power is restored. Such interruptions of system calls, however, are not uncommon[3] and thus should be handled appropriately anyways.

Figure 2 provides an example for such device-configuration restoration: An application uses (i.e., opens) and configures (i.e., writes to) a sensor. The `write` syscalls

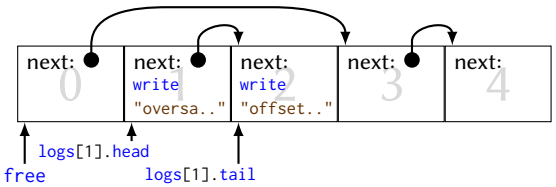



**Figure 1: Example of the data structures prior to the power-failure interrupt illustrated in Figure 2**

used for configuration are enclosed by the replay operations (with id 1) and, by that, persistently logged for replay. Figure 1 illustrates the state of the static log array after executing the replay block.

When recovering from a power outage, the sensor is reopened, and the previously persisted configuration is written back to the sensor, allowing the application to continue using the sensor. Figure 2b shows the effective sequence of operations as obtained from the log array shown in Figure 1. The system calls `open` and `close` are used for signaling the device and energy manager that a device is (no longer) in use, as detailed in the following Section 4.2.

## 4.2 Device and Energy Manager

The second addition to Neverlast's design is the device and energy manager (DEM) that serves two key functions:
1) managing the system's access to external devices
2) powering devices depending on the system's power input

Based on the syscalls described in Section 4.1, the DEM tracks the number of `open` and `close` syscalls for each device. Once a device is not or no longer required (i.e., no calls to open yet, or all opened handles closed again), the DEM powers it down to reduce power consumption. When the device is acquired again, it is powered up and initialized.

Additionally, the DEM implements an emergency switch-off of energy-intensive devices (see Figure 3 for a comparison of power demands of devices used in edge systems) to prevent the whole system from shutting down during phases of low power input. Therefore, all devices are assigned priorities and power thresholds to allow the DEM to prioritize devices critical to the system's computational progress. This emergency switch-off is realized by way of Neverlast's power-failure event handling. The system receives a power-failure interrupt once the power supply falls below one of the predefined thresholds and signals the DEM that energy is scarce and devices must be powered down. Any process currently communicating with one of the powered-down devices is blocked (and thus excluded from scheduling) until the corresponding device becomes available again. The same mechanism is applied to processes newly requesting access to one of the powered-down devices. Once the power supply

[2]Note that a system call not necessarily involves a switch between contexts or permission boundaries but can be seen as an interface between the application and the operating system.
[3]On POSIX-compliant systems, for instance, system calls might be interrupted and aborted by arriving signal and must be restarted.

```c
int sensor = open(/* ... */);
replay_start(sensor, 1);
write(sensor, "oversampling x16");
write(sensor, "offset +2");
replay_stop(sensor);
/* ... */
// ϟ power-failure interrupt
```

**(a) Code prior to power-failure interrupt**

```c
// reopen sensor from line 1
int sensor = open(/* ... */);
// restore config set in line 3
write(sensor, "oversampling x16");
// restore config set in line 4
write(sensor, "offset +2");
```

**(b) Restoring the device configuration after power recovery**

**Figure 2: Pseudo-code example illustrating the device-related operations after recovery from power failure**

has stabilized on a level above the required threshold, the devices are again powered up, and their previous state is restored. All processes blocked during the downtime of these devices are reactivated to allow for their rescheduling.

## 5 EVALUATION

In this Section, we assess our prototype's temporal and energetic behavior, assess the overhead induced by our approach, and illustrate the need for device awareness.

*Experimental Setup.* The following evaluations are based on the TI MSP-EXP430FR5994 LaunchPad Development Kit with an MSP430FR5994 16-Bit RISC processor running at a maximum frequency of 24 MHz. The MSP430FR5994 is equipped with 256 KB of FRAM and an additional 8 KB of SRAM. The FRAM runs at a maximum frequency of 8 MHz, resulting in access times of up to three processor cycles (at a processor frequency of 24 MHz), while the access time to the SRAM is always one cycle. The FRAM controller comes with a 2-way-2-line read cache with a cache-line size of 8 Byte [25, 29], which is shared between code and data residing in FRAM. If not stated otherwise, the code is executed directly from FRAM during the following evaluations.

The MSP-EXP430FR5994's onboard supercapacitor is disconnected during all evaluations. Other peripheral devices, such as the eUSCI modules, DACs, or timers, remain disabled to prevent them from influencing the evaluation. The onboard debugging and flashing tool is disconnected during measurement to prevent external influences. For our energy measurements, we measure the voltage drop over a 110 Ω shunt resistor as well as over the whole system using a Tektronix MSO4034 oscilloscope. The measurement and evaluation process is automated using Python and based on PyVISA[4], and h5py[5]. The whole evaluation is conducted using our adaption and extension of Neverlast to the MSP-EXP430FR5994.

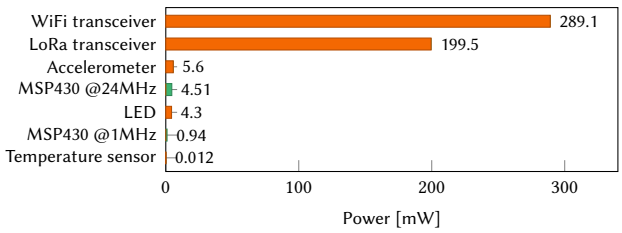



**Figure 3: Power demand of external devices [6, 11, 23] and the MSP430FR5994 (see Section 5.1 for details)**

### 5.1 Basic Power Consumption

This evaluation is designed to give a first impression of the power consumption of our evaluation platform, the MSP-EXP430FR5994, to explain the impact on the energy consumption when changing the processor's core frequency and turning on the power-failure detection, and for comparison to the energy consumption of other peripheral devices. To measure the power consumption, we change the processor frequency to the desired value and average the power demand over 10 s of spinning in an endless loop. For every frequency, we determine the average power demand with disabled/enabled voltage comparator (Comp E) and reference voltage generator (REF_A). Figure 4 illustrates the results of this measurement: When running at a frequency of 1 MHz, our system consumes 0.84 mW respective 0.94 mW without and with power-failure detection. With increasing frequency to up to 24 MHz, the maximum power consumption is 4.51 mW/3.72 mW. Enabling the power-failure detection increases the system's power consumption by 0.10 mW at 1 MHz to 0.79 mW at 24 MHz. When comparing the MCU's power consumption to the power consumption of external devices (see Figure 3), we observe that even small devices (such as an indicator LED) can have considerable power consumption, rendering the additional power consumed by the power-failure detection reasonable. While depending on the device, external devices can play an important role in the system's overall power demand and thus must be considered.

[4]https://github.com/pyvisa/pyvisa
[5]https://www.h5py.org/

## 5.2 Implications of Using FRAM

When building a system whose design centers around using non-volatile memory (i.e., FRAM) to enable fast power-failure handling, the memory's timely and energetical behavior becomes relevant. To evaluate the FRAM's behavior, especially in comparison to its SRAM counterpart, we measure the time and energy required for bytewise reading and writing of 2048 Bytes from/to SRAM and FRAM. As compiler optimizations are known to influence benchmarks [27], for instance, by employing loop optimizations or inserting additional memory accesses, we use hand-written assembly routines directly executed from SRAM for this evaluation. For comparability, we ensure that the corresponding read and write benchmarks exhibit the identical number of calculation cycles, that is clock cycles required only for the calculation without any additional delays from memory accesses. To further increase the measurement precision, we unsoldered most of the buffering and smoothing capacitors for this evaluation, reducing the supply-rail capacitance from 11.5 $\mu$F to only 100 nF. This capacitance reduction does not change the overall power consumption but allows for measuring faster changes in power consumption, and increases the accuracy of measuring the energy of particular operations.

Figure 5 illustrates the measurements of execution time and energy consumption for processor frequencies ranging from 1 MHz up to 24 MHz. All measurements represent the average of ten repetitions (the observed deviations are insignificant and too little to visualize them).

The fast-read benchmarks use the register-indirect–autoincrement addressing mode that is only available for the source operand on the MSP430 (i.e., there is no fast-write implementation). In addition, we demonstrate the effects of the FRAM's 2-way-2-line read cache by using a different access pattern for reading and writing the 2048 Bytes (acread and acwrite in Figure 5): Instead of sequential accesses, we read/write every 8th Byte to reduce the cache's hit rate.

We observe that, for processor frequencies below 8 MHz (which is also the FRAM's maximum access frequency), the execution times for the corresponding read and write operations to FRAM and SRAM are identical. When

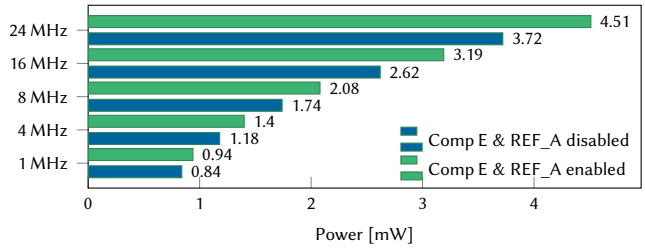



**Figure 4: Power consumption executing an endless loop (avg. over 10 s) per CPU frequency with enabled / disabled power-failure detection (Comp E & REF_A)**

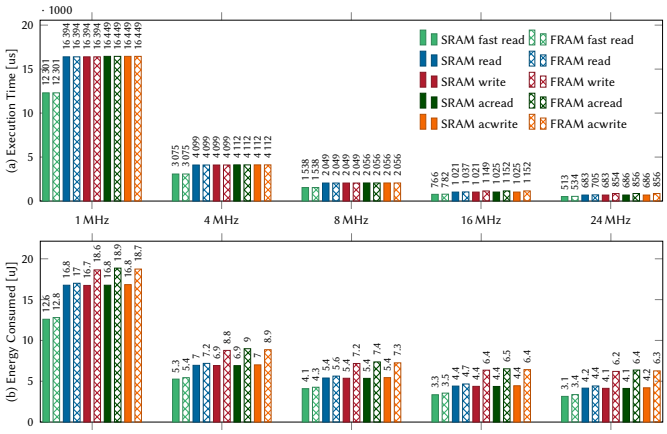



**Figure 5: Execution times and energy consumptions for reading/writing 2048 Bytes from/to SRAM or FRAM using different access patterns.**

surpassing the boundary of 8 MHz, the execution times for reading from and writing to the SRAM remain equal. The timings for FRAM accesses, however, diverge: Due to the FRAM's read cache, sequential reading remains fast, while non-sequential readings incur cache misses and are thereby observably slower. As the FRAM controller does only have a read but not a write cache, write accesses always suffer from the FRAM's limited access speed.

Due to the MSP430FR5994's comparably high static power consumption when running at low frequencies (see Section 5.1), the overall energy consumed for accessing the whole 2048 Bytes is dominated by the benchmark's execution time and, by that, gradually declines with increasing processor frequencies (see Figure 5b). However, when running at the same processor frequency, the FRAM variants consistently consume slightly more energy than their SRAM counterparts, even for benchmarks with identical execution times. Even though the FRAM's access speed is limited to 8 MHz, we see a reduction of execution time and energy consumption with frequencies up to the maximum of 24 MHz.

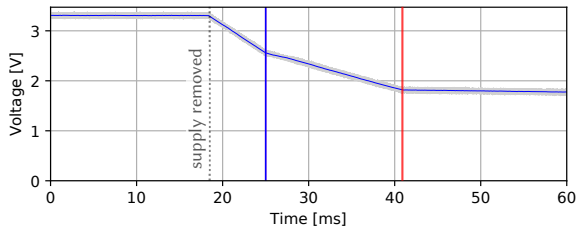
For our approach, we conclude that the advantage of the FRAM's non-volatility (and thus the fast and energy-efficient whole-system power-failure handling) outweighs its slightly increased energy consumption and (at high frequencies) slightly increased execution time.

## 5.3 Time Remaining after Power Failure

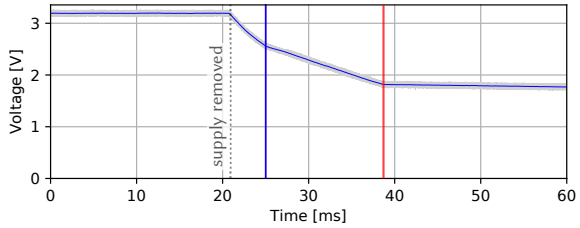
As edge systems have an imminent risk of power failures, it is important to ensure power failures can be handled properly. The central indicator for successfully handling power outages is the execution time remaining after the power outage, which is used for persisting the whole system's state. The remaining time is defined by two properties: The remaining

stored energy and the rate at which the whole system consumes this energy. Thus, the time remaining depends both on the MCU's power demand and the peripheral devices.

This last evaluation concerns the energy remaining after a power failure to get an in-depth understanding of what happens when our system loses power by measuring the voltage drop after disconnecting the system's power supply. During this evaluation, the MSP430FR5994 runs CPU- and memory-intensive matrix multiplications at 8 MHz, while an external device, a LED (4.3 mW), is connected and all data is kept in the FRAM. Figure 6 shows the low-pass–filtered voltages over the whole system (blue line), with the grey shadow representing the measurement's raw, non-filtered data. The left line indicates the time the power supply was disconnected, and the middle vertical blue line marks the beginning of the context-saving routine (i.e., when the supply voltage falls below the configured threshold). The right red line marks the time the brownout reset occurred, a protection mechanism provided by the hardware-level on-chip supply-voltage supervisor. The context saving routine (that in this evaluation only stores the processor registers to FRAM) takes 4.7 $\mu$s, irrespective of the state of the external device. In this experiment, the LED stays activated until brownout to demonstrate the necessity of power management. We observe that the activated LED reduces the overall time between the power outage and the brownout reset from 22 ms to 18 ms. In addition, once the power-fail ISR was executed, the rate the system voltage decreases slows down as the system no longer computes matrix multiplications in the FRAM and, by that,




**(a) External device (LED, 4.3 mW) off**




**(b) External device (LED, 4.3 mW) on**

**Figure 6: The voltage across the MSP430FR5994 (8 MHz) with and without one active external device over time after the power supply was disconnected.**

exhibits a reduced power consumption. From these observations, we conclude that our approach of handling external devices during power failures is vital to ensure timely and proper handling of these power failures. For devices with higher power demand (such as WiFi transceivers), the safety margins can be increased by hardware approaches, for instance, by increasing the capacitance (e.g., by using the onboard supercapacitor) even further, in addition to the software-based approach we proposed (see Section 4.2).

Overall, the presented evaluations not only provide an in-depth insight into our evaluation platform but also analyze our operating system's timely and energetical behavior both during runtime and on power failure and demonstrate the need for such systems.

## 6 CONCLUSION & FUTURE WORK

The increasing demand for computing systems that reliably operate on limited energy resources yields new challenges at runtime for operating systems to handle spontaneous power losses. In particular, deeply embedded devices (i.e., smart dust) and systems that build the backbone of larger computing infrastructures (i.e., edge computing systems) must ensure that computational progress is ensured despite frequent power failures. As these systems typically encompass peripheral devices, approaches to persisting computational progress also need to persist devices and their internal state. In this paper, we presented the extension of the NVM-based operating system Neverlast to include support for persisting peripheral devices by utilizing an extended syscall interface.

As part of our future work, we plan on optimizing our logging mechanism to further reduce the amount of data that has to be written to the log. Additionally, we intend to adapt several device drivers to adhere to Neverlast's driver interface and thereby provide an extended, real-world evaluation scenario including those devices to further illustrate the flexibility and performance of our approach. Finally, once we have access to more powerful, complex devices coming with NVRAM available, we intend to port Neverlast to these devices, including adaptions such as explicit cache handling.

*The source code of Neverlast is available at:*
https://gitlab.cs.fau.de/neverlast

## ACKNOWLEDGMENTS

We thank our shepherd, Sudarsun Kannan, for aiding us in preparing the paper's final version. This work is partly funded by the German Research Foundation (DFG) – Project Number 146371743 – TRR 89 Invasive Computing, under Individual Research Grants NO 625/7-2 and SCHR 603/10-2 (COKE), and by the German Federal Ministry of Education and Research (BMBF) under project AI-NET-ANTILLAS 16KIS1315.

# REFERENCES


[1] Katelin Bailey, Luis Ceze, Steven D Gribble, and Henry M Levy. 2011. Operating System Implications of Fast, Cheap, Non-Volatile Memory. In *Proceedings of the 13th USENIX Workshop on Hot Topics in Operating Systems (HotOS'11)*, Vol. 13. 2–2.

[2] Domenico Balsamo, Alex Weddell, Geoff Merrett, Bashir Al-Hashimi, Davide Brunelli, and Luca Benini. 2015. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *IEEE Embedded Systems Letters* 7, 1 (2015), 15–18.

[3] Gautier Berthou, Pierre-Évariste Dagand, Delphine Demange, Rémi Oudin, and Tanguy Risset. 2020. Intermittent Computing with Peripherals, Formally Verified. In *Proceedings of the 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'20)*. 85–96.

[4] Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. 2018. Sytare: A lightweight kernel for NVRAM-based transiently-powered systems. *IEEE Trans. Comput.* 68, 9 (2018), 1390–1403.

[5] Naveed Anwar Bhatti and Luca Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-Powered Embedded Sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'17)*. 209–220.

[6] Bosch Sensortec. 2020. BME280 Combined humidity and pressure sensor.

[7] Adriano Branco, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. 2019. Intermittent Asynchronous Peripheral Operations. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems (SenSys'19)*. 55–67.

[8] 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'16)*. 514–530.

[9] Andy Davis, Jay Parikh, and William Edward Weihl. 2004. Edgecomputing: Extending Enterprise Applications to the Edge of the Internet. In *Proceedings of the 13th International World Wide Web Conference (Alternate Track, Papers and Posters) (WWW Alt.'04)*. 180–187.

[10] Christian Eichler, Henriette Hofmeier, Stefan Reif, Timo Hönig, Jörg Nolte, and Wolfgang Schröder-Preikschat. 2021. Neverlast: Towards the Design and Implementation of the NVM-based Everlasting Operating System. In *Proceedings of the 54th Hawaii International Conference on System Sciences (HICSS54)*. 1–10.

[11] Espressif Systems. 2018. ESP8266EX Datasheet Version 6.0.

[12] 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 (SenSys'17)*. Article 21, 6 pages.

[13] Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys'17)*. Article 17, 13 pages.

[14] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. 2013. Fine-Grained Fault Tolerance Using Device Checkpoints. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. Association for Computing Machinery, 473–484.

[15] Joseph M. Kahn, Randy H. Katz, and Kristofer S. J. Pister. 1999. Next Century Challenges: Mobile Networking for "Smart Dust". In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99)*. 271–278.

[16] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent Computing: Challenges and Opportunities. In *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL'17)*, Vol. 71. 8:1–8:14.

[17] 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)*. 575–585.

[18] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution without Checkpoints. *Proceedings of the ACM on Programming Languages (PACMPL'17)* 1, OOPSLA, Article 96 (2017), 30 pages.

[19] Kiwan Maeng and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 129–144.

[20] Dushyanth Narayanan and Orion Hodson. 2012. Whole-System Persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. 401–410.

[21] Benjamin Ransford and Brandon Lucia. 2014. Nonvolatile Memory is a Broken Time Machine. In *Proceedings of the 9th Workshop on Memory Systems Performance and Correctness (MSPC'14)*. Article 5, 3 pages.

[22] Alberto Rodriguez Arreola, Domenico Balsamo, Geoff V. Merrett, and Alex S. Weddell. 2018. RESTOP: Retaining external peripheral state in intermittently-powered sensor systems. *Sensors* 18, 1 (2018), 172.

[23] Shenzhen HOPE Microelectronics Co. 2019. RFM95/96/97/98(W) - Low Power Long Range Transceiver Module.

[24] Hyeonho Song and Sam H. Noh. 2017. FSL: Fast system launch through persistent computing with nonvolatile memory. In *Proceedings of the 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA'17)*. IEEE, 1–5.

[25] Texas Instruments Incorporated [n.d.]. *MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide (SLAU367P)*. Texas Instruments Incorporated. October 2012 – Revised April 2020.

[26] Melekam Tsegaye and Richard Foss. 2004. A comparison of the Linux and Windows device driver architectures. *ACM SIGOPS Operating Systems Review* 38, 2 (2004), 8–33.

[27] Reinhold P. Weicker. 1988. Dhrystone benchmark: rationale for version 2 and measurement rules. *ACM SIGPLAN notices* 23, 8 (1988), 49–62.

[28] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 17–32.

[29] Michael Zwerg, Adolf Baumann, Rüdiger Kuhn, Matthias Arnold, Ronald Nerlich, Marcus Herzog, Ralph Ledwa, Christian Sichert, Volker Rzehak, Priya Thanigai, et al. 2011. An 82$\mu$A/MHz microcontroller with embedded FeRAM for energy-harvesting applications. In *2011 IEEE International Solid-State Circuits Conference (ISSCC'11)*. IEEE, 334–336.