Neverlast: Towards the Design and Implementation of the NVM-based Everlasting Operating System

Christian Eichler[1], Henriette Hofmeier[1], Stefan Reif[1], Timo Hönig[2], Jörg Nolte[3], and Wolfgang Schröder-Preikschat[1]

[1]Friedrich-Alexander University Erlangen-Nürnberg (FAU)
{eichler,hofmeier,reif,wosch}@cs.fau.de

[2]Ruhr University Bochum (RUB)
timo.hoenig@rub.de

[3]Brandenburg University of Technology (BTU) Cottbus-Senftenberg
joerg.nolte@b-tu.de

Abstract

Novel non-volatile memory (NVM) technologies allow for the efficient implementation of "intermittently-powered" smart dust and edge computing systems in a previously unfamiliar way. Operating with rough environmental conditions where power-supply failures occur often requires adjustments to all parts of the system. This leads to an inevitable trade-off in the design of operating systems—the overhead of persisting *the achieved computation progress over power failures is detrimental to the possible amount of progress with the available energy budgets. It is, therefore, crucial to minimize the overhead of ensuring persistence.*

This paper presents the case that persistence should be provided as an operating-system service to achieve everlasting operating capabilities. Triggered by power-failure interrupts, an implicit persistence service for the processor status of a process preserves progress on the CPU-instruction level. This interrupt only triggers if necessary so that no power-state polling is needed. We outline architectures for everlasting systems and discuss their benefits and drawbacks compared to existing approaches. Thereby, the operating system provides persistence as a service *at run-time to the application, with minimal overhead. Our approach enables the separation of the application from energy-supply state estimation, as well as state-preserving logic for software and hardware components.*

1. Introduction

Recent hardware advancements enable computing systems to operate in harsh environments where the most fundamental resource for switching gate logic, energy, is scarce. When no power-supply infrastructure is available, computing nodes apply energy-harvesting techniques, and operate whenever their environment supports it [1, 2, 3, 4, 5, 6]. This approach of "intermittently-powered" hardware enables a plethora of novel applications, such as environment-monitoring systems that incorporate smart dust [7] and edge computing systems [8].

Operating with rough environmental conditions where power-supply failures are becoming the norm, rather than the exception, requires adjustments to all parts of the system—the processor and memory hardware as well as the software components (e.g., operating system and system software) [9]. The system must be appropriately prepared for spontaneous power loss, yet the mechanism to cope with it (e.g., to preserve progress) may eat up a significant amount of the scarcely available energy. This leads to an inevitable trade-off in the design of everlasting[1] operating systems—the overhead of *persisting* the achieved computation progress over power failures is detrimental to the possible amount of progress with the available energy budgets. It is, therefore, crucial that the persistence comes at the minimum possible overhead.

A variety of approaches have been developed to implement *everlasting* systems that cope with temporary power loss. Many of these approaches were designed for systems where power loss is an infrequent occurrence, such as suspend-to-disk mechanisms. The hardware requirements for everlasting systems are anything but new; they were already implemented in computers from the 1970s (e.g., PDP-11, VAX-11/780: power failure trap, "occurs whenever the AC power drops below 95 volts or outside 47 to 63 Hertz. Two milliseconds are then allowed for power down

[1]Everlasting operating systems boot exactly once, become idle and suspend during operation very often, but run forever, eventually.

URI: https://hdl.handle.net/10125/71491
978-0-9981331-4-0
(CC BY-NC-ND 4.0)

processing." [10]). This seemingly forgotten technology is now being modeled on a USB basis, for example, for computer systems with an uninterruptible power supply (UPS), which can deliver a power-fail interrupt to the CPU. There are special adaptations for microcontroller-based systems, independent of USB, which, as a workaround, reintroduce the original characteristic of a power-fail-aware CPU. In contrast to the original mandatory trap-based exception, however, these interrupt-based approaches can generally be masked, which can increase the latency for exception handling and limit applicability.

For an intermittently-powered system, the overhead of saving and restoring the entire system state is far too expensive—the available energy budget might be insufficient to restore the system from permanent storage. An alternative is relying on transactions which guarantee that every state is *recoverable*: a logically consistent state can always be restored from which computation then resumes [11, 12]. For transactions, overhead originates from operations that implement necessary roll-back or roll-forward functions for recovering the system state after power losses. This raises the question of the granularity of transactions. On the one hand, if the granularity is too coarse, it is possible that power loss occurs before reaching a consistent "checkpoint", and the following roll-back has to undo all achieved progress. On the other hand, if the granularity is too fine, the operations for checkpoint creation consume unnecessarily much energy.

This paper presents the case that persistence should be provided as an operating-system service to achieve everlasting operating capabilities. We outline architectures for everlasting systems, discuss their benefits and drawbacks compared to existing approaches, and also outline the potential for future developments. Thereby, the operating system provides persistence as a run-time *service* to the application, with minimal overhead. Our approach enables the application to be independent of energy-supply state estimation, as well as state-preserving logic for software and hardware components.

The contributions of this paper are the following. First, we discuss system designs to achieve everlasting systems that can handle frequent energy-supply failures. Second, we show the benefits of system-level approaches over existing solutions. Third, we present a prototype OS, Neverlast, based on an MSP430 platform, and evaluate its behavior on power loss—considering time and energy—on real hardware.

The rest of the paper is structured as follows. Section 2 introduces the necessary background information and discusses related work. Design considerations of everlasting systems are presented in Section 3. Section 4 presents the implementation of our prototype operating system, which is evaluated in Section 5. The final Section 6 concludes.

2. Background and Related Work

Byte-addressable non-volatile memory (NVM) technologies offer data persistence without flash memory's long read and write latency while also showcasing a significantly improved write endurance. Some technologies even come close to DRAM and SRAM in regard to latency, energy, and cell size. Table 1 shows a comparison between a selection of the most promising NVM technologies and SRAM, DRAM, and NAND Flash. One of the best-established NVM technologies is ferroelectric RAM (FRAM), which is similar to DRAM in its structure and offers latencies in the range of DRAM. An FRAM cell, like DRAM, is composed of a capacitor and a transistor. The difference between the two technologies lies in the material of the capacitors, which is dielectric in a DRAM cell and ferroelectric in FRAM. This material allows for FRAM's non-volatility but also hinders downscaling the cell size [15]. In addition to the destructive read operations of FRAM, which limit its endurance, the comparatively large cell size reduces FRAM's potential to replace DRAM as the dominant main memory technology. Therefore, current research and development of persistent main memories are more focused on PCRAM, STT-RAM, and RRAM. However, to the best of our knowledge, off-the-shelf hardware with integrated NVM is only available with FRAM.

NVM technologies enable systems that preserve their computation progress at fine granularity, in particular, all store operations to non-volatile RAM are persistent [9]. 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" [22] 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. An example is a reference where the pointer is persistent, but the referenced data is still in volatile caches and therefore lost upon power failure. After a power loss, the pointer itself would be valid, but the referenced data is lost, leading to an inconsistent state of the system. 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.

		SRAM	DRAM	PCRAM	STT-RAM	FRAM	NAND-Flash
Latency	Write	0.2 ns	10 – 50 ns	50 – 500 ns	2 – 35 ns	50 – 65 ns	200 – 350 μs
	Read	0.2 ns	10 – 50 ns	20 – 50 ns	2 – 20 ns	20 – 80 ns	15 – 35 μs
Endurance [cycles]		$> 10^{16}$	$> 10^{15}$	10^{8} – 10^{12}	10^{12} – 10^{15}	10^{12} – 10^{14}	$> 10^{5}$
Minimum Cell Size [F^2]		120 – 200	6 – 10	4 – 12	6 – 50	20 – 40	4 – 6

Table 1: Overview of the properties of different memory technologies, as reported by [13, 14, 15, 16, 17, 18, 19, 20, 21]. F denotes the feature size of transistors.

Traditional approaches to ensure logical data consistency despite potential power losses include transactions in databases. For example, Android uses SQLite databases stored on flash storage. However, this approach has significant overheads within each database access, including database-level and file-system-level transaction logs, as well as complex file-system caches [23, 24]. Similarly, suspension-based approaches that dump the entire system state to disk have a significant data-copy overhead. In summary, traditional approaches are not suitable for intermittently-powered systems due to their large overheads to suspend and resume execution.

The state of the art techniques in programming intermittently-powered systems can be broadly classified into one of two categories. First, specific programming models separate computation from non-volatile storage. Exemplary systems are Dino [3] and Chain [25], which contain atomic (i.e., all-or-nothing) “tasks” that communicate via persistent “channels”, and InK [26], which applies an event-driven program structure. Similarly, dedicated programming languages such as Mayfly [6] enable programmers to express their application through data flows, abstracting away from the persistence of storage. Second, transaction-based systems [1, 11, 12, 27] use compiler extensions and run-time support systems that automatically identify suitable checkpoint locations and insert checkpoints accordingly that backup volatile data to persist the achieved progress. As mentioned in Section 1, the checkpoint frequency is crucial, because it determines the progress granularity, but the necessary operations for each checkpoint cause overhead in energy and time demand. However, the compiler has no information about the power-supply state at run-time. This problem has led to approaches with *conditional* checkpoints, whereby the software effectively polls the current energy-supply state at high frequency. The system safely skips checkpoints as long as the battery is sufficiently charged, and enables checkpointing only if the charge is low. However, these frequent battery-state check operations may still come at the cost of increased execution time and energy consumption.

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. Our design thus decouples the power-monitoring functionality from the application-checkpoint logic. Another approach that also uses an interrupt on power loss is described by Narayanan et al. [28], but their system targets server-size computers with completely different performance and power characteristics and much less frequent power failures.

One of the remaining problems with language-based approaches are peripheral devices that lose internal states on power failure [29, 30], and interrupt handling [31]. Volos et al. [32] discuss the design of suitable device drivers.

3. Design

The design of Neverlast, the *everlasting operating system*, makes persistence *implicit*, so that all progress is preserved over power losses by default. Instead of explicit models where application code is transformed into transaction patterns to maintain data consistency over power losses, our design provides a *virtually persistent* processor. This virtual processor executes the application with a transparent persistence service that augments the physical processor—with volatile internal state—for implicitly persistent code execution. The granularity of progress in Neverlast is a single CPU instruction, rather than “tasks” [25, 27] or application-level transaction checkpoints. While some approaches that Neverlast utilizes internally have already been proposed in previous work, we are not aware of systems that provide persistence implicitly.

The Neverlast persistence service covers the applications, and necessarily all OS-level resources that are referenced by the applications, as well. Thus, the persistence service has to operate logically on a layer between the hardware and the operating system, similar to a hypervisor, yet it is not entirely transparent to the operating system—Neverlast itself has to interact

explicitly with this layer, for example, in the interrupt handling subsystem.

This design enables the persistence service to select a suitable *persistence strategy* independently of the application code. This strategy can cooperate with other system-level strategies, for example, frequency scaling. As shown in the evaluation, frequency scaling has a significant effect on both execution time as well as the energy demand of application-level code. In consequence, the persistence service should be aware of frequency scaling to optimize energy efficiency jointly.

A key advantage of Neverlast over compiler-based approaches is that run-time information on the power-supply state is available. This information enables an interrupt-based power-shortage notification mechanism instead of wasteful polling. The interrupt-based model enables fine-grained progress, where each executed instruction is implicitly provided a persistence guarantee. 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.

Besides a virtually persistent processor, Neverlast provides *virtually persistent devices*, using device drivers that abstract away from power outages. For example, these drivers store volatile states of peripheral devices on power failure and restore their state once the system continues operation. In this paper, however, we focus on persistent process execution because in-depth discussions about device drivers for intermittently-powered systems can be found in the literature [29, 30, 31].

3.1. Power-failure Event Handling

The key component to enable implicitly persistent application execution is the timely detection of power loss. This detection should be *safe*, in the sense that the detection happens early enough to guarantee complete execution of the event handler functions. A safe detection requires cooperation between hardware, software, and an analysis component. First, the hardware has to detect power loss in advance while still retaining enough energy to execute the power-loss event handler. This is, in practice, feasible because microcontrollers use capacitors for ultra-short-term power supply, and a brown-out detection hardware to avoid under-voltage situations where the gate logic can no longer operate reliably. By adapting the brown-out detection to deliver a processor interrupt when the power supply starts to faint, rather than a hard reset after power-supply failure, the operating system can appropriately prepare for the imminent power loss. A similar interrupt-based—but not fully transparent—approach is used by Hibernus [2].

In addition to the hardware power-supply failure detection, and the interrupt handler that copies all volatile state to non-volatile memory, a system analysis is required to guarantee that the remaining energy is sufficient to save the entire volatile state. Such an analysis can use established sound worst-case energy consumption (WCEC) analysis techniques [33].

The time and energy needed to execute the power-failure event handler depend on the hardware, in particular on the amount of volatile state that requires a write-back operation. By default, we intentionally minimize the amount of volatile state in Neverlast to prepare for unreliably-powered systems operating in environments where the power supply fails often.

3.2. Fine-grained Data Persistence

One of the necessary trade-offs for endless systems is the amount of volatile state. During operation, volatile memory can be faster and more energy-efficient [34, 35], but on power-failure, its data must be copied to non-volatile storage.

This trade-off leaves three design options for Neverlast, where the efficiency depends on the frequency of power-failure situations. First, data is placed in non-volatile memory exclusively to minimize the overhead of the copy operation on power failure. Narayanan et al. [28] have demonstrated that this approach improves energy efficiency over transaction-based approaches in server systems. The minimal set of volatile data are the CPU registers, and if present, CPU caches. Since our target embedded platform has no volatile caches[2], only the CPU registers need to be saved on power loss. Second, a part of the application data can be located in volatile storage, and on power failure, this data has to be copied. This hybrid approach has been used by Hibernus [2], but it is optional in Neverlast since the data-copy overhead is detrimental to the granularity of progress if power-supply failures occur often. Third, the application can be *volatility-aware* and employ individual strategies, such as transactions, for performance-critical parts. This explicit approach does not contradict our model of implicit persistence. Instead, it enables applications to voluntarily combine both techniques if the combination improves efficiency, with the drawbacks of higher application complexity and potential roll-back overheads.

[2]Only the FRAM controller contains an internal cache, but it ensures write-back on power failure using an emergency capacitor.

For Neverlast, we assume that the amount of volatile data is minimal to support unreliably-powered systems. Our motivation is that each data-copy operation requires time, and if power-supply failures occur often, the copy-related overhead harms the application progress.

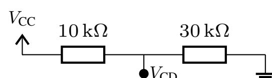
In summary, Neverlast provides an *implicit* persistence service that preserves progress on the CPU-instruction level. As far as possible, data resides in persistent memory to avoid copy operations on power failure. Furthermore, the power-failure interrupt triggers only if necessary, so that inefficient power-state polling is not needed.

4. Implementation

This Section presents *Neverlast*, our prototypical operating system for NVRAM systems that targets the TI MSP-EXP430FR5739 Experimenter Board and builds the foundation for our evaluations. The key services provided by our system are (a) basic operating system features, such as process management and semaphores, and (b) application-transparent, OS-level detection and handling of power failures.

4.1. Power-failure Detection

The MSP430FR5739 comes with an on-chip analog voltage comparator, COMPARATOR_D. COMPARATOR_D can, amongst other features, compare the voltage level applied to its input channels against a built-in reference voltage of $1.5\,\mathrm{V}$, $2.0\,\mathrm{V}$, or $2.5\,\mathrm{V}$ and issue an interrupt once the input-voltage level drops below the configured level. To increase the interrupt threshold to a value above $2.5\,\mathrm{V}$, such as to the $2.67\,\mathrm{V}$ used by our prototype, we use a 1:3 voltage divider, two resistors ($10\,\mathrm{k\Omega}$ and $30\,\mathrm{k\Omega}$) in series, as illustrated below. The circuit below reduces the voltage measured by COMPARATOR_D V_{CD} to $3/4$ of the supply voltage V_{CC}.



The prefactor of $3/4$ and the choice of $2.0\,\mathrm{V}$ as reference voltage makes COMPARATOR_D issue an interrupt once the supply voltage falls below $V_{\mathrm{CD,th}} = 2.67\,\mathrm{V}$ (i.e., $4/3 \cdot 2.0\,\mathrm{V}$). The corresponding interrupt service routine pushes the processor registers to the stack and subsequently writes the resulting stack pointer to a non-volatile memory area. To detect invalid states (e.g., due to the execution stopping midst the ISR due to missing power), we store an additional valid bit after storing the execution context, which is checked and reset on restoring the execution context.

Other parts of the system state, such as the SRAM contents required for our volatile implementation presented in the following, are also immortalized prior to marking the stored execution context as valid.

4.2. Ensuring Non-volatility

To maximize the forward progress of processes running on Neverlast, the amount of data moved during backup- and restore-routines has to be kept to a minimum. Therefore, our system keeps all data permanently in the FRAM, including the stacks for all processes. This leaves the processor registers as the sole remaining state-relevant data in volatile memory. With the power-failure detection described above, an interrupt is triggered on imminent power failure, initiating the backup routine. As part of the interrupt service routine, backups of the volatile processor state, namely the general-purpose registers and stack pointer, as well as a valid-flag, are copied to the non-volatile memory before the system suspends its execution. During the boot routine, the system restores its old state if a consistent backup is detected and resumes its execution.

The entire backup- and restore mechanism emulates an interrupt service routine. With this design, Neverlast can resume its execution at the exact instruction on which it was interrupted as if returning from an interrupt handler and thus making non-volatility transparent to the application.

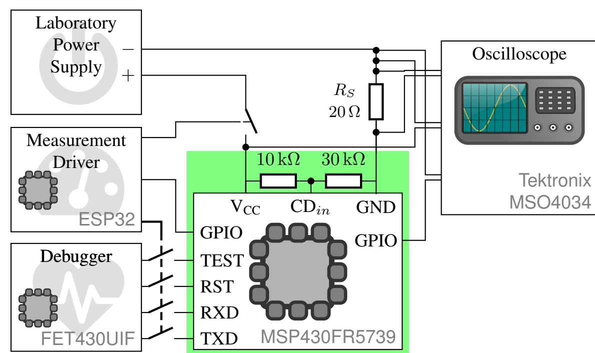
4.3. Process Management

As the first element of Neverlast, we designed and implemented process management. Processes can occupy one of three states: ready, running, or blocked.

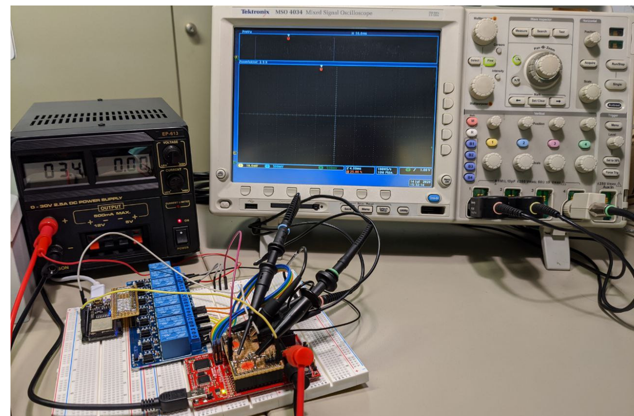
Once a process has been created, it is marked as ready and added to the ready queue. Neverlast's processes can be created at any time during the execution of our system. A cooperative scheduler is responsible for dispatching the process from the ready queue, which then starts its execution until it yields control, allowing the next process to run. Apart from voluntarily yielding control of the CPU, a process can also be forced to relinquish control. Shared resources are synchronized with the help of semaphores. If a process requests a resource that is already in use by its maximum number of processes, the semaphore forces the requesting process to give up control and adds it to the blocked queue. The process is woken and marked as ready as soon as the required resource is available again.

4.4. Volatile Implementation

Our system also offers to place all required data structures in the volatile SRAM instead of allocating



(a) Schematic



(b) Photo

Figure 1: Evaluation setup used for time and energy measurement

space for them in the FRAM during the linking process. When choosing this configuration, different backup- and restore-routines are executed, which copy all relevant data from the SRAM into the FRAM, in addition to saving the processor registers. Therefore, all process stacks, as well as metadata about the process management, are included in the backup. This drastically increases the amount of data transferred during backup and restore, but also allows for a more individual configuration in case non-volatile memory integrated into the chosen hardware should be scarce.

5. Evaluation

In this Section, we assess the temporal and energetic behavior of our prototype, demonstrate the practicality of our implementation, and assess the overhead induced by our approach.

5.1. Experimental Setup

The following evaluations are based on the TI MSP-EXP430FR5739 Experimenter Board[3] with an MSP430FR5739 16-Bit RISC processor running at a maximum frequency of 24 MHz (8 MHz if not specified otherwise). The MSP430FR5739 is equipped with 16 KB of FRAM and an additional 1 KB of SRAM. The FRAM runs at a maximum frequency of 8 MHz, resulting in access times of up to three processor cycles (if running at the processor's maximum frequency of 24 MHz). Unlike the FRAM, the access time to the SRAM is always one processor cycle. To overcome the FRAM's speed limitation, the FRAM controller comes with 2-way-2-line read cache with a cache-line size of 8 Byte [36]. Other peripheral devices present on the MSP430FR5739, such as the eUSCI modules, ADCs, or timers, remain disabled for this evaluation to prevent them from influencing the evaluation. If not stated otherwise, the measurement values presented in this Section are the arithmetic mean of ten repetitions.

The MSP-EXP430FR5739 comes with an on-board debugging and flashing tool that is disconnected during measurement to prevent external influences. For low-level interaction with the MSP430, such as toggling power using relays and checking for error conditions using the MSP430's GPIO pins, we use a FireBeetle ESP32[4] microcontroller along with an off-the-shelf relay card. The ESP32 is connected to the measurement host via USB and provides a human-readable text interface by emulating a serial port. For our energy measurements, we measure the voltage drop over a $20\,\Omega$ 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[5] and h5py[6], Figure 1 pictures the measurement and evaluation hardware described previously along with it's schematic.

5.2. Basic Power Consumption

This evaluation is designed to give a first impression of the power consumption of our evaluation platform, the MSP-EXP430FR5739, and to explain the impact on the energy consumption when changing the processor's core frequency. To measure the power

[3] https://www.ti.com/tool/MSP-EXP430FR5739

[4] https://www.dfrobot.com/product-1590.html

[5] https://github.com/pyvisa/pyvisa

[6] https://www.h5py.org/

```
#define READ_LOOP(MEM, SIZE)              \
    asm volatile(                         \
        "   mov %0, r14                \n"\
        "loop_%=:                      \n"\
        "   mov.b 0(r14), r15          \n"\
        "   inc r14                    \n"\
        "   nop                        \n"\
        "   cmp %1, r14                \n"\
        "   jl loop_%=                 \n"\
      : : "i"(MEM), "i"(&MEM[SIZE])  \
      : "r14", "r15");
```

```
#define WRITE_LOOP(MEM, SIZE)             \
    asm volatile(                         \
        "   mov %0, r14                \n"\
        "loop_%=:                      \n"\
        "   mov.b r15, 0(r14)          \n"\
        "   inc r14                    \n"\
        "                              \n"\
        "   cmp %1, r14                \n"\
        "   jl loop_%=                 \n"\
      : : "i"(MEM), "i"(&MEM[SIZE])  \
      : "r14", "r15");
```

Figure 2: Assembly routines sequentially leading from/writing to memory for comparison of SRAM and FRAM

consumption, we change the processor frequency to the desired value and average the power demand over 5 ms of side-effect–free code. Figure 3 illustrates the results of this measurement: When running on 1 MHz, our system consumes 8 mW. With increasing frequency the power consumption rises to up to 11.3 mW when running at 24 MHz. The observed values indicate that the MSP-EXP430FR5739's yields a comparatively high static power demand, reading to the conclusion that race-to-sleep, if possible, is preferable on this platform.

5.3. SRAM vs. FRAM

When building a system whose design centers around using non-volatile memory (i.e., FRAM), the memory's timely and energetical behavior and their implications on operating-system development become 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 512 Bytes from/to SRAM and FRAM. As compiler optimizations are known to influence benchmarks [37], for instance by employing loop optimizations or inserting additional memory accesses, we use hand-written assembly routines (see Figure 2) for this evaluation. To ensure identical execution times between the corresponding read and write benchmark,

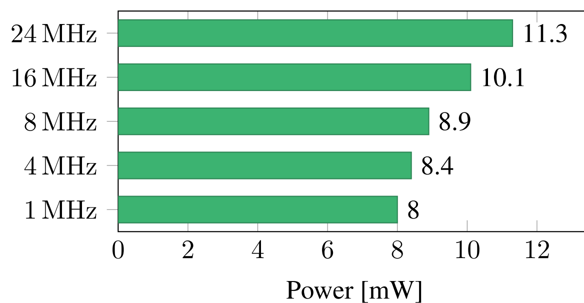


Figure 3: Average power consumption when executing side-effect-free code at different processor frequencies

which is essential for comparability, we purposefully do not use the register-indirect–autoincrement addressing mode (e.g., `mov.b @r14+, r15`) that is only available for the source operand on the MSP430. Further, we insert an additional `nop` instruction to compensate for the one cycle difference in execution time between `mov.b 0(r14), r15` and `mov.b r15, 0(r14)`. Figure 4 illustrates the measurements of execution time and energy consumption for processor frequencies ranging from 1 MHz up to 24 MHz: The fast-read benchmarks use the register-indirect–autoincrement addressing mode and do not contain the additional `nop` instruction. As the register-indirect addressing modes are only available for the source operand on the MSP430, there is no fast-write implementation. In addition to the read and write benchmark, 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 512 Bytes (acread and acwrite in Figure 4): Instead of sequential accesses, we read/write every 8th Byte (i.e., 0 8 16... 1 9 17...) 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 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 MSP430FR5739's comparably high static power consumption when running in active mode, but at low frequencies (see Section 5.2), the overall energy consumed for accessing the whole 512 Bytes is dominated by the benchmark's execution time and,

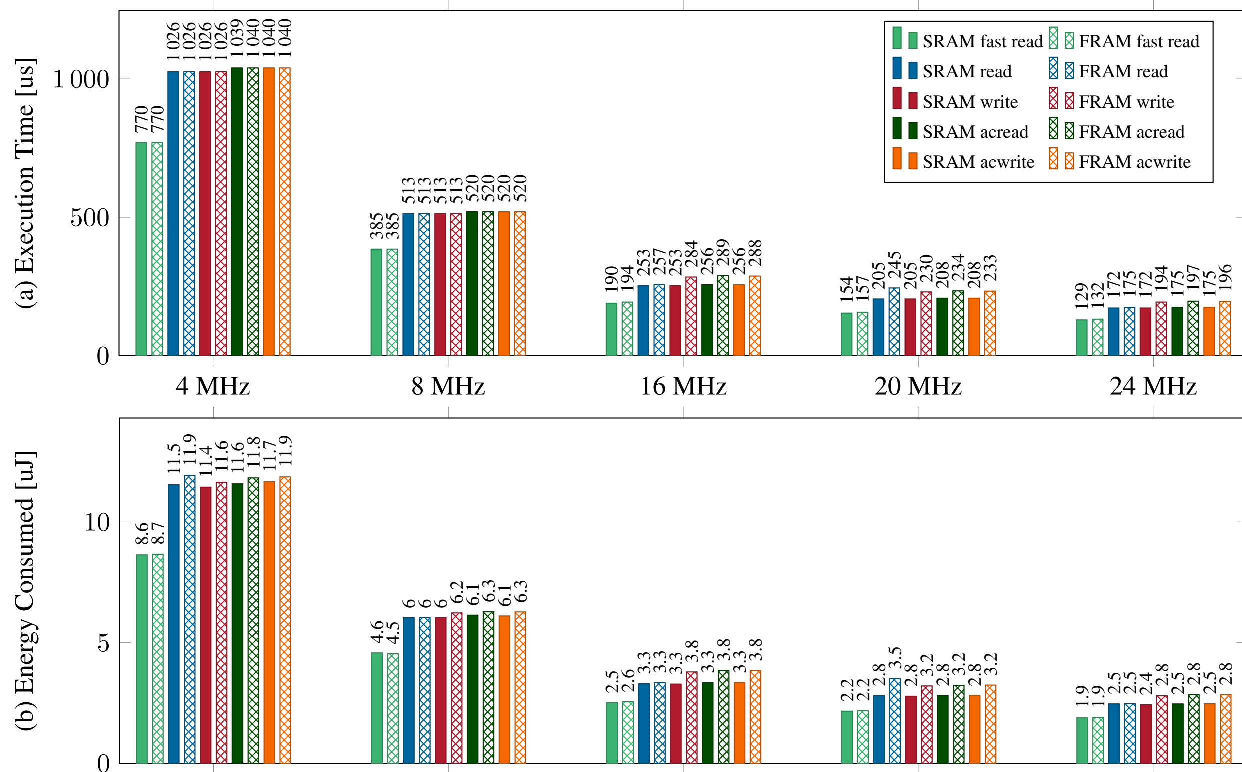


Figure 4: Execution times and energy consumptions for reading/writing 512 Bytes from/to SRAM or FRAM

by that, gradually declines with increasing processor frequencies (see Figure 4b). 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 time. 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 frequency of 24 MHz.

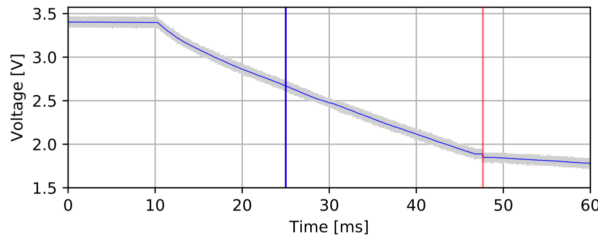
For our operating system, we conclude that the advantage of the FRAM's non-volatility outweighs its slightly increased energy consumption.

5.4. Resilience Against Power Failures

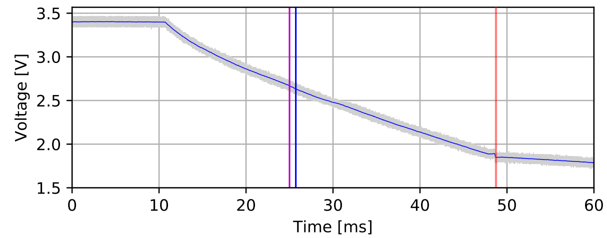
One of our operating system's core features is the resilience against power failures by placing most of the system's data in FRAM and, on power failure, saving register values (and, optionally, the SRAM contents) to FRAM. Therefore, we demonstrate the reliability of our prototype implementation by repeated simulations of power failures. For this purpose, we cut the power supply 1000 times after randomized execution times equally distributed in the range of 0.1 s to 2.5 s. During this test, we monitor the system's error indicator, a dedicated GPIO pin that is set by the operating system whenever an error occurs, such as having an invalid system state or a failing assertion in either the operating system or one of the applications. For both variants, the system using FRAM and SRAM, as well as the system using only FRAM, we did not observe any errors.

5.5. Energy Remaining after Power Failure

This last evaluation is concerned with the energy, and thereby the execution time, remaining after a power failure to get an in-depth understanding of what happens when our system loses power. Figure 5 shows the low-pass filtered voltage over the whole system (blue line), with the grey shadow representing the measurement's raw, non-filtered data. The left-most two vertical lines mark the beginning (magenta) and end (blue) of the context-saving routine, the right, red line marks the time the brownout reset occurred, a protection mechanism provided by the on-chip supply-voltage supervisor that protects the system from malfunctioning. Figure 5a shows the measurement for code only using FRAM, and thereby the context-saving routine only stores the processor registers to FRAM on power loss. The context-saving routine used in



(a) Use FRAM only



(b) Copy SRAM to FRAM

Figure 5: Voltage across the MSP430FR5739 over time after external power supply was disconnected

Figure 5b additionally copies the whole $1\,\mathrm{KB}$ SRAM to FRAM to support applications using the on-chip SRAM.

Storing only the processor registers to FRAM takes $5.6\,\mathrm{us}$, storing both registers and copying the whole $1\,\mathrm{KB}$ of SRAM increases the execution time to $723.4\,\mathrm{us}$. In both scenarios, the time remaining between power outage (starting at $\sim 10\,\mathrm{ms}$) and the brownout reset (indicated by the red lines) is $37\,\mathrm{ms}$ and by that, even for the scenario involving the SRAM dump, more than 50 times the time required for storing the execution context to FRAM.

Overall, the presented evaluations not only provide an in-depth insight into our evaluation platform, but also analyze our operating system's behavior on power failure and demonstrate its practical applicability.

6. Conclusion

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 power failures that occur very often. Based on recent hardware advances (i.e., NVM) we have explored new ways for providing progress guarantees with an everlasting operating system which provides persistence as a service towards applications. The presented approach ensures that persistence is achieved with minimal overhead in time and energy.

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

Acknowledgments: This work is partly funded by the German Research Foundation (DFG) – Project Number 146371743 – TRR 89 Invasive Computing and under Individual Research Grants NO 625/7-2 and SCHR 603/10-2 (COKE).

References

[1] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pp. 129–144, 2018.

[2] D. Balsamo, A. Weddell, G. Merrett, B. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 15–18, 2015.

[3] B. Lucia and B. Ransford, "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)*, pp. 575–585, 2015.

[4] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent computing: Challenges and opportunities," in *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL'17)*, vol. 71, pp. 8:1–8:14, 2017.

[5] J. Hester and J. Sorber, "The future of sensing is batteryless, intermittent, and awesome," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys'17)*, 2017.

[6] J. Hester, K. Storer, and J. Sorber, "Timely execution on intermittently powered batteryless sensors," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys'17)*, 2017.

[7] J. M. Kahn, R. H. Katz, and K. S. J. Pister, "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)*, pp. 271–278, 1999.

[8] A. Davis, J. Parikh, and W. E. Weihl, "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)*, pp. 180–187, 2004.

[9] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "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, pp. 2–2, 2011.

[10] Digital Equipment Corporation, *PDP-11/40 Processor Handbook*, 1972.

[11] J. V. D. Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pp. 17–32, 2016.

[12] N. A. Bhatti and L. Mottola, "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)*, pp. 209–220, 2017.

[13] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1537–1550, 2015.

[14] S. Mittal, J. S. Vetter, and D. Li, "A survey of architectural approaches for managing embedded DRAM and non-volatile on-chip caches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 6, pp. 1524–1537, 2014.

[15] L. Wang, C.-H. Yang, and J. Wen, "Physical principles and current status of emerging non-volatile solid state memories," *Electronic Materials Letters*, vol. 11, no. 4, pp. 505–543, 2015.

[16] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng, "Overview of emerging nonvolatile memory technologies," *Nanoscale research letters*, vol. 9, no. 1, p. 526, 2014.

[17] A. Suresh, P. Cicotti, and L. Carrington, "Evaluation of emerging memory technologies for HPC, data intensive applications," in *2014 IEEE International Conference on Cluster Computing (CLUSTER'14)*, pp. 239–247, IEEE, 2014.

[18] F. Xia, D.-J. Jiang, J. Xiong, and N.-H. Sun, "A survey of phase change memory systems," *Journal of Computer Science and Technology*, vol. 30, no. 1, pp. 121–144, 2015.

[19] S. Baek, J. Choi, D. Lee, and S. H. Noh, "Energy-efficient and high-performance software architecture for storage class memory," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 3, pp. 1–22, 2013.

[20] J. Wang, X. Dong, Y. Xie, and N. P. Jouppi, "Endurance-aware cache line management for non-volatile caches," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 1, pp. 1–25, 2014.

[21] S. Mittal, "A survey of techniques for architecting processor components using domain-wall wemory," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 2, pp. 1–25, 2016.

[22] B. Ransford and B. Lucia, "Nonvolatile memory is a broken time machine," in *Proceedings of the 9th Workshop on Memory Systems Performance and Correctness (MSPC'14)*, 2014.

[23] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones," in *USENIX 2013 Annual Technical Conference (ATC'13)*, pp. 309–320, 2013.

[24] D. Park and D. Shin, "iJournaling: Fine-grained journaling for improving the latency of fsync system call," in *USENIX 2017 Annual Technical Conference (ATC'17)*, pp. 787–798, 2017.

[25] A. Colin and B. Lucia, "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)*, pp. 514–530, 2016.

[26] S. Kasım Yıldırım, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester, "InK: Reactive kernel for tiny batteryless sensors," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SenSys'18)*, pp. 41–53, 2018.

[27] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, 2017.

[28] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, pp. 401–410, 2012.

[29] A. Branco, L. Mottola, M. H. Alizai, and J. H. Siddiqui, "Intermittent asynchronous peripheral operations," in *Proceedings of the 17th Conference on Embedded Networked Sensor Systems (SenSys'19)*, pp. 55–67, 2019.

[30] G. Berthou, P.-E. Dagand, D. Demange, R. Oudin, and T. Risset, "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)*, pp. 85–96, 2020.

[31] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "Sytare: A lightweight kernel for NVRAM-based transiently-powered systems," *IEEE Transactions on Computers*, vol. 68, no. 9, pp. 1390–1403, 2018.

[32] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, pp. 91–104, 2011.

[33] P. Wägemann, C. Dietrich, T. Distler, P. Ulbrich, and W. Schröder-Preikschat, "Whole-system worst-case energy-consumption analysis for energy-constrained real-time systems," in *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS'18)*, pp. 24:1–24:25, 2018.

[34] H. Jayakumar, A. Raha, and V. Raghunathan, "Energy-aware memory mapping for hybrid FRAM-SRAM MCUs in IoT edge devices," in *29th International Conference on VLSI Design and 15th International Conference on Embedded Systems (VLSID'16)*, pp. 264–269, IEEE, 2016.

[35] H. Jayakumar, A. Raha, J. R. Stevens, and V. Raghunathan, "Energy-aware memory mapping for hybrid FRAM-SRAM MCUs in intermittently-powered IoT devices," *ACM Transactions on Embedded Computing Systems (TECS'17)*, vol. 16, no. 3, pp. 1–23, 2017.

[36] M. Zwerg, A. Baumann, R. Kuhn, M. Arnold, R. Nerlich, M. Herzog, R. Ledwa, C. Sichert, V. Rzehak, P. Thanigai, *et al.*, "An 82μa/mhz microcontroller with embedded FeRAM for energy-harvesting applications," in *2011 IEEE International Solid-State Circuits Conference (ISSCC'11)*, pp. 334–336, IEEE, 2011.

[37] R. P. Weicker, "Dhrystone benchmark: rationale for version 2 and measurement rules," *ACM SIGPLAN notices*, vol. 23, no. 8, pp. 49–62, 1988.