Accelerating Programmer-Friendly Intermittent Computing

Kortbeek, V.

DOI
10.4233/uuid:76ff65e4-cf07-4ff4-b3b5-937860e0f675

Publication date
2023

Document Version
Final published version

Citation (APA)
Kortbeek, V. (2023). *Accelerating Programmer-Friendly Intermittent Computing*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:76ff65e4-cf07-4ff4-b3b5-937860e0f675

Accelerating Programmer-Friendly Intermittent Computing

ACCELERATING PROGRAMMER-FRIENDLY INTERMITTENT COMPUTING

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op donderdag 29 juni 2023 om 12:30 uur

door

Vito KORTBEEK

Master of Science in Embedded Systems,
Technische Universiteit Delft, Delft, Nederland,
geboren te Heemskerk, Nederland.

Dit proefschrift is goedgekeurd door de promotoren.

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof. dr. K.G. Langendoen,	Technische Universiteit Delft, promotor
Dr. P. Pawełczak,	Technische Universiteit Delft, promotor
Onafhankelijke leden:	
Prof. dr. ir. G.N. Gaydadjiev,	Technische Universiteit Delft
Prof. dr. ir. M.J.G. Bekooij	Universiteit Twente
Prof. dr. J. Reineke,	Universiteit van het Saarland, Duitsland
Dr. M. Hicks,	Virginia Tech, Verenigde Staten
Dr. L. Wang,	Vrije Universiteit Amsterdam
Prof. dr. M.M. de Weerdt,	Technische Universiteit Delft, reservelid

The work presented in this dissertation was partially supported by the Netherlands Organisation for Scientific Research (NWO), partly funded by the Dutch Ministry of Economic Affairs, through TTW Perspective program ZERO (P15-06) within Project P4.

Keywords: intermittent computing, battery-free, compilers, interpretation, embedded systems, energy harvesting, non-volatile memory

Printed by: Ipskamp Printing

Front & Back: Romello Kortbeek, https://romellokortbeek.nl/

ISBN 978-94-6473-147-7

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

To my mom Paula,
for always believing in me.

Contents

SUMMARY

The Internet of Things (IoT) is taking the world by storm, from smart lights to smart plant monitoring. This revolution is not only present in consumers' homes, but companies are also looking for more and more ways to monitor every aspect of their production process. This transition to ubiquitous monitoring is made possible by extremely low-power embedded devices, mostly powered by batteries. However, with the projected number of IoT devices reaching tens of billions within the next few years, this growth will directly contribute to a massive increase in battery waste, negatively impacting the environment. This increase in battery waste alone is already a well-founded reason to explore alternative energy sources. However, batteries come with more downsides. Many of these IoT devices will operate in hard-to-reach places (e.g., embedded into walls), and the sheer quantity in which these devices will be deployed will make it nearly impossible to replace batteries periodically without employing a costly dedicated workforce.

One alternative energy source for—often low-power—embedded devices comes from energy harvesting. Energy harvesting is collecting energy from the system's surroundings, such as energy from the sun, vibrations, wind, and radio waves. However, because of size constraints and the limited harvestable energy, there is often not enough energy to power these embedded systems constantly. Additionally, the energy from these sources is unreliable (e.g., when a cloud floats by, a solar cell's output will drop significantly or even stop outright). These limitations can be partially addressed by adding a small buffer capacitor to store a small amount of energy. Still, despite this capacitor, the power supply to an embedded device, and therefore its operation, will remain intermittent. To still perform meaningful computations, the device must continue (close to) where it left off after a power failure to avoid re-executing the same fragment of code over and over again (i.e., it should make forward progress). The approach of saving state to continue computation across power failures is called *intermittent computing*, and reducing the overhead introduced to support intermittent computing is the focus of this thesis.

More specifically, we set out to answer the question: *"How to reduce the overhead caused by automatically supporting intermittent computing?"*. To this end, this thesis explores different techniques to improve the performance of intermittent computing by lowering the introduced overhead. The overhead introduced by intermittent computing results from saving the system's state, i.e., a checkpoint, which is required to allow the program to continue executing where it left off. To achieve the aforementioned forward progress, the system must create checkpoints periodically or on demand (when the system reaches critical energy levels), which takes time and is the leading cause of overhead. A checkpoint contains all the information needed to continue execution, i.e., the system's volatile state, which typically includes the memory and the registers. The improvements developed in this thesis are achieved without requiring the programmer or application designer to manually change their existing code or adopt a *special-purpose* programming language to realize intermittently powered applications. Additionally, all the techniques presented in this work have the quality of being "incorruptible." That is, no matter when the power fails, even if it happens while storing the devices' state, the most recently completed checkpoint must still be able to be restored successfully.

This thesis is split into two parts. Part one focuses on intermittent computing support for systems with volatile main memory, which is the standard for nearly all embedded systems currently in use. When using **volatile** memory, its content must be saved somewhere in non-volatile memory so it can be restored when attempting to continue operation after a power failure. The benefit of this approach is that it can be directly applied to existing embedded architectures, which is what we will show first in a system that is designed for rapid prototyping of intermittently powered applications. We optimize the process of saving the volatile memory reducing the amount of volatile state that is copied to non-volatile memory during the creation of a checkpoint by only saving the changes in memory since the last checkpoint as patches. Then, during the restoration processes, these patches are combined to completely restore the memory state as it was during the more recently completed checkpoint. Optimizing the checkpoint time is beneficial because the system usually creates more checkpoints than it performs restorations. Checkpoints are often created when the system's power becomes dangerously low or sometimes periodically. Depending on the incoming harvested energy, the system can execute for longer. Therefore, the application executes opportunistically until the power runs out entirely, possibly creating many checkpoints until the power failure occurs.

Part two of this thesis focuses on systems with **non-volatile** main memory. With the main memory being non-volatile, power failures do not change the memory content. Hence, the memory content does not need to be saved before a power failure, significantly reducing the checkpointing cost. However, a new challenge appears. Because the processor state, i.e., the registers, is still volatile and saved in a checkpoint, we must keep this checkpoint state consistent with the memory state, which is no longer saved in a checkpoint. If the memory changes, the power fails, and the previous checkpoint is restored, then the system state is not identical to the state when the checkpoint was created.

Addressing the consistency problem requires checkpoints to be kept consistent with the memory, which can be done in multiple ways. One way is to undo all the changes made to the non-volatile main memory. Another way is to create idempotent sections concerning the main memory—i.e., regions of code that can be re-executed without side effects to the main memory. Finally, it is also possible to realize the creation of idempotent sections in hardware.

Undoing modifications made to the memory is the technique that will be demonstrated first, employing stack segmentation and undo logging. Logging is an effective technique when longer regions without checkpoints are preferred. In existing logging-based systems recursive functions have been problematic, but our system addresses these limitations. Additionally, our system introduces time-based decision-making to avoid processing stale information after a long power outage. For load-store architectures (i.e., architectures where memory is modified exclusively through dedicated load and store instructions), compiler transformations that create idempotent sections separated by checkpoints are better suited. This technique is the basis for the subsequent approach in this thesis, which reduces the number of needed checkpoints by rescheduling instructions. This method reduces the overhead without requiring a runtime logging component that introduces dynamic execution times or code regions. Lastly, we will consider a hardware-based approach that addresses the overhead caused by introducing many checkpoints and reduces the execution time and power consumption associated with using non-volatile main memory. The introduced data cache delays writing to the non-volatile memory for as long as possible, instead using the faster volatile cache, and can trigger checkpoints when detecting an idempotent region's end. The detection of an idempotent region's end is achieved by novel changes to the cache controller algorithm combined with additional bits in the cache lines.

In conclusion, the results presented in this thesis offer diverse solutions that can be used to reduce the overhead introduced to support incorruptible intermittent computing on various platforms automatically.

Samenvatting

Het Internet of Things (IoT) verovert de wereld stormenderhand, van slimme lampen tot toezicht houden in fabrieken. Deze revolutie is niet alleen aanwezig bij de consument thuis, ook bedrijven zoeken steeds meer manieren om elk aspect van hun productieproces digitaal bij te houden. Deze overgang naar alomtegenwoordige monitoring wordt mogelijk gemaakt door *embedded systems* met een extreem laag energyverbruik, meestal gevoed door batterijen. Echter, met het verwachte aantal IoT-apparaten dat de komende jaren tientallen miljarden zal bereiken, zal deze groei direct bijdragen aan een enorme toename van batterijafval, wat een negatieve invloed heeft op het milieu. Alleen al deze toename van batterijafval is een gegronde reden om alternatieve energiebronnen te verkennen. Batterijen hebben echter meer nadelen. Veel van deze IoT-apparaten zullen op moeilijk bereikbare plaatsen werken (bijv. weggewerkt in muren), en de enorme aantallen waarin deze apparaten zullen worden ingezet, maakt het bijna onmogelijk om batterijen periodiek te vervangen zonder dure toegewijde arbeidskrachten in dienst te nemen.

Een alternatieve energiebron voor *low-power embedded systems*, is het oogsten van energie, ofwel *energy harvesting*. Energy harvesting is het verzamelen van energie uit de omgeving van het systeem, zoals energie van de zon, trillingen, wind en radiogolven. Vanwege de beperkte omvang en de beperkte oogstbare energie is er echter vaak niet genoeg energie beschikbaar om deze embedded systemen constant van stroom te voorzien. Bovendien is de energie van deze bronnen onbetrouwbaar (als er bijvoorbeeld een wolk voorbij drijft, zal de energietoevoer van een zonnecel aanzienlijk dalen of zelfs helemaal stoppen). Deze beperkingen kunnen gedeeltelijk worden verholpen door een kleine buffercondensator toe te voegen om een kleine hoeveelheid energie op te slaan. Ondanks deze condensator blijft de stroomtoevoer naar het apparaat, en dus de werking ervan, intermittent. Om nog steeds zinvolle berekeningen uit te voeren, moet het apparaat (ongeveer) doorgaan waar het was gebleven na een stroomuitval om te voorkomen dat hetzelfde codefragment steeds opnieuw wordt uitgevoerd (d.w.z. het apparaat moet vooruitgang boeken). De techniek van het opslaan van de status om door te gaan met rekenen na stroomonderbrekingen wordt *intermittent computing* genoemd, en het verminderen van de extra rekenkosten die is geïntroduceerd om intermittent computing te verwezenlijken, is de focus van dit proefschrift.

Meer specifiek wilden we de volgende vraag beantwoorden: *"Hoe de kunnen we de vereiste extra rekenkosten verminderen die wordt veroorzaakt door het automatisch ondersteunen van intermittent computing?"*. De extra rekenkosten geassocieerd met intermittent computing zijn het gevolg van het opslaan van de systeemstatus, d.w.z. een controlepunt (*checkpoint*), dat nodig is om het programma verder te laten gaan met uitvoeren waar het was gebleven. Om de voorwaartse voortgang te bereiken, moet het systeem periodiek of op indien nodig (wanneer het systeem kritieke energieniveaus bereikt) checkpoints creëren, wat tijd kost en de belangrijkste oorzaak van de extra rekenkosten is. Een checkpoint bevat alle informatie die nodig is om de uitvoering voort te zetten,

d.w.z. de vluchtige toestand van het systeem, typisch het geheugen en de registers. De verbeteringen die in dit proefschrift worden ontwikkeld, worden bereikt zonder dat de programmeur of applicatieontwerper zijn bestaande code handmatig hoeft te wijzigen of een *special-purpose* programmeertaal moet gebruiken om intermittent aangedreven applicaties te realiseren. Bovendien zijn alle technieken die in dit proefschrift worden gepresenteerd *incorruptible*. Dat wil zeggen, ongeacht wanneer de stroom uitvalt, zelfs als dit gebeurt terwijl de status van de apparaten wordt opgeslagen, moet het meest recent voltooide controlepunt nog steeds met succes kunnen worden hersteld.

Dit proefschrift is opgesplitst in twee delen. Deel één richt zich op het mogelijk maken van intermittent computing voor systemen met vluchtig werkgeheugen (volatile main memory), de standaard voor bijna alle embedded systemen. Wanneer **vluchtig** geheugen wordt gebruikt, moet de inhoud ervan ergens in niet-vluchtige geheugen (non-volatile memory) worden opgeslagen, zodat deze kan worden hersteld om de werking voort te zetten na een stroomstoring. Het voordeel van deze techniek is dat deze direct kan worden toegepast op bestaande architecturen, wat we als eerste zullen laten zien in een systeem dat is gemaakt voor het snel ontwerpen van prototypes voor batterijloze applicaties. We optimaliseren het proces van het opslaan van het vluchtige geheugen en verkleinen de hoeveelheid vluchtige toestand die naar het niet-vluchtige geheugen wordt gekopieerd tijdens het maken van een checkpoint door alleen de wijzigingen in het geheugen sinds het laatste checkpoint op te slaan in de vorm van *patches*. Tijdens de herstelprocessen worden deze patches gecombineerd om de geheugenstatus volledig te herstellen zoals deze was tijdens het meest recent voltooide checkpoint. Het optimaliseren van de checkpoint-tijd is voordelig omdat het systeem meestal meer checkpoints aanmaakt dan herstelt. Checkpoints worden vaak gecreëerd wanneer de stroom van het systeem gevaarlijk laag wordt of soms periodiek. Afhankelijk van de binnenkomende geoogste energie kan het systeem langer doorgaan. Daarom wordt de applicatie opportunistisch uitgevoerd totdat de stroom volledig op is, waardoor er vaak vele checkpoints worden gecreëerd voordat de stroom uitvalt.

Deel twee van dit proefschrift richt zich op systemen met **niet-vluchtig** werkgeheugen (non-volatile main memory). Aangezien het werkgeheugen niet-vluchtig is, veranderen stroomonderbrekingen de geheugeninhoud niet. Daarom hoeft de geheugeninhoud niet te worden opgeslagen voordat de stroom uitvalt, waardoor de checkpointing-kosten aanzienlijk worden verlaagd. Er dient zich echter wel een nieuwe uitdaging aan. Omdat de processorstatus, d.w.z. de registers, nog steeds vluchtig is en wordt opgeslagen in een checkpoint, moeten we deze checkpoint-status consistent houden met de geheugenstatus, die niet langer wordt opgeslagen in een checkpoint. Als het geheugen verandert en vervolgens de stroom uitvalt en het vorige checkpoint wordt hersteld, is de systeemstatus niet identiek aan de status toen het checkpoint werd gemaakt.

Om het consistentieprobleem aan te pakken, moeten checkpoints consistent worden gehouden met het geheugen, wat op verschillende manieren kan worden gedaan. Eén manier is om alle wijzigingen in het niet-vluchtige werkgeheugen ongedaan te maken. Een andere manier is om idempotente secties te maken met betrekking tot het werkgeheugen, d.w.z. codegebieden kunnen opnieuw worden uitgevoerd zonder bijwerkingen in het werkgeheugen. Ten slotte is het ook mogelijk om de creatie van idempotente secties in hardware te realiseren.

Het ongedaan maken van wijzigingen in het geheugen is de techniek die als eerste zal worden gedemonstreerd, waarbij gebruik wordt gemaakt van *stack-segmentatie* en *undo logging*. Logging is een effectieve techniek wanneer langere regio's zonder checkpoints de voorkeur hebben. In bestaande logging gebaseerde systemen waren recursieve functies problematisch, maar ons systeem lost deze beperkingen op. Bovendien introduceert ons systeem tijd gebaseerde besluitvorming om te voorkomen dat verouderde informatie wordt verwerkt na een lange periode zonder energie. Voor load-store-architecturen (d.w.z. architecturen waarbij het geheugen uitsluitend wordt gewijzigd door speciale laad- en opslaginstructies), zijn compiler-transformaties die idempotente secties creëren, gescheiden door checkpoints, beter geschikt. Deze techniek is de basis voor de daaropvolgende aanpak in dit proefschrift, die het aantal benodigde checkpoints reduceert door instructies te herorganiseren. Deze methode vermindert de extra kosten van intermittent computing zonder dat er een runtime-logging component nodig is die variabele uitvoeringstijden van coderegio's introduceert. Ten slotte zullen we een op hardware gebaseerde benadering presenteren die de extra rekenkosten aanpakt die wordt veroorzaakt door de introductie van veel checkpoints en die de uitvoeringstijd en het stroomverbruik vermindert die gepaard gaan met het gebruik van niet-vluchtig werkgeheugen. De geïntroduceerde data-cache vertraagt het schrijven naar het niet-vluchtige geheugen zo lang mogelijk, en gebruikt in plaats daarvan de snellere vluchtige cache, en kan checkpoints activeren bij het detecteren van het einde van een idempotent gebied. De detectie van het einde van een idempotent gebied wordt bereikt door wijzigingen in het algoritme van de cachecontroller in combinatie met extra bits in de cacheregels.

Samenvattend, de resultaten die in dit proefschrift worden gepresenteerd diverse oplossingen die kunnen worden gebruikt om de benodigde extra rekenkosten te verminderen die wordt geïntroduceerd om *incorruptible intermittent computing* op verschillende platforms automatisch te ondersteunen.

1

Introduction

I love deadlines.
I love the whooshing noise they make as they go by.

Douglas Adams

Smart devices are taking over the world, with the Internet of Things (IoT) at the forefront [22, 212]. Omnipresent embedded systems have changed how people monitor [83] and interact with the world [150], detect diseases [140, 204], protect wildlife [10], monitor infrastructure [9], and play games [55]. Not only do people want to be more connected, but they also want every detail of their life accessible at their fingertips at all times [22]. Collecting and receiving information about every facet of our environment can significantly increase the productivity of industrial applications and add to the comfort level when applied for personal use. However, many embedded IoT devices must be deployed in the coming years to achieve this goal. With the number of IoT devices already having surpassed the number of humans living on earth and on track to reach 125 billion by 2030 [215, 150], this increase comes with considerable downsides.

IoT devices are increasingly embedded in hard-to-reach places and rely more and more on batteries instead of being constantly tethered to mains power. Replacing their batteries when such systems reach the end of their lifespan can be difficult, if not impossible. For example, it is very desirable to detect moisture in the walls of a building in a timely fashion, as prolonged exposure to moisture can cause irreparable damage. Therefore, it would be highly desirable to monitor the moisture content of the walls throughout the lifetime of the building. However, this is not currently viable due to the limited battery life of existing monitoring devices—compared to the lifespan of a building. To replace the monitoring device's battery, a technician would need to break open the wall to reach the device, destroying a part of the wall. To exacerbate the problem, many monitoring devices must be embedded throughout to observe the entire wall, all of which will eventually need their battery replaced.

In addition to being difficult and costly to replace [62], IoT's reliance on batteries raises concerns regarding its sustainability [247, 45]. The batteries that power these embedded

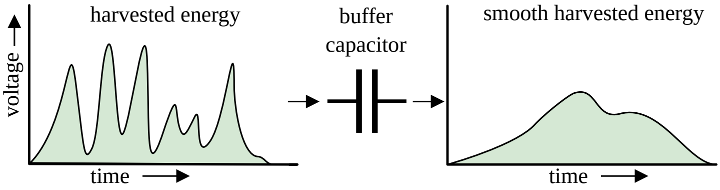


Figure 1.1: Harvested energy (left) and smooth harvested energy using a buffer capacitor (right).

devices need to be properly disposed [72] and can be a fire hazard [123].

These negative aspects of batteries stain the otherwise impressive potential that IoT can bring, which invites us to look further to develop battery-free solutions.

1.1. BATTERY-FREE COMPUTING

Just like clean alternative energy sources are taking over society's energy needs on a grand scale [232], so can energy harvesting supersede the use of batteries for embedded devices [8, 198, 74, 107, 108]. Harvested energy can come from many sources, such as radio waves, vibration, temperature differentials, wind, and the most classic example, solar.

The major downside of energy harvesting comes in the form of unreliability [201, 198]. Not only when the corresponding harvesting source is unavailable, e.g., at night when using solar, but also the stability of the outputted energy during harvesting. Fluctuations can occur very rapidly when harvesting energy from the environment. Additionally, the form-factor of low-power embedded devices are often constrained, and many energy harvesting solutions have a strong correlation between their size and the amount of energy they can harvest. This limitation means that the amount of energy harvested is often insufficient to power the system directly, making it impossible to simply replace the battery by an energy harvester. We can mitigate both of these limitations by introducing a buffer capacitor as shown in Figure 1.1. A buffer capacitor smooths out the voltage coming from the energy harvester. Additionally, a buffer capacitor can store a small amount of energy, allowing the system to continue operation for a short time after the energy harvesting has stopped.

With a buffer capacitor comes a trade-off. A larger capacitor can keep the system running longer without incoming energy. However, it also takes more time for the capacitor to reach a sufficient voltage threshold to boot the system. Exacerbating this trade-off are leakages in the capacitor, making it more challenging to store energy over an extended period. No matter the capacitor size, power failures—although less frequent but with longer charge times—are still the norm.

1.2. INTERMITTENT COMPUTING

The duration when the embedded device is powered and executes code is called the on-duration. This on-duration depends on the capacitor size, as mentioned earlier, but also the amount of incoming and outgoing energy. A downside of an energy-harvesting-based

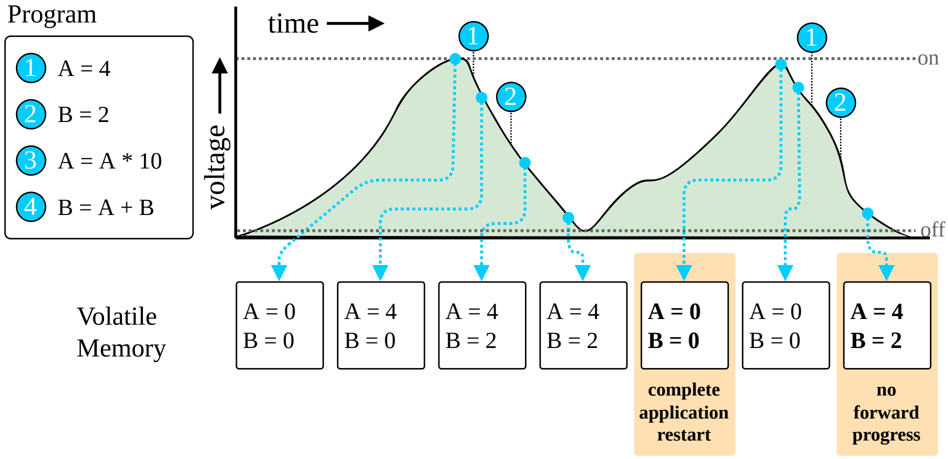


Figure 1.2: Program execution that lacks forward progress. Without a mechanism to save the program's state, the application restarts from the beginning after each power failure.

solution is that an on-duration is often insufficient to execute an embedded application, as many embedded programs infinitely execute a loop without a designated end. Even a single one of these iterations might not be able to complete within an on-duration. To complicate matters, successive iterations of the program's main loop might depend on information from previous iterations by means of the program's state. When the program keeps executing the same code, there is no forward progress, as demonstrated in Figure 1.2. Therefore when power failures occur, specially designed mechanisms are needed to continue the program's operation in a subsequent on-duration. This mode of operation is called **intermittent computing** [135, 85, 136, 235, 31], and it is the focus of this dissertation.

1.2.1. Storing and Restoring System State

To continue the execution of a program during the subsequent on-period, one must save the current execution state of the program's execution in non-volatile memory (e.g., flash) before the end of the on-period. This saved state is called a *checkpoint* and holds all the information needed to continue execution. The content of a checkpoint depends on the method used to continue execution, but in its simplest form, it contains the state of the volatile components in the system, shown in Figure 1.3. Volatile components in a microcontroller-based embedded system include the register state (which includes the continuation point of the program), the volatile memory state (which holds the variables), and potentially any peripherals configuration for interfaces such as UART, SPI, and I^{2}C (e.g., clock speed). During the subsequent on-period, the system restores the last successfully created checkpoint from the non-volatile memory, restoring all the volatile states from the non-volatile memory and allowing the system to continue execution as if nothing happened, as shown in Figure 1.4. In addition to the content of the checkpoint, it is important where the checkpoint occurs during the program's execution. The checkpoint

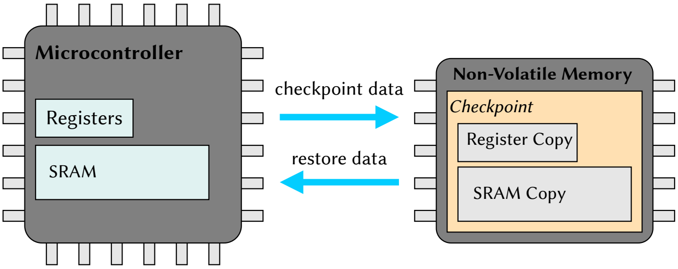


Figure 1.3: Microcontroller connected to non-volatile memory.

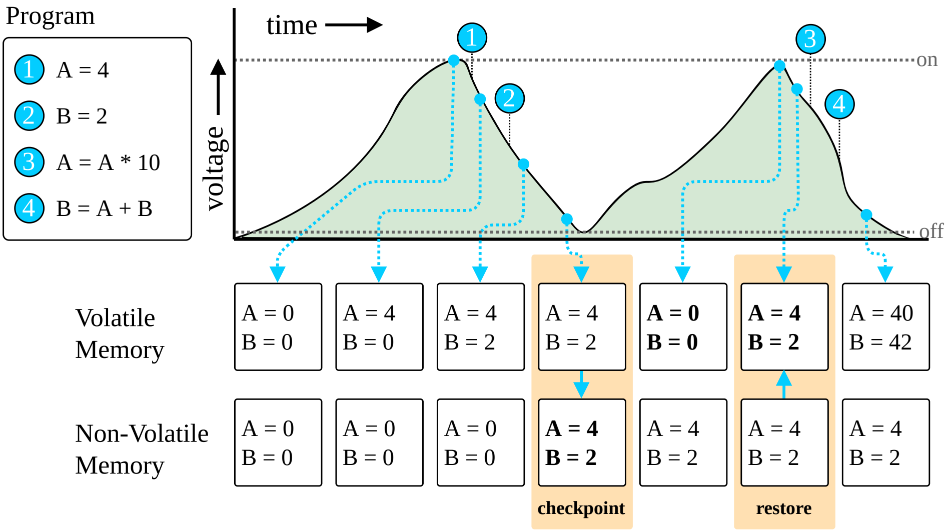


Figure 1.4: Execution of the same program as in Figure 1.2, but with forward progress. Because the program's state is stored in non-volatile memory before the power failure using a checkpoint, the execution can continue where it left off after the restoration.

location does not necessarily correlate to the point where the power fails. Instead, the checkpoint happens sometime before the power failure causing part of the code to be re-executed after a power failure.

Handling System Corruption

A consideration for intermittent computing support is whether the solution results in corruptible or incorruptible applications. When a system is corruptible, there is a chance that the system will reach a state where the content of the memory and the registers is not what it should be after a power failure. The root of this corruption comes from the uncertainty associated with the harvested energy. Some intermittent computing approaches regard checkpoints as atomic actions that must complete before a power failure [144, 30, 12]. This assumption of atomicity relies on predicting the impending power failure and leaves no room for error. If the new checkpoint overwrites (part) of the previous checkpoint and the power fails *while* creating the new checkpoint, the checkpoint will be corrupted. During the restoration of the checkpoint, part of the memory will reflect the state of the old checkpoint, and part will reflect the state it was during the failed checkpoint. This corruption can often be addressed by double buffering the checkpoint, i.e., not overwriting the previous checkpoint but the one before it. This way, the last fully completed checkpoint is always available. However, applying this technique is not always straightforward, as will be demonstrated in Part 1 of this thesis.

An incorruptible intermittent computing framework allows for power failures during any part of the program's execution, even during the creation or restoration of checkpoints. Hence, these systems can not rely on predictions to operate correctly. However, they can use predictions to improve performance if they can not impact the program's computational correctness. However, these incorruptible approaches can introduce execution time penalties that are significantly more substantial than their corruptible counterparts. Nonetheless, many systems will have incorruptibility as their core requirement.

Numerous battery-free applications will be deployed in hard-to-reach places and must operate for considerable amounts of time, longer than can be supported by batteries. For this reason, stable operation for long periods—ideally indefinitely—is at the core of intermittent computing. Restarting the application from scratch because of corruption can be detrimental in these scenarios. There can be cold start problems with the program. The application might expect a more extended period of power to perform its initial boot and calibration (as any subsequent boot would normally continue where the program left off). Additionally, starting the application from the beginning might overwrite any data collected thus far. Incorruptible systems are, therefore, the only viable approach for many applications, and users that require incorruptibility will have to accept the additional overhead it brings.

Manual Intermittent Computing Support

Intermittent computing support comes in two main varieties, automated and manual. The manual approach depends on programmers manually transforming existing code or designing new programs to allow for power failures. Splitting the program into different tasks is the most popular method to convert applications manually [250, 142, 47]. However, these tasks differ from those known in traditional computer science literature. Here, tasks must confine their execution time to finish within a set energy budget. The tasks are then connected to execute the complete program. Because these tasks have well-defined transition points and execute atomically within an on-period, a checkpoint only holds which task is the current task and the data it received from the previous task. Note that this means the tasks are idempotent and can not modify any global state, such as global variables. By saving such a small amount of data to the non-volatile memory—only the data communicated from one task to the next—manually created tasks have the potential to be highly efficient. However, the downside of manually transforming applications is the need to redesign and rewrite applications completely, requiring a lot of programmer intervention and eliminating the use of existing code in these new battery-free projects. Additionally, how to create tasks that have the correct length is challenging. As discussed earlier in Section 1.1, harvested energy is unreliable. Additionally, the power consumed by the embedded device is not constant and changes depending on the operation and the interaction with sensors and actuators. This uncertainty regarding the required task length can cause the developer to create smaller tasks to account for the variability of the harvestable energy, leading to considerably more task transitions than are necessary to achieve forward progress. Having many task transitions reduces the benefit of a task-based system, as every task transition saves its output in non-volatile memory; unnecessary task transitions lead to more time spent writing to and reading from non-volatile memory. Moreover, programming in a higher-level programming language makes it extremely challenging for a programmer to predict the actual energy cost of a section of code. Together these uncertainties regarding the desired task duration and the difficulty in correctly estimating a task's actual duration make it very challenging for programmers to design a task-based intermittent system. Previous works have tried to alleviate this dependence on manual work by automatically profiling these tasks or by profiling the application to better estimate available execution time between power failures [48]. Again, the problem with this approach is that the harvesting conditions can frequently change, and developers often want to reuse created tasks in other applications with different harvesting conditions. Therefore most task-based systems tend to target significantly shorter task durations than technically possible, reducing their efficiency. However, even with perfect code profiling, it is often still challenging to logically split regions of code into tasks. Take, for example, encryption or AI libraries. The code in these libraries tackles challenging topics that often include many calculations in loops that are only fully understood by experts. An application developer using a task-based model must manually go through these libraries to split the functionality into tasks if they exceed the allocated energy budget, which introduces a significant chance of introducing difficult to find bugs in the system.

Automated Intermittent Computing Support

Automated intermittent computing support converts a program written in an existing (often general purpose) programming language to run intermittently without or with minimal programmer intervention. This technique has the benefit that application designers and programmers do not have to rewrite existing code bases. Automated support for systems with volatile main memory frequently works by routinely interrupting the program's execution to save the current state in a checkpoint. When this happens depends on the technique used, but triggers can include the voltage level of the capacitor, the time elapsed since the last checkpoint, or a combination. Creating the checkpoint can either be corruptable [30, 12, 35], or incorruptable [203] as discussed earlier in Section 1.2.1. However, automatically transforming applications to work intermittently comes with performance penalties. A generic programming language does not allow the programmer to specify their intent when writing battery-free applications. Therefore automated methods can not perform specialized optimizations that depend on this information, such as which variables are still in use, which are explicitly defined when using a task-based mechanism. Additionally, generic code lacks clear transition points where a minimal amount of state needs to be saved, increasing the checkpointing cost.

Nevertheless, the ease of use and the adaptability of automated intermittent computing support often outweigh these limitations. Developers can directly use existing code and retarget applications to function in environments with different harvesting or energy usage characteristics. Additionally, automated transformations eliminate the error-prone interventions required by programmers to convert their applications manually.

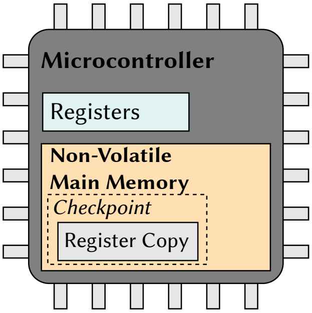


Figure 1.5: Microcontroller with onboard non-volatile main memory.

1.2.2. Retaining the Main Memory Across Power Failures

For decades the non-volatile memory available in embedded systems was either FLASH or EEPROM based. Fortunately, new types of non-volatile random-access memory have entered the market, such as Magnetoresistive Random Access Memory (MRAM), Ferroelectric Random Access Memory (FRAM), and Resistive Random Access Memory (ReRAM). These memory types are similar to the Static Random Access Memory (SRAM) traditionally used as the main memory for embedded devices, but with the addition of being non-volatile. What differentiates these memory architectures from architectures like FLASH is that they are byte-addressable and low-power and are designed in such a way that they can be used as drop-in replacements for traditional SRAM memory [71]. The non-volatility of these new memory architectures, in combination with their likeness to traditional SRAM, is key to unlocking an alternative method to achieve intermittent computing, using the non-volatile memory as the system's *main memory* (Figure 1.5).

The systems discussed in Section 1.2.1 use standard volatile main memory. Data stored in such memory does not always need to be saved all at once (as will be demonstrated in Chapter 3). However, the volatile memory does need to be restored fully to resume operation. Removing the memory state from the checkpoint by using non-volatile main memory significantly reduces its size, which reduces the time needed to create the checkpoint. Having shorter checkpoints and restoration times allows for shorter on-durations, allowing for smaller capacitors and energy harvesters, reducing the total size of the system without compromising its functionality.

Additional Corruption Causes

Not including the non-volatile main memory in the checkpoint reduces the cost of creating a checkpoint, but it introduces additional challenges that need to be overcome to provide an incorruptible solution. As is the case with corruption caused in volatile systems (Section 1.2.1), in systems with non-volatile main memory, corruption is caused by failing to perform a checkpoint before a power failure. However, when using non-volatile main memory, the memory is not stored within a checkpoint. Nonetheless, the memory is continuously modified during execution, so restoring the checkpoint—containing only the registers—is not enough to restore the system correctly. The memory state will likely not be the same as when the checkpoint was created, causing the checkpoint (containing the state of the registers) and the main memory to be desynchronized, leading to corruption when program execution continues, caused by re-execution using altered memory compared to the previous execution. Therefore, we need to take more advanced approaches to avoid desynchronization, either by undoing modification to the memory before restoring a checkpoint or avoiding scenarios that can cause desynchronization altogether.

MANUAL TRANSFORMATION

Similar to Section 1.2.1, manual transformation attempts to segment the program into sections that can be executed atomically during an on-duration. Because these are idempotent sections, depending only on the (read-only) input and producing an output that can be read by the subsequent task, task-based solutions avoid the desynchronization problem by double-buffering these task transitions. Hence, no additional support is needed to utilize these techniques using non-volatile main memory. However, there are also no real benefits to using non-volatile main memory in a fully task-based system, as the data within the task never needs to be saved and restored. Only the input and output reside in non-volatile memory, as was already the case for systems with volatile main memory.

AUTOMATED SUPPORT

Automated support becomes significantly more complicated when non-volatile main memory is used. For volatile systems, the automatic checkpoint support often comes in the form of either time or capacitor voltage-level triggered checkpoints that then store the memory and register content in non-volatile memory. Because the memory is stored in the checkpoint, double buffering the checkpoint is sufficient to avoid corruption. For systems with non-volatile memory, this is not the case. When a checkpoint is created, the memory is not included in the checkpoint. One approach is to use just-in-time checkpoints, which use prediction or capacitor voltage level triggers to signal the creation of a checkpoint. However, this technique is susceptible to corruption, as discussed in Section 1.2.1. Hence, this technique is undesirable for many applications.

To allow for incorruptible intermittent computing, the automated approach must consider the fact that the memory is not included in the checkpoint and is continuously modified during the program's execution. Being able to completely and correctly restore the system state introduce significantly more complexity at runtime, compile-time, or both.

1.3. PROBLEM STATEMENT

We introduced intermittent computing as an alternative for battery-powered embedded systems in Section 1.2. However, moving from a battery—a constant and reliable energy source—to unreliable harvested energy comes with memory synchronization problems that are hard to overcome in an automated manner and introduce a significant overhead. Therefore, we formulate the **main research question** of this dissertation as follows:

> How to reduce the overhead caused by automatically supporting incorruptible intermittent computing?

First, we address this question for embedded systems with **volatile main memory**, and we formulate following sub-questions:

Sub-Question ①: How to support rapid intermittent computing prototyping on an existing system with volatile main memory using checkpoints?

Sub-Question ②: How to reduce the overhead caused by checkpointing for embedded systems with volatile main memory by reducing the checkpoint size?

Second, we address the main research question when considering embedded systems with **non-volatile main memory**. As discussed in Section 1.2.2, introducing non-volatile main memory has benefits but presents challenges regarding memory and checkpoint synchronization. For such systems, we formulate the following sub-questions:

Sub-Question ③: How to reduce the overhead caused by keeping the checkpoint synchronized with non-volatile main memory with the help of software?

Sub-Question ④: How to reduce the overhead caused by keeping the checkpoint synchronized with non-volatile main memory with the help of hardware?

1.4. CONTRIBUTIONS AND OUTLINE

In this thesis, we provide multiple solutions to support intermittent computing on battery-free embedded devices. The primary goal is to start with regular programs targeted at embedded systems, enabling them to execute intermittently without programmer intervention. Additionally, we want to enable incorruptibile intermittent computing without needing additional information regarding the proximity to a power failure, i.e., no matter when the power fails, the system must be able to continue execution from the most recently successfully created checkpoint.

This dissertation consists of two parts, as depicted in Figure 1.6, addressing each of the four sub-questions. The first part focuses on solutions for systems with volatile main memory. The second part focuses on solutions for systems with non-volatile main memory. Let us now outline the contributions of the two chapters in part one.

► **Enabling Battery-free Prototyping**—Chapter 2. In this chapter, we convert an existing battery-powered embedded sensor platform to operate intermittently using harvested power. The main goal is to identify the challenges of making intermittent computing without programmer intervention possible on an existing platform to allow for quick prototyping of battery-free applications using Python. To this end, we take an off-the-shelf embedded platform, the Adafruit Metro M0 Express, and add an extension hardware board that harvests energy from the environment. We chose the Metro M0 Express because it supports an embedded version of Python. By using interpretation to execute programs instead of directly running on the hardware, we show that we can support intermittent computing without requiring any modifications to the Python application code. Interpretation, although considerably slower than native execution, allows us to modify the underlying execution runtime to hide the intermittent behavior from the programmer without any alterations to the Python code. This can be done because all actions the program performs must go through the Python interpreter, allowing us to track all access to data and peripherals and undertake appropriate actions to ensure the

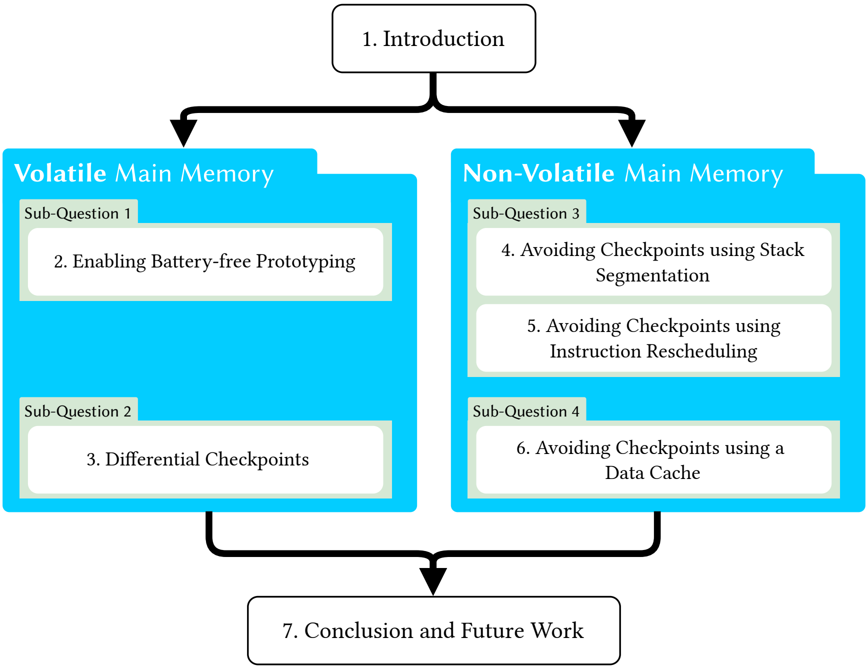

1. Introduction
Volatile Main Memory
Sub-Question 1
2. Enabling Battery-free Prototyping
Sub-Question 2
3. Differential Checkpoints
Non-Volatile Main Memory
Sub-Question 3
4. Avoiding Checkpoints using Stack Segmentation
5. Avoiding Checkpoints using Instruction Rescheduling
Sub-Question 4
6. Avoiding Checkpoints using a Data Cache
7. Conclusion and Future Work

Figure 1.6: Visual representation of the thesis outline.

system operates as intended amidst power failures. Because interpretation is already quite slow, adding intermittent computing support has a relatively small impact on the system's overall performance. We demonstrate that our solution correctly handles the use of peripherals in addition to computational programs. We also show multiple applications that run intermittently on harvested energy without modifying the Python code.

► **Differential Checkpoints**—Chapter 3. Copying all the volatile to non-volatile memory during a checkpoint is time-consuming and the most significant cause of overhead when supporting automatic intermittent computing on systems with volatile main memory, as is demonstrated in Chapter 2. However, the focus of Chapter 2 is on quick prototyping rather than on performance, and the relatively low performance of the Python interpreter itself lessens the impact of the checkpointing overhead. Nonetheless, the high cost of saving all the volatile memory is not always acceptable. In this chapter, we introduce a novel, patch-based, checkpointing framework called MPatch. MPatch targets more performant scenarios than the Python interpreter in Chapter 2. MPatch reduces the amount of memory stored during a checkpoint by creating differential checkpoints, i.e., by checkpointing only the memory that was changed since the last checkpoint. What makes MPatch unique is that it does this while remaining incorruptible. Traditional differential checkpoint approaches rely on prediction to guarantee that they correctly copy the changed memory into an existing checkpoint. However, if the power fails while overwriting the existing checkpoint, the system is left in an unrecoverable state and has to restart completely. MPatch takes a different approach by creating patches with the changed memory that are applied during the restoration procedure to reconstruct the memory. These patches are stored in different parts of the non-volatile memory and do not change the content of the previous checkpoint or patch. This way, we can incorruptibly create differential checkpoints without requiring the data to be double-buffered, significantly reducing the checkpointing overhead.

The second part of this dissertation concerns solutions using non-volatile main memory. As outlined in Section 1.2.2, numerous benefits are related to using non-volatile memory, but they come at a cost. The chapters in this part of the thesis address these limitations in more detail and provide three different approaches to reduce the overhead associated with non-volatile main memory, all with different considerations and trade-offs.

▶ **Avoiding Checkpoints Using Stack Segmentation**—Chapter 4. When using non-volatile main memory, the overhead introduced by supporting intermittent computing comes from keeping the checkpoint synchronized with the non-volatile memory state. To this end, the system must revert all modifications to the non-volatile memory after a checkpoint to avoid incorrect re-execution. Existing architectures solve this by creating idempotent sections by inserting register checkpoints between memory accesses, resulting in an enormous number of checkpoints. Or they employ logging to track all memory accesses after a checkpoint to restore the memory to the state when the most recent checkpoint occurred after a power failure. This chapter introduces a hybrid approach, where a part of the stack—a stack segment—is included in the checkpoint. An active stack segment allows the system to avoid logging memory changes to this part of the stack, significantly reducing the overhead. This method does increase the size of a checkpoint, as it now includes a portion of the main memory; however, we show that the lower memory logging cost offsets this, resulting in a lower total overhead.

▶ **Avoiding Checkpoints Using Instruction Rescheduling**—Chapter 5. In this chapter, we consider an alternative software-only approach to reduce the overhead of supporting intermittent computing on systems with non-volatile main memory. It includes several novel compiler transformations that reschedule memory operations to reduce the number of checkpoints required to avoid potential re-execution errors. The benefit of this approach compared to the one presented in Chapter 4 is that it does not rely on runtime logging, which makes the execution time of regions of code constant instead of it depending on how full the undo-log buffer is and whether a checkpoint is forced due to the buffer reaching its maximum capacity. However, this approach works best on load-store architectures without instructions that can read and write memory in one instruction (a register–memory architecture), as these need to be split up into multiple instructions—as reading and writing to a memory location without a checkpoint inbetween can lead to invalid re-execution—which would severely impact the performance. Therefore this chapter targets an ARM processor with such a load-store architecture instead of the MSP processor used in the previous chapter that utilizes a register-memory architecture. Our main contribution lies in a loop transformation that unrolls the inner loops in a program and then reschedules writes across loop iterations. We show that this method drastically reduces the number of required checkpoints, significantly lowering the introduced overhead.

► **Avoiding Checkpoints Using a Data Cache**—Chapter 6. In this chapter, we take a different approach to reduce the overhead introduced by intermittent computing for systems with non-volatile main memory. Instead of taking a software approach, combined with compiler modifications to track the memory and place checkpoints, we take a hardware approach where we modify the data cache to detect memory sequences that can lead to errors during re-execution and trigger a register checkpoint. Data caches are not heavily researched in the intermittent computing domain because they are typically not utilized in low-power embedded systems. Data caches are rarely used because the main memory in traditional embedded systems often consists of on-chip SRAM, which is the same as would be used in a data cache; therefore, a data cache usually has minimal benefits. However, for intermittent computing with non-volatile main memory, the non-volatile memory in question is slower than the SRAM that makes up the cache. Therefore, a data cache is a great addition to a dedicated processor designed specifically for intermittent computing. In existing solutions, whenever data caches are used in intermittent systems, they are introduced as a black box combined with existing hardware or software approaches. In this chapter, we introduce NACHO, a data cache solution that is aware of intermittent computing and is not only used as a data cache but also as the detection mechanism that protects against re-execution errors by utilizing novel changes to the cache lines—in the form of two additional bits—and cache controller to use these extra bits to trigger the creation of checkpoints. NACHO eliminates the need for additional software and hardware support to avoid corruption during execution while significantly increasing the application's performance compared to the state-of-the-art.

Part One: Volatile Main Memory

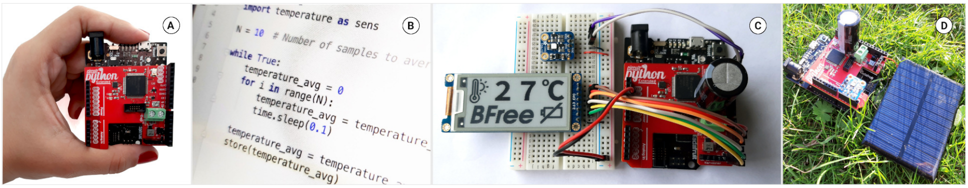
In the first part of the thesis, we will target embedded devices for intermittent computing with volatile main memory. Volatile main memory, e.g., SRAM, is the most common form of embedded RAM and is therefore present in most off-the-shelf microcontroller devices, making the techniques introduced directly implementable in current IoT systems. However, the volatile nature of the main memory means its content is lost on a power failure. Therefore the memory content needs to be saved and restored to support intermittent computing. Saving all the memory is achievable, as embedded devices are often very constrained, having mere kilobytes of memory at their disposal. However, storing and retrieving all this memory to and from a non-volatile medium such as MRAM, FRAM, or Flash still comes at a high cost.

2

Enabling Battery-free Prototyping

In this chapter, we identify the challenges and requirements for converting an existing embedded platform to a battery-free system. We target an off-the-shelf hobby IoT platform that uses Python interpretation to perform actions such as reading sensors, processing data, and controlling actuators. We transform this platform to work intermittently by introducing a special hardware add-on that harvests energy and contains non-volatile memory to which data can be saved. We modify the Python interpreter to handle power failures by saving and restoring the state of the natively executing code, the interpreter, and the internal peripherals to and from non-volatile memory. Together, these modifications allow unmodified embedded Python programs to be executed intermittently using harvested energy.

This chapter is based on the following publication:
Vito Kortbeek, Abu Bakar, Stefany Cruz, Kasım Sinan Yıldırım, Przemysław Pawełczak, and Josiah Hester. *BFree: Enabling Battery-free Sensor Prototyping with Python.* Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT), volume 4, issue 4, pages 1–39, December 2020. https://doi.org/10.1145/3432191.
The accompanying archive containing all the software and data can be found at https://doi.org/10.5281/zenodo.7714111.



```
import temperature as sens
N = 10  # Number of samples to aver
while True:
    temperature_avg = 0
    for i in range(N):
        temperature_avg = temperature_
        time.sleep(0.1)
    temperature_avg = temperature
    store(temperature_avg)
```

Figure 2.1: BFree shield connected to an embedded hobbyist-grade embedded computer (Adafruit Metro M0 board [5]) enables developing battery-free applications powered by ambient energy **(A)**. Makers and technology hobbyists can for the first time program a battery-free platform in Python **(B)**, and can easily connect sensors to BFree for fast prototyping **(C)**. BFree can be deployed indefinitely, supporting application domains where untethered and long-term sensing is desired **(D)**.

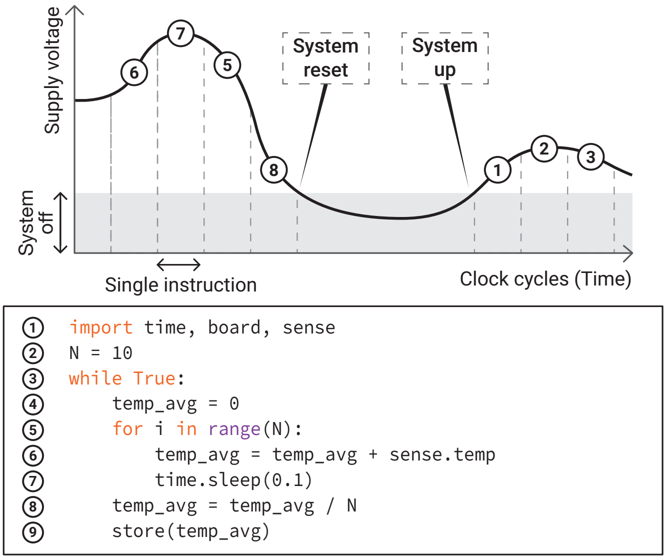
2.1. INTRODUCTION

The maker and hobbyist movement has brought computing and programming to the masses [132, 162, 205]. Hobbyists can now build functional embedded computing systems, such as temperature sensors, motion-controlled actuators, interactive lighting systems, or simple robotic platforms, that used to be the purview of experts only. Building with Arduino [20], and more recent platforms such as CircuitPython [6], MBed [25], Micro:bit [63], or Microsoft MakeCode [160], has empowered novice developers and makers to think beyond the traditional computing constraints with a desktop or laptop and what can be accomplished with computing everywhere. All these platforms allow for quick prototyping and programming of complex embedded systems, reducing the time to demo.

Concurrently, concerns about the sustainability of computing within wireless sensor networks [193, 115], and more generally, the Internet of Things [215] have arisen. Battery-free devices allow very unique applications recently demonstrated and deployed in consumer-grade products like phones [222], in space [49], in implantables [204, 140], in machine learning [125], in handheld gaming consoles [55], and even underwater [102]. Despite this emerging technology trend, the majority of hobbyist and maker projects are still plugged into the wall or laptop, or battery-powered, meaning that the hobbyist programmer is *unprepared for (or not even aware of) a battery-free future* [190, 85] and does not have the ability to create novel and exciting untethered applications.

Currently, developing battery-free applications powered by ambient energy only is in the realm of experts, as a combination of system-level difficulties that span hardware, software, and design make it difficult to work with these devices. The de facto programming language for programming battery-free systems is the C programming language. Developers must program in C and use specialized software (i.e., runtimes) and hardware (i.e., non-volatile memory) to allow a regular program to run correctly and consistently on intermittent power. However, using these runtimes requires in-depth knowledge of tools like LLVM, GCC, Make, and custom APIs—again, things that are standard for experts but *arcane for the novice.*

If we are to enable hobbyists to program these devices and participate in the future of sustainable computing, we must streamline this process. We foresee that Python is the strongest programming language candidate to enable this. It is one of the fastest-



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

Figure 2.2: An illustrative power supply trace of a battery-free device executing a simple Python program. The program never reaches line 9, where it stores the result. With energy harvesting devices, power failures can occur at any time.

growing and most popular languages currently ranked as the number one language in 2020 based on IEEE Spectrum multi-metric multi-source study [40]. It is the most searched language on online language tutorials [39] and one of the top three languages measured through StackOverflow and GitHub data mining and developer surveys [175]. Python is an interpreted language popular with beginners, hobbyists, and advanced users for myriad applications from machine learning to embedded programming [227]. Its simplicity and ubiquity have already made it an ideal language for electronic hobbyists and makers with the advent of MicroPython and AdaFruit's CircuitPython [6].

However, using Python to develop battery-free applications is not trivial—an illustration of a battery-free device executing a simple Python code is shown in Figure 2.2. Power failures cause the program to restart from the top of the program, keeping the entire program (or an iteration of an infinite loop) from finishing, wasting time and resources, and essentially making the device useless. These power failures also change the peripherals state (such as a connected sensor or radio) and cause delays (as the time it takes the battery-free device to restart) to be too long and too energy-intensive for the device to function on harvested energy only.

In this chapter, we propose an end-to-end system, BFree, shown in Figure 2.1 that seeks to *fill the systems gap preventing hobbyists and makers from participating in the battery-free energy-harvesting future of ubiquitous computing*. We tackle the technical hurdles of implementing a power failure resilient Python interpreter on low power and ultra-constrained embedded systems. In particular, we are concerned with making the tools for resilient, useful, in-the-wild computation to be build-able (make-able) by a typical person. We try to integrate as closely as possible in the existing workflows of Python developers and hobbyist maker communities; for this reason, we adapt the most popular Python runtime for embedded systems—MicroPython [69]. More specifically we adapt its fork, CircuitPython [6], targeting ease of learning and use for hobbyist microcontroller users. Because CircuitPython is actively developed by Adafruit (one of the major providers of hardware to makers and hobbyists [239]), we start from the base CircuitPython runtime and implement numerous additions that enable CircuitPython to function effectively without a battery or tethered to a power outlet. These additions are invisible to the programmer; they do not change the existing CircuitPython workflow taught by Adafruit[1]. In our system, makers and hobbyists can develop untethered, battery-free computers that can do interesting tasks like read a humidity sensor, transmit information through LoRa radios, take a thermal image snapshot, or recognize human movements.

Contributions: We make various innovations across hardware and software to make this happen, rewriting and extending the CircuitPython interpreter (written in C) to checkpoint and restore state automatically such that Python programs can run despite frequent power failures on a very constrained computing device (an ARM Cortex-M0 [24]). This was done by reworking the initialization of CircuitPython to enable fast reboots, supporting general peripherals like SPI and I2C for interesting applications using sensors, and rewriting system libraries to support restarting. We build custom hardware based on Adafruit's CircuitPython device lineup that functions as a shield and allows for a plug-and-play way to add energy harvesting and battery-free operation to the standard CircuitPython microcontroller board [5]. The shield and new runtime work together to ensure power failure resilient operation—enabling, for the first time, battery-free computation for hobbyists, makers, and early-stage embedded programmers.

In summary, the specific contributions of this chapter are:

a) Introducing power failure resilience to an embedded Python runtime for ARM microcontrollers;

b) The integration of re-initialization of active peripherals;

c) The design and development of a hardware module that enables off-the-shelf maker platforms to be used for battery-free development and deployment;

d) Integrating both of these into an existing workflow supported by the hobbyist and makers community.

BFree is the first general-purpose platform for battery-free, energy harvesting devices, that runs Python. We release all code and hardware designs as open source to the makes and hobbyists community via [230].

2.2. Developing a Battery-free Internet of Things

As discussed in Chapter 1, battery-powered computing's negative impacts have become more apparent. Unfortunately, building battery-free sytems is a complex task, not easily

[1] The authors are not affiliated with, or funded by, Adafruit.

done by a *hobbyist*[2], and not lending itself to the maker movement [162, 205]. If these devices are the future, and everyone will be a programmer, then these devices must be made accessible to the novice, hobbyist, and maker movement.

2.2.1. Maker Platforms

The first steps towards this vision were enabled by platforms like Arduino [20], MicroPython [69], and CircuitPython [6]. CircuitPython is a Python interpreter written in C that can run on a microcontroller like an ARM Cortex-M0 [24]. Python programs are written on the desktop computer, sent over USB to the microcontroller (MCU), compiled into bytecode on the MCU, then executed. CircuitPython wraps common hardware functions like digital I/O, analog, and serial protocols into libraries that are accessible to the programmer. Other libraries that support things like cameras and radios can be created with Python. CircuitPython has a large community of users, hundreds of libraries for sensors and radios, stable releases, and a large number of hardware devices to use. We chose to build off CircuitPython because the Python programming language has seen a surge in popularity [40, 175], especially with low skill or hobby programmers, and to maintain access to this motivated existing user base. CircuitPython currently *cannot* support battery-free energy harvesting applications because of a multitude of reasons centered around the frequent power failures caused by requiring the device to live off energy harvesting only and not have a battery.

2.2.2. Challenges of Battery-free Programming

When a battery is removed from a microcontroller such as an Arduino Uno [21] or Adafruit Metro M0 [5], and the device opportunistically uses energy harvested from the ambient (like solar radiation), power failures become the norm. Once started, the device begins a race to get things done before a power failure. Once the power failure occurs, all the device's volatile state is lost (for example, memory content or register values). Then the process begins again.

Without some type of system-level handling of these power failures, programs could easily get stuck repeating old tasks and never completing all the tasks required (see again Figure 2.2), never getting to the end of the program. Recent work has explored instrumenting C programs with checkpoints [136, 203] or partitioning C programs into task graphs [250, 142] to enable easier checkpoint and restore cycles so that forward progress can be maintained. These techniques are useful, but only to the expert. In this work, we rethink the problems in battery-free, intermittent computing in terms of an interpreted and easy to use language—Python—aiming at the vision depicted in Figure 2.3. Unlike C/C++, which is compiled directly to machine code with each instruction executed by the CPU, interpreted languages have a runtime system that interprets the bytecode of the language into machine code. Allowing the runtime to handle high-level programming models and concepts, and even compile run code on the fly without compilation. Our choice to focus on an interpreted language instead of already very well explored C, stems not only from the fact that Python is probably the

[2] By *hobbyist* we refer to people that have at least minimum experience in any of classical (popular) programming languages such as C, Java or Python; not hobbyist in a sense of being exposed only to extremely simplified (almost kids-accessible) languages such as Scratch [161].

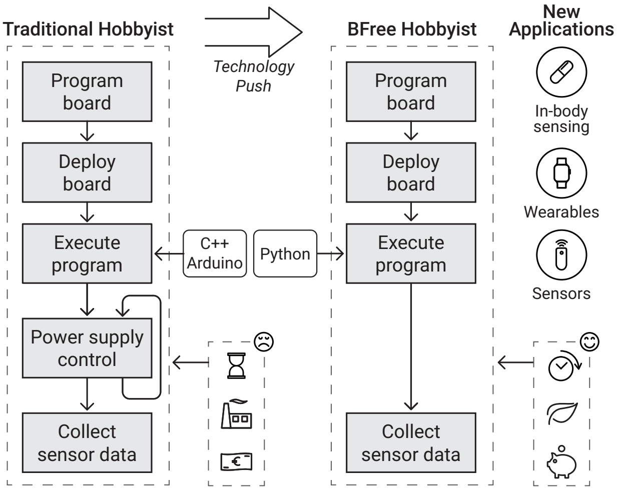


Figure 2.3: The vision of computing we hope to enable by allowing hobbyists to build and then deploy battery-free devices easily and quickly for years: (i) reducing time in development (*easy-to-use Python versus C/C++*), (ii) reducing carbon footprint and saving money (*elimination of batteries*) and (iii) making program execution understandable despite frequent power failures (*guaranteed forward progress despite interrupts*).

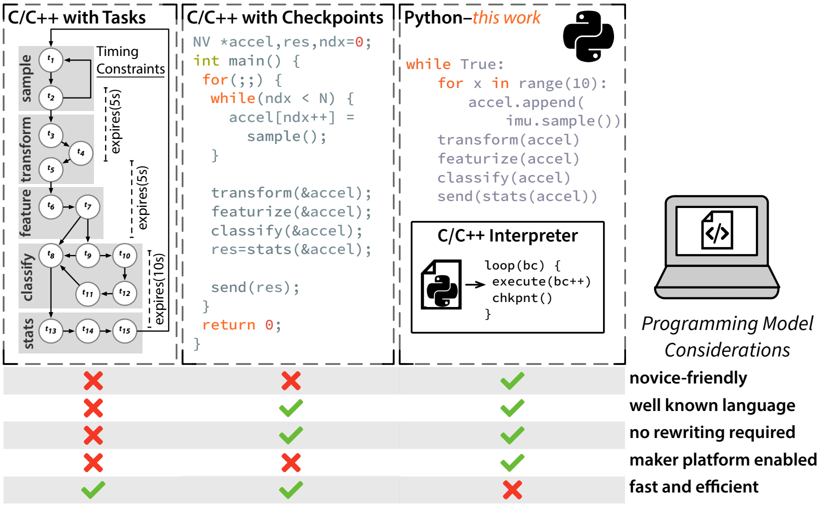
most popular programming language at the time of writing this article [39], but also due to its code compactness, fast extensibility, easier comprehension, and its simple and forgiving syntax. Quoting from a study published in 2000 [196]: “Designing and writing the program in Perl, *Python*, Rexx, or Tcl takes no more than half as much time as writing it in *C, C++*, or Java-and the resulting program is only half as long.” Naturally, both C and Python have features that are represented better in one of these languages [153], but the fact that Python has *not yet* been used in the context of intermittent execution called for action.

Unique challenges come from trying to build and program battery-free and energy harvesting “things” that were addressed for the C language, and must now be explored and overcome in any interpreted language, such as the Python runtime in this work.

Power Failures: Energy needs usually outweigh the energy availability, meaning that even with consistent energy harvesting, power failures are the norm as the supply becomes depleted. Programs can be interrupted mid-execution at any code line, which damages program consistency and frustrates the programmer.

Long Reboot Time: Initializing a system, especially a sophisticated runtime, takes time and energy. Any upfront energy cost for rebooting takes away from valuable user application time and, in some cases, can cause a power failure before the completed reboot. This long reboot happens because the expectation is that these systems will almost always have continuous power (via a battery or USB plug) and rarely, if ever, need to reboot, so the reboot does not need to be optimized for speed.

Peripherals: Interfaces such as SPI and I2C have their own state stored in volatile registers.



```
NV *accel,res,ndx=0;
int main() {
  for(;;) {
   while(ndx < N) {
     accel[ndx++] =
       sample();
   }

   transform(&accel);
   featurize(&accel);
   classify(&accel);
   res=stats(&accel);

   send(res);
  }
 return 0;
}
```

```
while True:
    for x in range(10):
        accel.append(
            imu.sample())
    transform(accel)
    featurize(accel)
    classify(accel)
    send(stats(accel))
```

```
loop(bc) {
  execute(bc++)
  chkpnt()
}
```

C/C++ with Tasks	C/C++ with Checkpoints	Python–*this work*	*Programming Model Considerations*
✗	✗	✓	**novice-friendly**
✗	✓	✓	**well known language**
✗	✓	✓	**no rewriting required**
✗	✗	✓	**maker platform enabled**
✓	✓	✗	**fast and efficient**

Figure 2.4: Current and future programming models for batteryless systems. Two left-most dashed rectangles shown the state-of-the-art, expert-focused programming models using either *task graphs* where each task is a focused C/C++ fragment, to define a program (as in InK [250]), or automatically (with some caveats) checkpointing a C/C++ program (as in TICS [119]). These models run bare-metal, as in the CPU directly executes the machine code. A right-most dashed rectangle shows a novice-focused new model for batteryless systems programming, using Python. While Python requires an interpreter which executes bytecode, and is therefore much much slower when executing on an MCU compared to other approaches, the benefit of a familiar, forgiving (no C-pointers) programming model combined with automatic handling of power failures is far more important.

These interfaces connect to external peripherals (like a temperature sensor or a radio), which also have a volatile state. Both the interface and the peripheral will lose their configured settings when a power failure occurs. Developers have to explicitly handle the loss of the state of both the interface and the peripheral.

This list of challenges is not exhaustive (please refer to a broad discussion of these challenges in [135]). Indeed the area of intermittent computing for performance-driven embedded systems is an active research area. However, the fundamental limits listed above must be overcome once again, rethought, and re-imagined to enable an interpreted and easy to use language for hobbyist programmers.

2.2.3. State-of-the-Art in *Programming* for Batteryless Systems

With the multitude of challenges for batteryless development, two prevailing programming models have arisen, *task*- and *checkpoint*-based, both focusing on the C language, which we compare and contrast here. Figure 2.4 shows a visual comparison of these two programming models against Python, and each model's pros and cons.

Like in any area, the best programming language to use depends on the context, user skill, and available tools. As discussed in Chaper 1, task-graph-based models like Mayfly [86], Alpaca [142], and InK [250], require a list of tasks that are strung together in a task-graph. The graph specifies the order and branching/control of execution. Tasks themselves must be atomic and idempotent (i.e., have no side effect, so they must bring up and tear down peripherals like a radio if used in the task). The runtime system that executes tasks commits the results after task completion to non-volatile memory, preserving forward progress. Most task-graph-based languages allow annotations on the edges to define time constraints and data handling. Together these task-based programming models are by far the fastest and most energy-efficient methods for intermittent computing, but they require significant attention and expert rewriting to implement correctly.

The other major programming model is to take regular C code and automatically instrument it with checkpoints at compile time [119, 143, 235] or runtime [30]. The key research challenge here is to reduce the size of the checkpoint, such that the execution time becomes feasible and usable. C/C++ is not known for being novice-friendly, with its bare metal execution, use of pointers, and requirement (especially in the embedded context) for the programmer to have in-depth familiarity with computer organization concepts like memory, addressing, and types, to be useful.

The third option, newly presented by this work, is to use Python, as shown in Figure 2.4, which compares all three programming models. This option is radically different as it suggests an entirely new way to write software for batteryless systems, using a programming paradigm (interpreted)—*never used on batteryless devices before.* Python does not require the programmer to port existing code to a niche programming model like InK (which is task-based) and does not require developers to know C/C++ and have intimate knowledge of computer organization—a non-starter for many programmers with non-traditional introductions to computing. Python is also directly in line with exciting, novice focused maker platforms that already have large communities and hardware. Notably, an interpreted language like Python will be slower, as an intermediate step exists between the Python code and the machine code, and is likely the primary reason up untill now Python was not used on batteryless, intermittently-powered devices. However, we posit that the flexibility and ease of using the language are worth that performance hit in many contexts. Finally, Python (and, more specifically, the interpreter) provides a ready mechanism for seamless—and invisibly to the programmer—checkpointing for power failure resilience.

2.2.4. State-of-the-Art in *Building* for Batteryless Systems

Researchers in intermittent computing have turned to novel hardware to make programming and building batteryless devices easier. C-based task graphs and checkpointed programming models are used on a multitude of hardware that all have one common characteristic, TI MSP430 FR series microcontrollers [96], which have built in FRAM, a byte addressable, non-volatile memory, that makes checkpointing state quick and cheap. Platforms like WISP [209], Flicker [84], Capybara [49] or Botoks [54] all use MSP430s. The key problem with these platforms is they are research platforms built for other researchers, not for novice developers or makers trying to build fun applications and learn things. Using a platform like WISP, an active RFID device, requires access to expensive RFID readers and custom programming modules. Flicker, Capybara, and Botoks all were built to explore specific hardware/circuit concepts in intermittent computing, including federating energy storage to reduce power failures (Flicker, Capybara), enabling rapid prototyping (Flicker), and using RC circuits for robust timekeeping across power failures (Botoks). Flicker is most closely related, claiming to enable novices to rapidly prototype. However, Flicker is a hardware platform *only* and does not enable a new programming model for batteryless systems, but supports existing ones. This is at once the best part, and great flaw of Flicker, as the programming models supported are all C-based, and as we discussed in Section 2.2.3, these expert level programming models are not sufficient for many novice programmers. This can be seen as Flicker was not readily adopted by the novice community, even now.

Finally, and most importantly, all of these platforms are highly constrained, comprising less than 256 KB of code memory, and 4 KB of SRAM for scratch space. Modern ARM Cortex M microcontrollers, such as [24], have up to 32 times the scratch space, and four times more memory, with processors ten times faster or more. The resource constraints of the TI MSP430 microcontroller preclude running a sophisticated interpreter with large memory footprint like Python (or JavaScript). Moreover, these platforms are bespoke, research prototypes, which are not part of any existing large community or ecosystem. To be truly effective and broadly adopted, a future platform targeting makers and novices must integrate closely with common hardware, like Adafruit's Metro M0 [5] boards.

2.3. BFREE SYSTEM DESIGN

We have developed BFree (shown in Figure 2.5) for novice developers, makers, or prototypers who want to program battery-free and energy harvesting Internet of Things applications easily with Python. To enable this vision, BFree's goals are:

1. build a power failure resilient version of the Python runtime that can execute arbitrary Python programs;

2. design hobbyist-usable hardware that can harvest ambient energy;

3. enable a rich set of built-in functionality with sensors and libraries, and

4. integrate (1–3) into a complete platform focused on entire stack (software and hardware) usability that can be used with common hobbyist platforms.

A key idea to enable the above goal number (4) is to build on and modify CircuitPython, and leverage the hardware ecosystem surrounding it. With an active user base and support by the maker-oriented company [5], this integration, though more difficult than developing a custom solution as in previous work [84, 49, 250], will finally enable the adoption of battery-free computing with BFree. Instead of designing everything from scratch, we seek to build on the existing maker and hobbyist electronics communities' momentum and enable them to go *batteryless*.

2.3.1. EXECUTING PYTHON CODE

As shown in Figure 2.5, BFree is split across the *BFree runtime* and *BFree hardware shield*, which sits on top of the Adafruit Metro M0 board (see again Figure 2.1). Python programs can be executed on the Adafruit Metro M0, a simple Arduino-style breakout board that has power circuitry, USB, LEDs, and pins broken out for easy prototyping, build around an ARM Cortex-M0 microcontroller [24]. Novice developers write Python code on their laptop, send it to the Metro M0 over USB, where it is then compiled and executed.

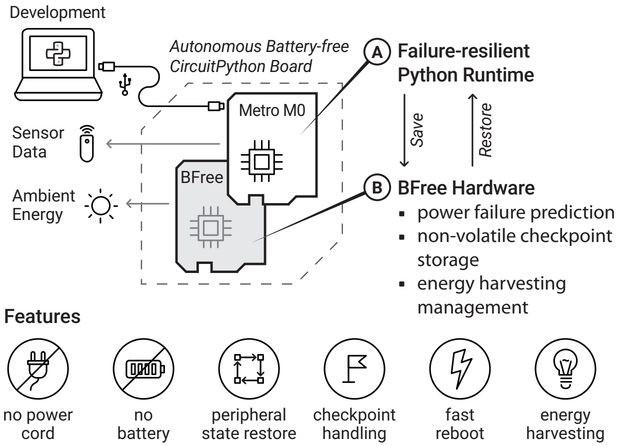


Figure 2.5: BFree system overview. Programmers develop Python applications on their PC and upload them to the Adafruit Metro M0 board so that their application is interpreted by a power failure-resilient Python interpreter. The necessary power-failure management is performed via communicating with the BFree hardware that harvests energy, handles checkpoints, keeps track of peripherals, and ensures fast reboot.

2.3.2. Energy Harvesting

The Metro M0 board is not equipped with energy harvesting circuitry for battery-free operation, nor does it have any fast and durable non-volatile memory (it only has FLASH—slow, energy-intensive, and not very durable). BFree hardware provides access to the energy available in the ambient; solar, RF, kinetic, or any other type (solar being a default one). The BFree hardware sits on top of the Adafruit Metro M0 board while still exposing pins for prototyping. Energy is harvested, stored in a capacitor, then made available to the Metro M0 board to execute the Python program embedded in the core. When the Metro M0 and BFree shield is disconnected from power (battery, USB, or wall socket) the energy harvesting circuitry takes over, enabling battery-free and untethered operation.

2.3.3. Checkpoints and Progress

The BFree shield and our Python runtime are co-designed to enable resilience to power failures. Before a power failure, the progress of the Python program is checkpointed. The checkpoint operation takes all the volatile memory and data stored on the Adafruit Metro M0 that is required to resume execution and saves it to the fast non-volatile storage (FRAM) on the BFree shield. When enough energy is stored in the BFree shield for the whole device to turn on, the Metro M0 turns on, downloads and restores the checkpoint of past progress from the BFree shield, and then resumes executing the Python program from where it left off. Doing so keeps the program from wasting cycles re-executing old code, keeps memory and progress consistent, and makes it easier for the novice programmer since they do not have to figure out what to do in the face of power failures. These checkpoints are carefully managed between the BFree shield and Metro M0 so that the Python code can be safely and consistently executed despite power failures.

2.3.4. LIBRARIES AND SENSING

As shown in Figure 2.5, programmers can attach sensors via a breadboard and then import built-in or third-party libraries to use with their Python program. Some built-in libraries provide access to hardware functions on the Metro M0 and are heavily used by MicroPython/CircuitPython programmers, for example, the `time`, `digitalio`, and `busio` libraries. These enable time delays and measurements, usage of the digital pins, and interaction with I2C and similar communication protocols (which enable use of external 'breadboarded' sensors), respectively. The BFree runtime provides modified versions of these libraries, building in features to enable intermittent computation, reducing surprises for the novice programmer who wants to move to battery-free operation. For example, the I2C and digital pin state are saved in a checkpoint so that on reboot the correct pin direction and output, as well as I2C configuration is restored. These system design points together enable a broad range of built-in functionality and external sensors and peripherals.

2.3.5. DEPLOYMENT

Programmers bring this all together to deploy real-world applications with BFree. Once programmed through the laptop or desktop, the Metro M0 equipped with the BFree shield is disconnected and deployed in the wild, harvesting energy and performing computation and sensing despite power failures.

2.4. IMPLEMENTATION

We now describe the implementation challenges and details stemming from the system requirements necessary to port a significant (CircuitPython) codebase to intermittent operation. Our core implementation requirement is to enable power failure resilient operation for interpreted Python programs over bare metal programming environments such as C/C++. To achieve this (i) the progress of computation and memory consistency against power failures should be ensured; (ii) the system reboot procedure should be optimized so that the system recovers from power failures faster; (iii) peripheral state should be restored so that peripheral interaction continues from a consistent state; and (iv) system level libraries `time`, `digitalio`, and `busio` must be adapted for persistent and reliable operation. The BFree additions to the CircuitPython interpreter include over 5500 lines of low-level code, written in C and assembly, split across the ARM Cortex-M MCU on the Metro, and the MSP430FR MCU on the BFree shield. This represents a substantial addition to the core codebase of CircuitPython to enable intermittent operation and checkpointing.

2.4.1. HARDWARE

We designed and built the BFree shield capable of energy harvesting, power-failure detection, managing checkpoints in non-volatile memory, and keeping track of time. This shield sits on top of the Adafruit Metro M0 board—shown in Figure 2.6. The details of the hardware design are as follows.

Energy Harvesting Circuitry: The energy harvesting circuitry on the BFree shield is the source of energy for all the components on both the BFree shield as well as the Adafruit

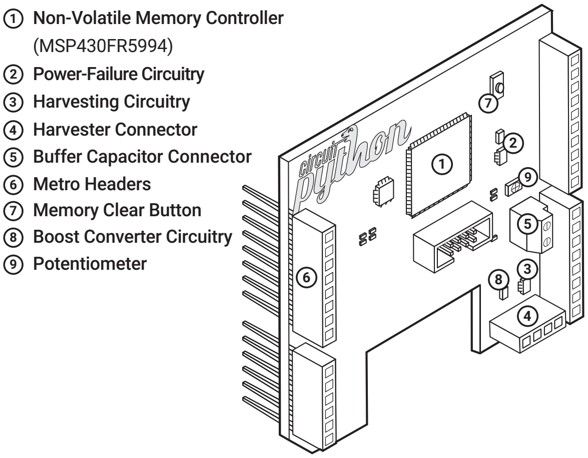

1 Non-Volatile Memory Controller (MSP430FR5994)
2 Power-Failure Circuitry
3 Harvesting Circuitry
4 Harvester Connector
5 Buffer Capacitor Connector
6 Metro Headers
7 Memory Clear Button
8 Boost Converter Circuitry
9 Potentiometer
circuit python
1
2
3
4
5
6
7
8
9

Figure 2.6: BFree hardware with annotated components and functions. The non-volatile memory controller captures and logs checkpoint data. Power-failure circuitry allows for a configurable power-failure signal when the energy in the storage capacitor is running low. Harvesting circuitry accumulates the harvested energy from the harvester connector into the buffer capacitor and provides hysteresis control. The memory-clear button provides the user with an easy way to restart the application (i.e., delete the checkpoints). Headers connect the shield to the Adafruit Metro M-1 board.

Metro M0 board during battery-free operation. The BFree shield can be equipped with any energy harvesting source, e.g., a solar panel through a connector to harvest ambient energy. The energy harvesting circuitry accumulates the harvested energy from this energy harvesting source into a capacitor, which is user-selectable[3]. A hysteresis control (via MIC841 voltage comparator [159]) is implemented to enable operation when the stored energy in this capacitor is above a predefined threshold.

Power Failure Prediction: The BFree shield is equipped with a user-configurable voltage comparator—using a potentiometer and a nanopower comparator (TI TLV3691 [225])—that is used to signal the BFree runtime that the storage capacitor voltage, and therefore the remaining energy, is running low. This threshold voltage is made configurable because the ideal setting highly depends on the capacitor's discharge speed. In turn, this is a relation between the current draw of the system, the incoming energy from the harvester, and the size of the capacitor. As BFree aims to support many different applications with wildly different requirements, this needs to be configurable to fit the applications' needs. This power failure prediction signal can optionally be used by the BFree runtime to change the checkpoint scheduling, reducing the checkpointing overhead when the system's remaining energy is not critically low.

Checkpoint Storage: The BFree shield is composed of non-volatile memory (FRAM) hosted by a Texas Instruments MSP430FR5994 microcontroller [96] that has 256 KB FRAM and 4 KB SRAM. The software on this microcontroller implements a map-based file

[3] We will report the exact values of both capacitor and the type of energy harvester while discussing results assessing the impact of energy trace on BFree program execution in Section 2.6.3.

system to store the checkpoints and takes care of the required double-buffering to keep checkpoints from becoming corrupted. The microcontroller on the Adafruit Metro M0 board triggers a checkpoint operation by sending commands to the BFree shield over an SPI bus. The MSP-based FRAM MCU was chosen to alleviate the BFree runtime from the checkpoint management burden. However, it would be easy to equip the system with a memory-only module and modify the BFree runtime to perform the data management (e.g., double-buffering), provided that the Adafruit Metro M0 board would have on-board FRAM.

2.4.2. SOFTWARE

We modified the CircuitPython runtime interpreter so that the necessary operations to ensure the progress of computation and memory consistency are taken.

Checkpoints capture the volatile state of the ARM Cortex-M0 microcontroller on the Metro M0 board that executes the Python interpreter. The volatile state of the ARM Cortex-M0 includes the contents of the volatile main memory, where the global and local variables are stored, as well as the contents of the registers and the state of any initialised interface such as I2C. Capturing this state before a power failure, then restoring this state on reboot ensures the progress of computation. Computation continues from where it left just after the recovery from a power failure. However, not all state has to be restored when the system restores a checkpoint. Some states such as USB and the file system should be reinitialized from scratch. The CircuitPython codebase has been explored and divided into two parts. One part that can be safely restored and one part that requires re-initialization when the system attempts a reboot. We instrumented the Python interpreter with *potential* (i.e., not forced) checkpoint locations. Most notably, we added them to the main interpretation loop. The locations are potential because whether a checkpoint is performed at these locations is decided by the current checkpoint strategy (e.g., by the period-based strategy). An alternative is to trigger checkpoints via power-failure prediction notification (which is possible with BFree hardware).

Checkpoint Content: The checkpoint contains all the necessary information for the system to continue where it left off. As CircuitPython itself is written in C, this contains: (i) the registers, (ii) global variables, (iii) the stack, and (iv) any dynamically allocated memory. The C programming language allows for custom memory schemes as all the memory can be accessed freely through pointer manipulation. A custom memory allocator is, therefore, not uncommon within embedded software development. This is also the case in CircuitPython by means of a garbage collector. The garbage collection sub-system occupies all the remaining memory in the system after the global variables and the stack are reserved, the size of which is determined within the linker script during compilation. The garbage collector is also responsible for all the dynamically allocated memory in the system, and all the Python interpreter specific stacks—as CircuitPython is a stack-based interpreter. Our checkpointing implementation allows for the dynamic specification of memory regions that are required to be checkpointed.

Excising C frameworks that support intermittent execution, such as [250], expect that the whole system needs to be restored. In real-world applications, such as CircuitPython, this does not hold. There are parts of the system that need to be reinitialized every reboot. In CircuitPython, this mainly consists of the USB and flash file system-related data and

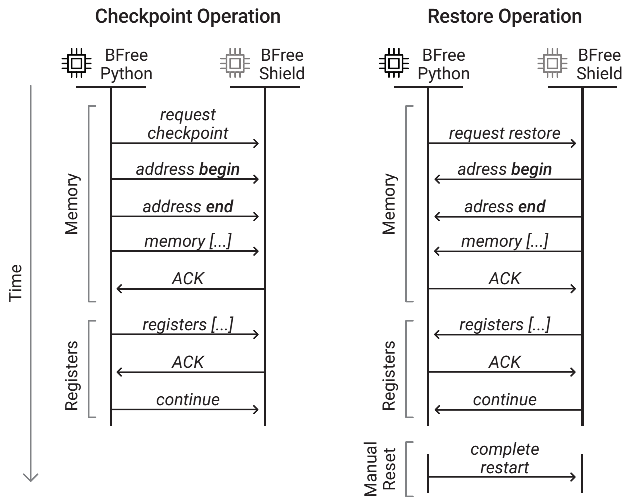


Figure 2.7: Checkpoints ensure the progress of computation and memory consistency. The BFree Python interpreter sends the checkpoint data (volatile memory contents and register values) over the SPI bus to the BFree shield so that this data is stored in non-volatile memory. Upon recovery from a power failure, the stored checkpoint data is used to restore the state of computation and the memory contents.

variables. Excluding this data from the checkpoint is not only to reduce the checkpoint size—and therefore the checkpoint time—but also having these values at anything other than zero can (and will) cause bugs during the initialization of these subsystems. We inspected the CircuitPython code base and selected all global variables that require to be checkpointed. These are put into a special configuration file, and the compiler will place these special memory regions that are excluded from the checkpoint.

Checkpoint Operation: The left-hand side of Figure 2.7 depicts the steps taken during the checkpoint save and restore operations. The checkpoint data is sent over the SPI bus from the ARM Cortex-M0 microcontroller on the Metro M0 board to the MSP430FR5994 microcontroller on the BFree shield. The checkpoint includes the start and end addresses of the volatile memory region on the ARM Cortex-M0 microcontroller and the contents of this memory space. This step is repeated for the different memory sections in CircuitPython. Moreover, the contents of the general-purpose and special registers should also be saved within the checkpoint context. Special attention is given to the registers, as these need to be checkpointed last, and their content must not be altered during the checkpoint procedure. Otherwise the state of the program will be corrupted when a restore is performed. To guarantee this, the register checkpointing is written primarily in assembly language.

All this information is sent over SPI and stored in non-volatile FRAM of the MSP430FR on the BFree shield. It is worth mentioning that the checkpointed data is stored in a double-buffered memory region in FRAM to ensure memory consistency: the checkpoint is stored in a temporary memory region where the original memory region holds the data of the previous checkpoint taken successfully. After the checkpoint data is entirely saved in the temporary buffer, an atomic variable is modified to swap the temporary and original buffers, thereby committing the checkpoint. If the preexisting checkpoint would be directly overwritten, a power failure during a checkpoint would lead to a corrupted state and would therefore require a complete restart of the application.

Restore Operation: Figure 2.7 also depicts the restore operation. After sufficient energy is harvested and the system reboots to start operating again, the latest successful checkpoint needs to be restored so that the computation continues from where it left. For checkpoint recovery, the Metro M0 board communicates with the MSP430FR5994 on the BFree board over SPI. It reads the contents of the checkpointed memory regions and the values of the registers and peripherals. The recovery is completed by jumping to the next instruction to be executed. The successful restoration of the volatile state captured in the checkpoint ensures the progress of computation and memory consistency. To enable a manual hard reset—restarting the application from the beginning—an additional reset button is provided on the BFree board. Pressing this button while powering up the system will delete the existing checkpoint. Additionally, this can be achieved by uploading a new Python script to the Metro M0 or by restarting the current Python script using the standard Python Read-Evaluate-Print-Loop (REPL) language shell made available over the serial interface.

Reducing System Restart Burden: Unmodified CircuitPython takes an entire second to boot, which wastes a significant amount of energy. The Metro M0 board has a bootloader to easily update the CircuitPython binary. This increases the bootup time because some time is reserved for the user to notify the system to mount it for a CircuitPython update. Additionally, approximately 700 ms is introduced to let the user enter a so-called safe mode, shown in Figure 2.16. We remove this delay and optimize other delays during the reboot in our modification of the runtime (in-depth discussion of this matter is provided in Section 2.6.5). We do not alter the bootloader code, as we want hobbyists to be able to freely change their CircuitPython image to our intermittent version (BFree) and back without the need for an external programmer.

Peripherals and Libraries: For the proper operation of battery-free hardware platforms, the states of peripherals, including digital pins, I2C, and time, should be restored after a power failure. As an example, during a particular I2C communication between the microcontroller and a sensor (i) first the dedicated I/O ports of the microcontroller are configured for I2C operation, (ii) then the sensor is configured for the desired operation, (iii) and finally the command for the desired operation is sent, e.g., sampling or actuation. Upon power failure, if the command is sent without re-configuring the I/O ports of the microcontroller and/or sensor, the sensor sampling will not work correctly.

In order to require minimal changes to existing CircuitPython applications that make peripherals and external sensors, a balance was struck between automated restoration and programmer aided restoration. The states of the peripherals are automatically restored during the restore operation. However, the initialization of different sensors can wildly differ on a case by case basis. Therefore any re-initialization required by the sensor must be performed by the programmer. In traditional CircuitPython libraries, the split between peripheral and sensor initialization is already present, therefore it is often only needed to relocate the sensor initialization library calls to just before the reading. The effects and impact of this trade-off will be further showcased in Section 2.5.

Checkpoint Strategies: We designed three checkpoint scheduling strategies that use the information provided by the power-failure prediction signal from the BFree shield (see the description of power failure prediction in Section 2.4.1). These strategies, named (i) *periodic*, (ii) *trigger*, and (iii) *hybrid*, have different overheads since they generate a different number of checkpoints during program execution. The periodic strategy generates a checkpoint of the system every p milliseconds, without considering the power-failure signal. Alternatively, the trigger strategy generates a checkpoint every p milliseconds *only* when the power-failure signal is active. The hybrid strategy is a combination of the other two strategies: it generates a checkpoint either every p milliseconds when the energy level is high (i.e., the power-failure signal is not active) or every q milliseconds when the power-failure signal is active (i.e., low energy operation).

Among the checkpoint strategies, the trigger strategy eliminates all checkpoints when the energy level is not low, a desirable property. However, when the power-failure signal is active, this strategy requires enough energy (i.e., active-time) to perform a checkpoint—which might not be guaranteed due to varying energy harvesting conditions and different capacitor sizes. Therefore, if the storage capacitor discharges faster than checkpoint generation, periodic checkpoints are ideal. Under varying energy harvesting conditions (e.g., solar energy harvesting frequently distracted by clouds), the hybrid strategy can perform the best. In our system, the programmer can switch between these checkpoint strategies as well as change their parameters (i.e., p) at runtime using a builtin Python library. This enables dynamic and configurable checkpointing with respect to the characteristics of the energy harvesting environment during the application's execution.

2.5. BFree Deployment and Use Cases

The key question for BFree is how useful it is for building battery-free applications. In this section, we engage in proof by demonstration, showing a range of battery-free applications that BFree enables for novice programmers. We build *useful*, sustainable, and battery-free applications around BFree with *unmodified* CircuitPython. Figure 2.8 shows hardware prototypes of two example applications, (i) LoRa[4] sensor mote and (ii) electronic paper temperature display. For each application, we design an experimental plan to explore how battery-free operation affects design and deployment. We proceed with discussing each prototype in detail.

2.5.1. BFree LoRa Sensor Mote

The classic use case for embedded systems is to measure environmental factors long term, with seminal examples deploying 'motes' like the TelosB [193] for applications including volcano monitoring [241], habitat, and wildlife monitoring [146], wildfire detection [78], and precision agriculture [121], among many others. BFree enables these types of environmental monitoring applications programmed in Python, without relying on batteries.

[4] LoRa is a low power, wide area, low data rate network protocol. It is increasingly common in distributed sensor networks because of unlicensed operation and very long communication ranges. More details on LoRa can be found in many academic surveys, e.g., [195].

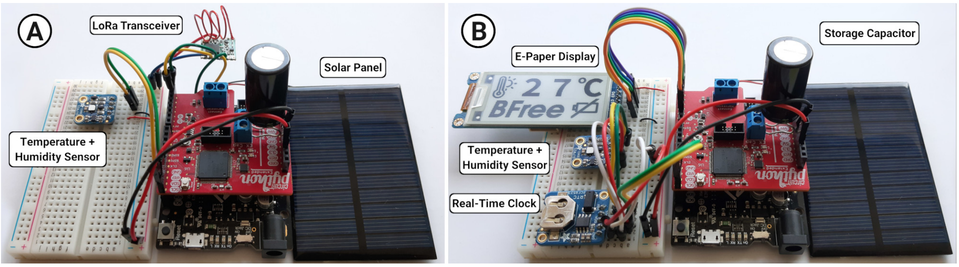


Figure 2.8: Assembled hardware to demonstrate two hobbyist-grade applications written in unmodified CircuitPython running on intermittent energy with BFree: **(A)** LoRa sensor mote, and **(B)** electronic paper temperature display.

All of these applications have similar functions typical of an edge computing system; they opportunistically measure some aspect of the environment, process and summarize the collected data, and (when a specific condition is met) share this information wirelessly. We prototyped an environmental monitoring system hardware on a breadboard (as is typical of a hobbyist) and programmed it using Python software. The sensor senses temperature and humidity, averages multiple readings, and then sends that data wirelessly to a base station. We used an unmodified Adafruit SI7021 [3] breakout board, which continuously measures temperature and humidity and sends the measured data to the Adafruit Metro M0 board for further processing through the I2C bus. When a predefined number of samples are collected, BFree averages them and broadcasts this value using an Adafruit RFM95W [7] LoRa radio transceiver breakout board connected to the Adafruit Metro M0 board via the SPI port. To receive and verify the messages broadcast by BFree, a dedicated LoRa messages collector operating on constant power that continuously listens for LoRa packets has also been built using a second Metro M0 board running vanilla CircuitPython.

For both the SI7021 sensor and the RFM95W LoRa module, we used the *unaltered* Python libraries provided by Adafruit. Within the application Python the only alterations were (i) moving the LoRa initialization to the transmission code and (ii) disabling checkpoints before the LoRa initialization and transmission and enabling them after, which is provided in an API for BFree programmers. Doing so is required as the LoRa module has an internal state that needs to be configured at boot, which will be lost after every power failure. The SI7021 has no internal state and is therefore automatically handled by the BFree runtime. So *only two* additional single-line statements from the programmer are required to fully unplug the USB cable, leave the battery behind, and survive off energy harvesting only. With BFree, programming in Python, and using hobbyist electronics, this relatively complex application is easily transformed into a battery-free system resilient to power failures.

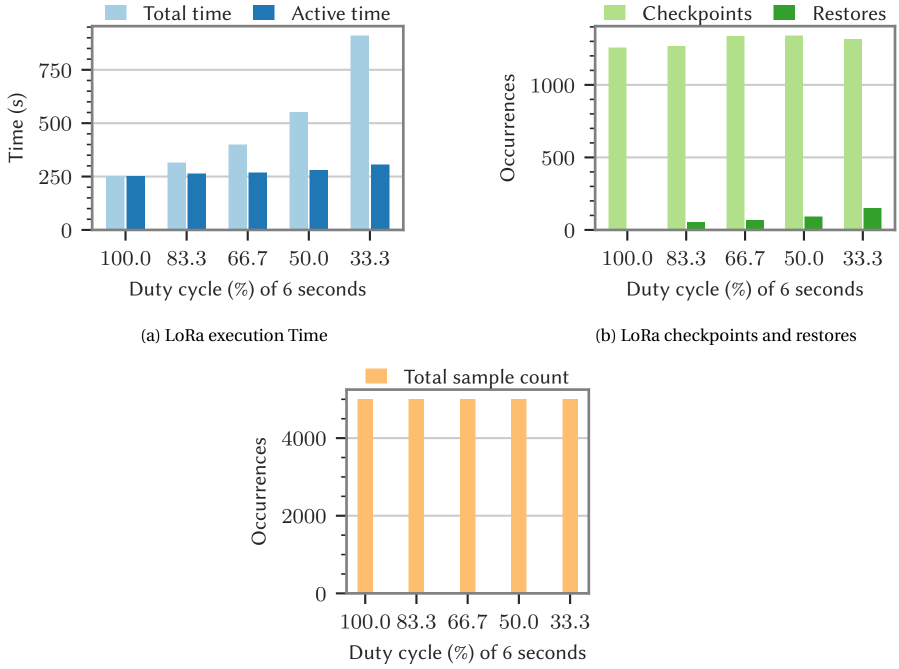
Experimental Setup: To demonstrate that BFree works as expected, we run a series of benchmarking tests on the LoRa application that exercises the checkpointing and restore the functionality of a complex, peripheral enabled application. We limited the number of LoRa packet broadcasts to 50, and the number of samples collected (both the temperature and the humidity) before a broadcast to 100. These numbers were chosen such that the

Table 2.1: Duty cycle to on/off relation.

Duty cycle (%)	**On time** (s)	**Off time** (s)
100	∞	0 (i.e., continuous power)
83.3	5	1
66.6	4	2
50	3	3
33.3	2	4

benchmark on continuous power completes in approximately 250 seconds. The payload of each LoRa packet contains three main fields coded in plain text: (i) broadcast packed ID, and (i) average temperature measurement, and (iii) average humidity measurement (comma-separated two decimals per each of the two measurements). To make our evaluation repeatable and remove any potential energy harvesting variability, we connect a controlled square wave (low state of zero volts and high state of positive voltage) to the harvester input. This way, we can experiment with the on/off time relation (i.e., duty cycle) of the system in a controlled way and more accurately synchronize the start of the application with the start of an on period. Additionally, by varying only the duty cycle and not the time it takes to perform one on/off cycle we can directly compare the results from run to run. The only limitation we encountered using this controlled setup is the lack of power failure detection, as the square wave causes an immediate power fault, limiting us to investigate only the periodic checkpointing strategy. We configured the system with a checkpoint period of 200 ms, i.e., when the system is active every 200 ms a checkpoint is created. Additionally, we chose a period of six seconds and ran the application using five different duty cycles shown in Table 2.1. Note that an on-time of one second is too short to be feasible due to the limitations imposed by the setup, initialization time of the LoRa module, and broadcast time of the LoRa module.

Summary of Results: We experiment with different levels of intermittency, i.e., different levels of energy availability. As the duty cycle, we selected (see again Table 2.1) decreases from 100% (i.e., continuous power) down to 33.3%, the *total time* to complete the benchmark increases. However, the *active time* (i.e., the time the system is on and executing code) remains somewhat constant, see Figure 2.9a. The difference between the active times is the accumulation of all the reboot procedures, checkpoint restores, and re-execution of code executed before a power failure and after the last checkpoint. As shown in the same figure, this only accounts for a small amount of the active time. The total number of samples, as can be seen in Figure 2.9c, stays constant throughout all the runs since our application performs 50 LoRa packet broadcasts and broadcasts data only after 100 samples are taken and averaged. The number of restorations in Figure 2.9b is equal to the number of off-time occurrences throughout the run. The number of checkpoints in this configuration is approximately equal to the active time divided by the checkpoint period.



(a) LoRa execution Time

(b) LoRa checkpoints and restores

(c) LoRa total number of samples

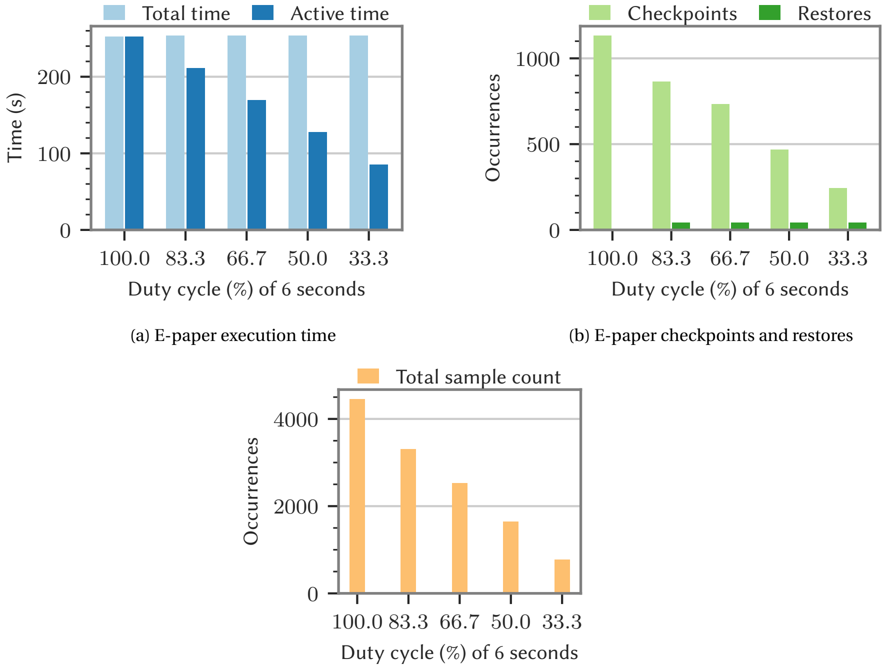
Figure 2.9: Evaluation of the LoRa sensor mote application. This application sends a total of 50 LoRa broadcast packets that carry the average temperature and humidity of 100 samples collected. The intermittency is varied using different duty cycles within 6 second period (see Table 2.1). The period checkpoint strategy with a checkpoint period of 200 ms is used. The results indicate that (a) as the off time increases, the total time increases as it takes longer to complete the benchmark application. The constant active time (the time the system is actually on) shows that the overhead caused by re-execution and restores is minimal; (b) the number of checkpoints stays constant due to the periodic checkpoint strategy; (c) as the benchmark is completed after 5000 samples are collected (50 times 100 samples), the total number of samples stays constant irrespective of varying duty cycles.

2.5.2. BFree Electronic Paper Display

The second demonstration system we developed is a BFree electronic paper display that updates a display output every n seconds with the average temperature since the last display refresh. These displays are becoming more common for electronic shelf labels in automated supermarkets and grocery stores, as they can automatically update values and pricing without human intervention. Of course, these systems suffer from the use of a battery. With BFree, a shelf label can be energy harvesting and auto-updating. For this demonstration, we breadboard a BFree system that takes average temperature readings over time and displays it on e-paper at a set update rate. We use the same temperature sensor as used in the LoRa application, i.e., Adafruit SI7021 (see Section 2.5.1). For the display, we use an electronic paper (e-paper) display. E-paper displays are ideal for intermittent applications as they retain their display state even if there is no power. A limitation with e-paper displays is the low maximum refresh rate and (in some cases) the requirement to perform a time-consuming update cycle whenever a section of the display requires changes. We set to show the average temperature every ten seconds, so the maximum update rate is not an issue. On the other hand, to overcome the need to update the whole screen (which can take between two and ten seconds depending on the model of the display), we chose a Wemos Electronics 2.13 inch 250 × 122 e-paper display [240] that supports partial updates. Partial updates allow us to only write to a small section of the screen, reducing the update time to around 0.7 seconds. As there are no e-paper libraries for CircuitPython that allow for partial updates, we wrote a custom library in C as a demonstration that BFree also works for built-in libraries written in C. Additionally, the e-paper application demonstrates a different kind of intermittent application. For LoRa, the number of samples and broadcasts was fixed, so a certain active time is required to complete the benchmark. In contrast, for the e-paper application, we added a real-time clock (RTC) and fixed the refresh-period of the e-paper display. The RTC is an Adafruit PCF8523 module [4] and is used without modifying the provided Python library. Note that the RTC module supports a coin cell battery by default (which was used in the experiment), but any (super-)capacitor can also be used to keep BFree completely battery-free (for example 220 mF capacitor should keep the RTC running for at least a month assuming a startup charge of 3.3 V, see [172, Page 34], simple circuitry could additionally be used to charge the RTC capacitor from harvested energy).

Experimental Setup: We chose to refresh the e-paper screen every ten seconds. To complete the benchmark, a total of 25 screen refreshes must be performed. These numbers were chosen to (i) not damage the e-paper display by exceeding the refresh rate and (ii) limit the total benchmark time on continuous power to approximately 250 seconds. For the same reasons mentioned in Section 2.5.1, we use a square wave with different duty cycles (see Table 2.1) to power the board and a checkpoint period of 200 ms to evaluate the applications.

Summary of Results: In contrast to the LoRa application presented earlier, the total time of the e-paper application remains constant (see Figure 2.10a) as the duty cycle decreases from 100% down to 33.3%. This is due to the fact that we fixed the amount of time the benchmark takes (using the RTC and a fixed number of screen updates) instead of fixing the number of samples required to complete the benchmark. Consequently, the



(a) E-paper execution time

(b) E-paper checkpoints and restores

(c) E-paper total number of samples

Figure 2.10: Evaluation of the e-paper application benchmark refreshing the screen a total of 25 times with the average temperature collected during a ten-second period between refreshes. The intermittency is varied using different duty cycles of a six-second period (see Table 2.1). The period checkpoint strategy is used with a checkpoint every 200 ms. The results indicate that (a) total time stays constant and the active time decreases (in contrast to Figure 2.9a). This is because of the time-sensitive nature of the application; (b) the number of checkpoints decrease when the duty cycle increases because it is directly tied to the active time when the period checkpoint strategy is used. The restores are constant for the same reason; (c) because the benchmark is completed after 250 seconds (25 times 10 seconds) the total number of samples decreases as the active time also decreases.

active time, the total number of samples (Figure 2.10c) and the number of checkpoints (Figure 2.10b) decrease. Since the number of times the system turns off during the benchmark is also constant—except for the case of continuous power—the number of restores is constant.

2.6. Evaluation

The main design goals of BFree are to (i) match the runtime performance of CircuitPython on continuous energy, (ii) enable the progress of computation during intermittent operation that relies only on harvested energy, (iii) introduce a minimal burden on baseline CircuitPython in terms of resources. In order to see if we meet these goals, we evaluate our BFree prototype implementation by comparing its performance and runtime overhead to unmodified baseline CircuitPython/MicroPython and regular C. In particular, we perform the following experimental evaluation:

1. Running benchmark programs in order to measure the performance versus baseline, and verify the correctness of execution despite multiple power failures;
2. Demonstrating the effects of using harvested energy and different checkpoint strategies;
3. Measuring the cold boot (startup time) of the system;
4. Profiling the resource requirements such as the main and non-volatile memory overhead;

Experimental Setup: For each experiment, we use an Adafruit Metro M0 with the BFree shield connected on top. We control power delivery in one of two ways: (i) via a microcontroller that gates power using a MOSFET from a Digilent power supply to the BFree device under test, allowing us to repeatably simulate intermittency rates, and (ii) via a lamp from which the BFree shield harvests light energy. We use a Keysight DSOX3014A oscilloscope and Saleae Logic Pro 8 Logic Analyzer to perform time and electrical power (P/I/V/E) measurements. When making comparisons, we use baseline CircuitPython firmware (version 4.1.0) running on the same experimental setup (i.e., software and hardware).

2.6.1. Comparing Execution Time: C, CircuitPython, BFree

An interpreted programming language, such as CircuitPython, will, in most cases, be slower (in terms of code execution duration) than a compiled language, such as C—a language of choice for professional development with embedded microcontrollers. Python (thus also CircuitPython) will trade-off execution speed for code extensibility, code clarity, and multi-purpose features. In this experiment, we measure the difference in execution time between interpreted CircuitPython and bare-metal C compiled to machine code. Additionally, we compare the execution time of BFree (as in our modified CircuitPython runtime coupled with the BFree shield) compared to vanilla (i.e., unmodified) CircuitPython.

Experiment: We measured CircuitPython's execution speed for selected three simple programs, (i) a Fibonacci sequence generator (for one specific value), (ii) a program that outputs the length of a predefined string, and (iii) a program that counts bits of a predefined bit sequence. Each of these programs was implemented in simple Python, and in C. All programs were written such that they did not rely on any internal or high-level

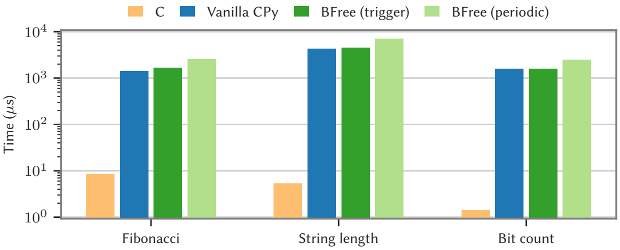


Figure 2.11: Comparison of execution time of three applications: (i) generator of Fibonacci sequence for the value of 40 (denoted as *Fibonacci*), (ii) program that outputs the length of a predefined string with a length of 40 characters (denoted as *String length*), and (iii) a program that counts the bits of the number 0x7FFFFFFF (denoted as *Bit count*) written with (i) vanilla C language, (ii) vanilla CircuitPython (denoted as *CPy*), and (iii) CircuitPython running under BFree runtime. Two cases of checkpoining were used: (i) *trigger*, with checkpoint triggered by an on-board BFree comparator and (ii) *periodic*, with checkpoint every 200 ms. Each application is run 10000 times and the result presented is the average of all the runs. Applications implemented in CircuitPython with and without BFree shield take up to a thousand times more time to complete than simple C implementations. On the other hand, BFree introduces a small overhead over vanilla CircuitPython.

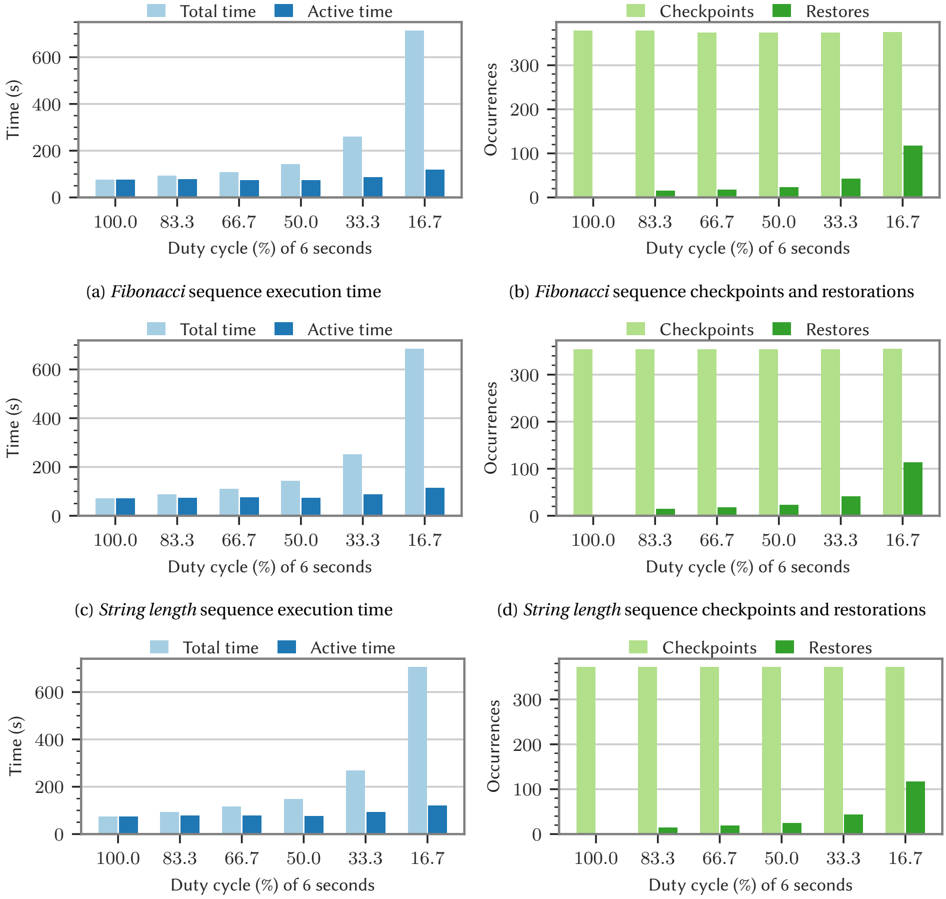
functions (of either C or Python). All three programs (for C and for Python), which source code is available in [230], were executed on Adafruit Metro M0 board with and without BFree shield. The same programs were also run on two versions of BFree: (i) *trigger*, when BFree on-board comparator triggered a checkpoint (refer to Section 2.4.1 (**Power Failure Prediction**)) and (ii) *periodic* when the checkpoint was triggered periodically every 200 ms, irrespective of supply voltage level. We compare all programs with continuous power to not confound execution time with delays from power failures and only compare the actual time computing.

Results: The result of the experiment is presented in Figure 2.11. Our results confirm our intuitive hypothesis that compiled C programs are faster than interpreted CircuitPython. In terms of actual values, compiled C demonstrates thousands of times faster execution than CircuitPython. On the other hand, we see that BFree gives only a small overhead compared to vanilla CircuitPython, which is already widely used by the maker community, proving that the end user of BFree will not experience significant performance degradation compared to vanilla CircuitPython. Our final observation is that event-driven checkpointing reduces the overhead of the intermittent runtime, which is clearly seen comparing BFree (triggered) and BFree (periodic) for all three programs in Figure 2.11.

2.6.2. BENCHMARKING FOR CORRECTNESS AND POWER FAILURE RESILIENCE

Next, we measured the execution time and correctness of the same benchmarks written in Python as the one used in the previous experiment (see Section 2.6.1) running on BFree. Each benchmark was executed on intermittent power of a varying duty cycle, the same way as done in Section 2.5.1, i.e., different duty cycles of a total period of six seconds, where a duty cycle of 100% equals constant power (see Table 2.1). Also similar to Section 2.5.1, a periodic checkpoint strategy was used with a period of 200 ms. The results of the evaluation are given in Figure 2.12.

The key takeaway from Figure 2.12 is that BFree allows these benchmark programs



(a) *Fibonacci* sequence execution time

(b) *Fibonacci* sequence checkpoints and restorations

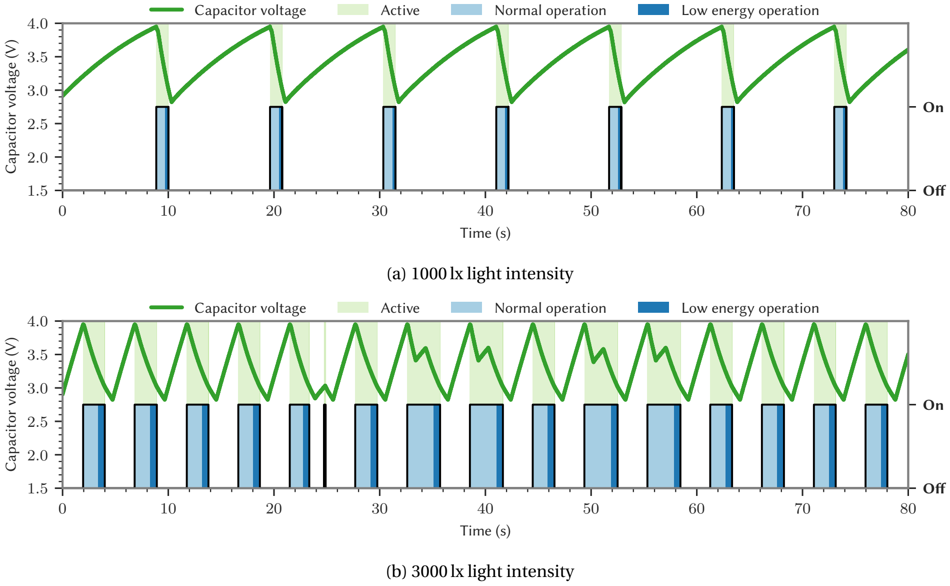
(c) *String length* sequence execution time

(d) *String length* sequence checkpoints and restorations

(e) *Bitcount* sequence execution time

(f) *Bitcount* sequence checkpoints and restorations

Figure 2.12: Evaluation of the *Fibonacci*, *String length*, and *Bit count* benchmarks written in Python (the same ones as used in Figure 2.11), using the same input as mentioned in Section 2.6.1. The benchmarks are executed in a loop of 30000 iterations for Fibonacci and bit count, and 10000 iterations for the string length application. The intermittency rate is varied using different duty cycles of a six-second period (Table 2.1) with the addition of the 16.7% duty cycle. The period checkpoint strategy is used with a checkpoint every 200 ms. (figures (a, (c), and (e)). As with the LoRa demonstration app (Section 2.5.1), these apps are computational and not time-based, therefore (again) the total time increases with the increase of the off-time. The slight increase in active time is due to the re-execution of code that happens after the last successful checkpoint, and restore operations (figures (b), (d), and (f)). Due to the period checkpoint strategy, the number of checkpoints remains the same. The number of restores is equal to the number of reboots of the system, and therefore is also equal to the number of off periods.



(a) 1000 lx light intensity

(b) 3000 lx light intensity

Figure 2.13: Recording of the energy trace (the voltage on the capacitor) when running the *Fibonacci* benchmark. The light intensity used was (a) 1000 lx, and (b) 3000 lx. Both traces used a 15 mF storage capacitor and the power failure signal (signaling low energy operation) is configured at 3.25 V on the storage capacitor. The low energy operation is nearly imperceptible at 1000 lx as the voltage of the storage capacitor is falling too fast.

to complete and make progress despite intermittent power failures. This would not be possible for this task using normal CircuitPython, as the benchmark will never complete. The time it takes for the benchmark to complete, including the time the system is off due to a power failure, is denoted as the *total time*. The *active time* is the time spent executing code, i.e., the total time minus all the off-times/time during power failures. Additionally, the active time (denoted as dark blue in Figure 2.12) only slightly increases when the number of power failures increases, signifying that our boot and restore overhead is small. Because we used a periodic checkpoint strategy, the number of checkpoints remains constant, and the number of restores is equal to the number of power failures. If a trigger-based strategy were used, the number of checkpoints would grow with the number of restorations. This is because the number of checkpoints created during low energy operation is determined by the programmer through the potentiometer setscrew on the BFree shield, the incoming energy and the consumed energy (Section 2.14).

We verified the outcome of each benchmark for correctness (for example, by comparing the values generated by the Fibonacci function). Since the programs running intermittent power had multiple power failures before the final result was computed, the intermediate results and checkpointing must work for the correct result. By getting the correct result for each of the benchmarks, this demonstrates that the BFree method preserves program consistency and correctness through power failures.

2.6.3. Execution in Varied Energy Harvesting Environments

To evaluate the power-management and harvesting subsystems of the BFree shield, we directly connected a solar panel to the BFree shield and recorded a trace of the system. The Python code running during this trace is the *Fibonacci* benchmark used in Section 2.6.2. We used a 6 V 0.6 W 80×55 mm off-the-shelf solar panel [168] and recorded the system's operation during an 80-second time window for two different light intensity levels. These levels are 1000 lux and 3000 lux and correspond, approximately, to an overcast day and shade during a sunny day, respectively. Note that no maximum power point tracking was performed as the solar panel was directly connected to the BFree shield; a 15 mF storage capacitor was used.

The results are shown in Figure 2.13. As can be seen the active time (light blue) in Figure 2.13a is significantly shorter than in Figure 2.13b, because of this the low energy operation (dark blue) is extremely short. We also clearly see the intermittent operation for both cases of light intensities.

2.6.4. Measuring the Effect of Checkpoint Strategies

Different energy scenarios require different checkpoint strategies (refer to Section 2.4.2). To observe the performance of each of the introduced strategies, we recorded a single active period for each of these strategies (periodic, trigger, and hybrid), each with two different configurations. The system configuration is identical to the one used in generating Figure 2.13b. The results are presented in Figure 2.14. The impact of the different strategies on the *Fibonacci* benchmark can be seen in Figure 2.15, with the summary presented in the caption of the figure.

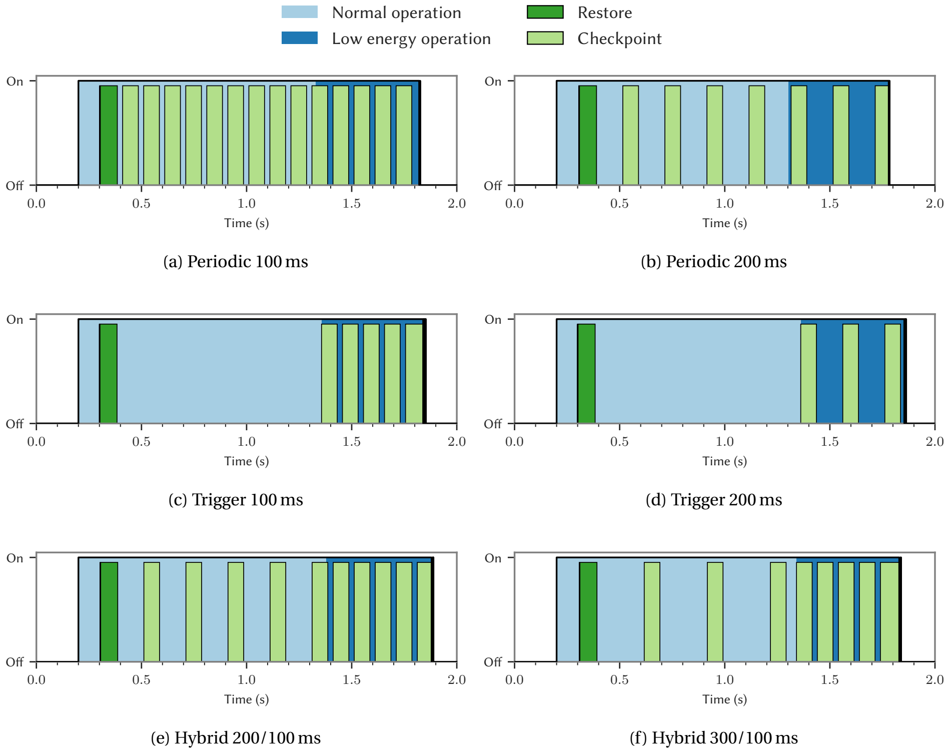
2.6.5. Startup Time

We measured the cold boot time of the baseline CircuitPython system to be 700 ms. By applying the techniques described in Section 2.4.2, we reduced the boot time of BFree considerably. A full boot of the BFree until the point the system is ready to restore a checkpoint is now approximately 270 ms. A detailed result of this comparison is presented in Figure 2.16.

2.6.6. Resource Usage

We measure the memory and speed overhead incurred from our checkpointing routines.

Checkpoint Content: BFree either periodically or on-demand checkpoints the volatile system state to the BFree shield over SPI. The transfer speed and the checkpoint size are dominating factors when it comes to the overhead introduced by BFree. The current SPI frequency is is set to 3 MHz. The checkpoint size is constant and dominated by the size of the garbage collector and is therefore always 27.6 kB. The distribution of the memory regions making up the checkpoint is shown in Table 2.2. Therefore, the time spent performing a checkpoint is also constant and measured it to be approximately 75 ms. A restoration is slightly slower, averaging around 80 ms. The SPI frequency can be increased to 4 MHz to improve the checkpoint time to approximately 52 ms, but this can leave the system vulnerable to interference on the SPI bus. Therefore, all the experiments were performed at the slower clock speed of 3 MHz.



(a) Periodic 100 ms

(b) Periodic 200 ms

(c) Trigger 100 ms

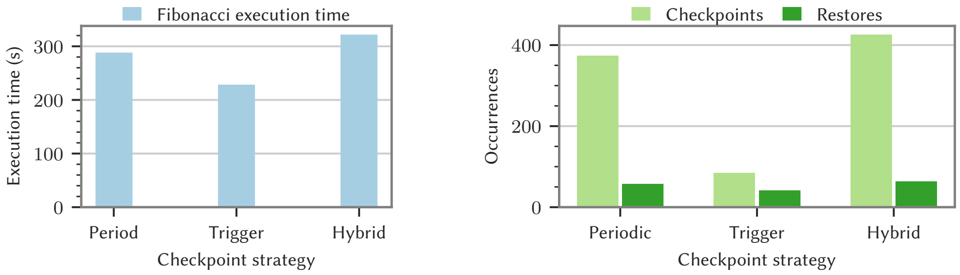
(d) Trigger 200 ms

(e) Hybrid 200/100 ms

(f) Hybrid 300/100 ms

Figure 2.14: Recordings of the checkpoint strategies—*period*, *trigger*, and *hybrid*—each with two different configurations that can all be configured at runtime by the user. The period based strategy, figure (a) and (b), does not take low energy into account when scheduling a checkpoint. When using the trigger based checkpoint strategy, figure (c) and (d), checkpoints are only performed when the energy level is low. Lastly the hybrid approach, figure (c) and (f), has two configuration periods: (i) one during normal operation, and (ii) one during low energy operation.

Checkpoint Code Overhead: The memory footprint of BFree compared to CircuitPython is presented in Table 2.3. Therein, it is seen that the FLASH footprint, representing the additional code, is almost the same for both CircuitPython and BFree due to the scale of the CircuitPython project. The additional mechanisms to allow for checkpoint creation and restoration introduced by BFree increase the RAM used by almost 17%, reducing the memory left for CircuitPython applications by 7.3%. To put these numbers into perspective, the RTC library used in Section 2.5.2 which was written by the CircuitPython community (in other words: not the authors of this thesis), requires almost seven times more memory than the additional memory required by BFree. Most of the increase in memory consumption is caused by additional data structures introduced to keep track of the peripheral state during execution and a 512 Byte ‘safe stack’ used to execute the checkpoint routines without changing any of the memory in use by the Python interpreter.



(a) *Fibonacci* sequence execution time

(b) *Fibonacci* sequence checkpoints and restorations

Figure 2.15: Evaluation of the Fibonacci benchmark from Section 2.6.2 running on intermittent power harvested using the same hardware setup as in Section 2.6.3. The chosen light intensity was 3000 lx; a 15 mF storage capacitor was used. We tried to keep the light intensity—and therefore the active time—constant, but in practice it varied from 2.1 to 2.6 seconds. The checkpoint strategies were configured as in Figure 2.14 (b), (d), and (f), respectively. The results show that the total execution time, see figure (a), not only depends on the chosen checkpoint strategy, but also the configuration parameters of the strategy. In theory, the total time of the hybrid strategy can be lower than the period strategy. In this case, the number of checkpoints taken, see figure (b), is lowest for the trigger strategy and highest for the hybrid strategy. The hybrid strategy can be improved by either (i) lowering the checkpoint frequency of any of the energy operation modes, or (ii) by tuning at what voltage of the storage capacitor the low energy operation starts (see Section 2.4.1 (**Power Failure Prediction**)). The number of restorations, see Figure (b), directly translate to the number of off periods encountered during operation.

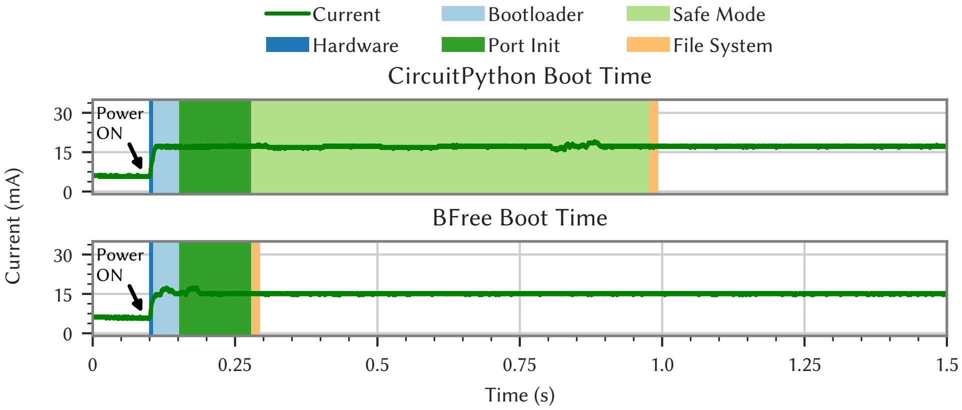


Figure 2.16: Measurement of the boot time of pure CircuitPython running on Adafruit Metro M0 board only (top graph), and CircuitPython running on top of Adafruit Metro M0 board, connected to BFree shield running BFree runtime (bottom graph). Current measurement was taken with a digital oscilloscope connected to a shunt resistor connected in series to Adafruit Metro M0 power supply port (powered by 3.3 V from an external source). The bottom figure clearly shows that we have optimized BFree boot time considerably by eliminating the introduced delay to enter 'safe mode' and by disabling the on-board RGB LED (used to show the current stage of operation on Metro M0 board), reducing the current consumption and voltage fluctuation during startup. BFree current consumption (shield and runtime) is approximately 2.2 mA and is the difference in current consumption between the top and the figure.

Table 2.2: BFree checkpoint content and its respective size in Bytes.

Checkpoint Content	**Size** (Bytes)
registers	68
.data	16
.bss	1320
stack	5280
garbage collector	20916
total	27600

Table 2.3: CircuitPython and BFree memory footprint. For ease of comparison the increase in FLASH and RAM use between CircuitPython and BFree is directly noted. The remaining memory for the CircuitPython application is roughly 24000 bytes when using BFree. To put these number into perspective, the RTC library used in Section 2.5.2 uses approximately 8000 bytes of memory, almost *seven* times more than the additional memory required by BFree.

CircuitPython (Bytes)		**BFree** (Bytes)		*Increase* (%)	
FLASH	RAM	FLASH	RAM	FLASH	RAM
193488	6840	204336	8000	5.61	16.96

2.6.7. DISCUSSION OF RESULTS

We consider different aspects of the evaluation results.

Performance: As shown with the benchmarks' performance, the additions made to the CircuitPython runtime system do not significantly hamper the operation of programs while plugged in. When running on harvested and intermittent power, the performance is mostly determined by the amount of energy that can be harvested and the length of power failures. We note that without careful recovery mechanisms like those implemented in BFree, CircuitPython programs will go into inconsistent states, corrupt memory, or crash when harvesting energy and running intermittently.

Overhead: The current mechanism for checkpointing takes nearly the entire volatile memory and saves it. This takes a significant amount of time to checkpoint versus bare-metal, C-based embedded runtime systems. However, this level of overhead is often acceptable for reactive, human speed, sensing-based, and low numerical complexity programs run by makers and hobbyists.

Hardware Limitations: We note that the CircuitPython Metro board is not necessarily designed for low power operation or untethered operation, as the 10–15 mA operating range is quite high (see again Figure 2.16), and the selection of circuitry for USB communication, power regulation, and others are designed for ease of manufacture and cost. Additionally, peripherals are connected in such a way that they use power even when off. A careful redesign of the Metro M0 board would easily reduce total power consumption by an order of magnitude by using lower power ARM microcontrollers, e.g., [14], selectively gating peripherals, and redesigning the power conditioning circuitry. Despite this, we have shown that application development with BFree is still possible and performant.

Fair Comparisons: We do not compare against state-of-the-art intermittent runtime systems like InK [250], Alpaca [142] or Chain [47], as these are written for professional programmers in C. Neither BFree nor CircuitPython/MicroPython can approach these runtimes' low overhead or performance (as shown in Section 2.6.1 comparing Python to C), as they spend significant clock cycles interpreting Python bytecode, managing the Python runtime, and facilitating operations that make the novice programmers life easier (such as handling USB connections and file systems). However, these systems are not in competition as they present radically different programming models, with the former serving expert developers. In contrast, we propose to serve novice and hobbyist developers seeking entrance into the ubiquitous and untethered computing world.

Harvesting Tradeoffs: During our evaluation, we used a single storage capacitor, and we only measured the trace using one solar panel (although with different light intensities). Changing these will affect energy harvesting and, potentially, the operation of the application. If we consider an application with a certain energy requirement, increasing the capacitor size will lead to a longer active time. This means that a different checkpoint strategy might be optimal. If the harvester output energy is increased (e.g., by using a different solar panel), the active time will also increase. However, increasing the capacitor size will also affect the off-time of the system, as the capacitor also takes longer to charge. Because BFree is intended for hobbyists and can be applied in various ways, in many different configurations, we did not exhaustively evaluate all the combinations. Instead, we attempted to provide an overview of the possible combinations and the significant number of parameters that can be tuned in BFree—both in the hardware and the software—to gracefully handle all these different harvesting scenarios considered in Section 2.6.3 and Section 2.6.4.

2.7. USER STUDIES

To complement the evaluation results of BFree presented in Section 2.6, we conducted two user studies[5], both approved by the Human Research Ethics committee of Delft University of Technology, that ask the following **research questions**:

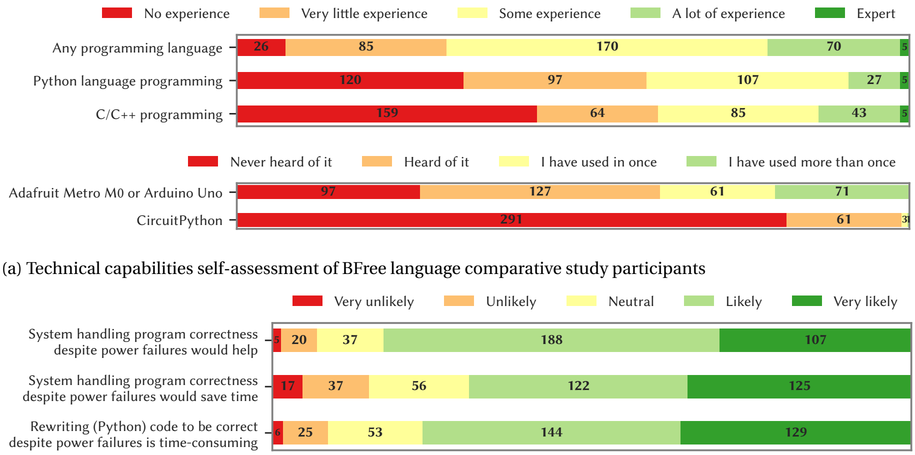
- **RQ1:** Would programming considering intermittency support in (Circuit) Python be *easier* than with state of the art intermittent programming runtimes (written for the C language)?
- **RQ2:** Is the system we built *usable and useful* with regard to making battery-free electronics (powered by ambient sources) for low skill, inexperienced makers?

These research questions help us to understand how novice programmers would best be able to program systems previously only used by experts.

2.7.1. BFREE LANGUAGE COMPARATIVE STUDY

To answer question **RQ1**, we conducted a study on a large group of computer science students of the Delft University of Technology. We asked them to compare Python and C/C++ in the context of application development for intermittent computing. The purpose of this study is by no means to evaluate the BFree programming language (i.e., CircuitPython)

[5] Details about the study, including questionnaire and the detailed answers given by participants of both studies are available in [230].



(a) Technical capabilities self-assessment of BFree language comparative study participants

(b) BFree usability questionnaire responses after exposure to task of finding a bug in a Python code transformed to handle power intermittency

Figure 2.17: BFree language comparative study results (356 participants; the average age of the participant of the study was 19 years, with the youngest participant being 17 years old and the oldest—35 years old). Figure (a) presents study participants' technical capabilities self-assessment, while figure (b) presents answers to a questionnaire provided after participants were exposed to a simple task of finding a bug in a Python code transform to handle power intermittency. A large group of participants with little programming experience and exposure to hobbyist embedded systems unanimously agreed that system that handles program correctness despite power interrupts would help a programmer and save development time. Note: number on each bar in both figures represent the number of responses to each question.

or its sub-components, but to verify the hypothesis that (Circuit) Python programming language (with intermittent execution support engine invisible to the programmer) is easier to use for a novice programmer, compared to state of the art software solutions targeting expert programmers, such as [119, 250, 142, 47] based on the C language.

Participants

As noted earlier, our study was performed among a large pool of students of computer science of Delft University of Technology (details on the student cohort are given in the caption of Figure 2.17). We aimed at students of varying levels of experience with computer programming and with hobbyist-level microcontrollers. Specifically, we have presented the questionnaire during a break of one lecture of the first-year undergraduate "Object Oriented Programming" class, second-year undergraduate "Digital Systems" class, and graduate-level "Analytics and Machine Learning for Software Engineering" class.

Participants have self-assessed their technical abilities and knowledge of the Adafurit Metro M0 and CircuitPython by answering a short questionnaire at the end of the study, which will be described in the subsequent section. Results of the self-assessment are given in Figure 2.17a, with the raw data accessible in [230].

Based on the outcome of the self-assessment, we conclude that majority of students never heard of CircuitPython and had very little experience with using Arduino Uno [21] or Adafruit Metro M0 [5]. Moreover, we conclude (see again Figure 2.17a) that the majority of students have little or no programming experience (especially in C/C++). This level of experience is representative of inexperienced hobbyists beginning with active use of embedded electronics platforms.

Design and Execution of the Study

Firstly, we presented the study participants with a short overview of the challenges of program execution on battery-free devices running intermittently, followed by the link to the questionnaire. The questionnaire started with a short textual introduction to CircuitPython and small embedded microcontrollers, followed by a short description of the intermittent computing problem. Neither during the overview, nor in the introduction to the questionnaire, it was revealed that the authors are the designers of BFree hardware and BFree Python, to avoid any bias in giving answers.

We then proceeded with the exercise that exposed the respondents to the cognitive burden of intermittent computing. That is, we provided a short explanation on how to convert a Python code into one that can run intermittently. This conversion is done by the process of *task transformation*, shown earlier in Figure 2.4, e.g., in the same way as proposed by the state of the art runtimes for C language such as InK [250] or Alpaca [142]. Then the participants were asked to find a bug in the Python code of a simple "variable swap" function, which does not use an extra temporary value, i.e., the code executed

`a = a + b; b = a - b; a = a - b`.

After this step, we presented three different task-transformed implementations of the original Python code of this "variable swap" function. Two of these implementations were 50 lines long, while the third one was 52 lines long. Exact code listing is provided in [230]. Among these three intermittently-executable program options two contained a bug: one option had an incorrect task transition and the other option did not save the program state completely. Respondents had to choose the bug-free version one from the three choices. After a choice was made respondents were not allowed to change their answer.

Result

From all respondents 78.7% (292 out of 371 responses to this question) found the correct answer and spent approximately five minutes on this task. This implies that the respondents were educated enough to answer the core set of questions.

After the participants gave an answer to this exercise we then asked three core questions pertaining to the simplicity of BFree Python. Answers could be provided on the five-level Likert scale. Questions and the results are shown in Figure 2.17b. Therein, we see that an overwhelming majority assessed that developing intermittent programs using BFree Python is *easier* (and also *faster*) than using existing systems (i.e. manual transformation of the code for intermittency protection) that lead to extensive cognitive burden. We consider this a positive answer to the research question **RQ1**[6].

[6] We recall that the task transformation is one of the core ways of preparing code for intermittency for advance embedded systems programmers, see [250, 142].

2.7.2. BFree User Experience Study

We proceed with the second study aiming at answering **RQ2**, by asking a small set of students to experiment with a real BFree platform (that runs a real battery-free application) and provide feedback on BFree's use.

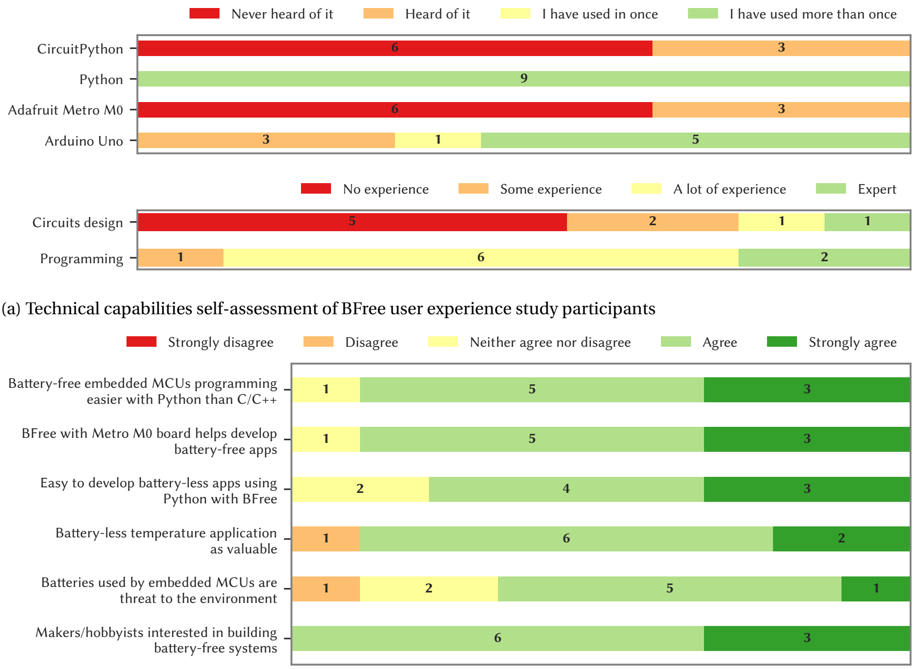
Participants

For the BFree user experience study we have selected nine participants—all university students of either MSc or PhD level (details on this student cohort are given in the caption of Figure 2.18). The study participants have self-assessed their technical capabilities at the end of the study by answering a set of questions. Detailed result of this self-assessment is presented in Figure 2.18a, with the raw data available in the BFree online repository [230]. Analyzing the responses in Figure 2.18a we see that the group was diverse and none of the students had any experience either with CircuitPython or with Adafruit Metro M0 board. This was a desired outcome, as this way the participants were not biased in assessing CircuitPython and the Adafruit Metro M0 board. At the same time all participants claimed to use a regular Python language and most study participants claimed to use another popular hobbyist-grade embedded microcontroller platform—Arduino Uno. Participants of the study were found via email announcements to student groups. No direct professional connection was present between the participants and the authors of this study.

Design and Execution of the Experiment

Each participant was invited to the room specifically prepared for the BFree experience study, as not to influence the participant with the supervisor (or other people's) presence. The participant was asked to sit in front of a desktop PC, standing on a regular office table. This PC was connected via an USB port to Adafruit Metro M0 board, to which the BFree board was attached. On the PC's monitor a web browser was opened that contained a short description of the experiment. The participant was asked to read this description first. This description was accompanied by the same short introduction to intermittent computing as given to the participants of the language comparative study presented earlier, see Section 2.7.1. After reading the description of the experiment, we explained the experiment to the participant again—this time in person—giving each participant the opportunity to ask questions about the experiment process.

The explanation itself was performed as follows. The same PC located in a room had also a pre-loaded program editor with a prepared temperature measurement application written in CircuitPython (code of this program is available in [230]). We asked each participant during the explanation to upload this code to the Adafruit Metro M0 board and subsequently disconnect it from the USB port. Disconnection from the USB port effectively made Adafruit Metro M0 board intermittently-powered by ambient indoor light. Additionally, a table on which the PC and BFree board was located, was also equipped with a light bulb (to imitate strong light source) which participants could turn on and off, controlling the rate of intermittent power supply. Participants were then asked to connect the board back to the PC (making it again continuously powered) to read-out the temperature measurement, which continued from the last moment the Metro M0 with BFree was powered (indicated on a terminal window with an increasing counter).



(a) Technical capabilities self-assessment of BFree user experience study participants

(b) BFree usability questionnaire responses after 20 minute session of experimenting with battery-free temperature measurement app

Figure 2.18: BFree user experience study results (with one female and eight male participants; two participants were 21 years old, one participant–24 years old, two participants–25 years old, three participants—26 years old, and one participant—29 years old). Figure (a) presents study participants' technical capabilities self-assessment, while figure (b) presents answers to the BFree usability questionnaire after hands-on session with a real battery-free application. A diverse group of participants, having no practical experience with CircuitPython and Adafruit Metro M0 board, responded positively to BFree aiding in developing battery-free applications. Note: number on each bar in both figures represent the number of responses to each question.

After this explanation the participant was ready to perform an actual experiment. For this we asked each participant to redo the process we demonstrated, asking first to modify (in any way the participant deemed interesting) the original code of the temperature measurement and then upload in to BFree. In other words we asked the participant to simply "play" with the code and BFree (by increasing or decreasing the light of the light bulb, covering solar panels with hands, plugging USB cable in then reading the measurement and unplugging it again as many times as possible, etc.). We then left the participant alone in the room. Each participant had about 20 minutes for this task. After completing the experiment the participant was asked to answer a short questionnaire related to the experience with BFree. The questionnaire was password-protected and the password was shared only after the participant completed the experiment.

Result

Answers to closed and open questions provided after the completion of the experiment are provided in Figure 2.18b and Table 2.4, respectively.

Based on the individual experience session with BFree majority of participants agreed that BFree helps to develop battery-free applications, making the development of such applications easy (see Figure 2.18b). Participants found the environmental impact of batteries is existent and makers/hobbyists would be interested in using BFree.

At the end of the study participants also suggested a set of applications, deployment scenarios, listed the strong and weak points of BFree, as well as provided remarks on BFree they briefly experimented with. A succinct list of answers is given in Table 2.4. These comments also point us to potential areas for future work.

Based on the above outcome we conclude that BFree is *usable and useful*, which positively answers research question **RQ2**.

Limitations of BFree Experience Study

The experience study has limitations. The main one is the lack of access to the real makers community that would provide a matched and in-depth assessment of our developed platform. Also, a larger participants pool and longer time provided to the participants would result in a more expressive evaluation of BFree. Longer experience would also enable users to develop more sophisticated applications, gaining more knowledge on the limitations of BFree. Finally, participants of this experience study were using the first version of BFree, with many, still unresolved at that time, bugs. We note that the results provided in this chapter (Section 2.6) are based on the much newer version of BFree shied and BFree runtime.

Finally, we remark that this user experience study was executed in mid-February 2020. Since then we could not redo the experience study with a new version of BFree due to COVID-19 restrictions regarding people's presence at our university. Simply, it was logistically hard to control flow of people in and out of the room where the experiment was performed.

Table 2.4: Selected responses to open questions regarding BFree usability made by participants of BFree user experience study. Exact response text from participants were compressed to fit this table. For exact and complete set of answers the reader is referred to [230].

Open question	Responses
What apps would you *develop* with BFree?	• Event counter powered by harvested energy • Weather sensors/wildlife counters in non-populated regions • Crypto-currency mining • Aerospace (when power cuts out due to vibrations rocket state machines could continue right where left off)
What apps would you *deploy* with BFree?	• I would not use this to deploy a product: it feels too much 'hobbyistic' • Cheap wireless sensors everywhere to measure temperature, humidity, light, motion
What are the *strong points* about BFree?	• Easy to use, does not need complicated setup, easy to compile code • No need to think about keeping state in your code • If succeeded making it 100% seamless[1] developers would not need to learn anything to use battery-free feature
What are the *weak points* about BFree?	• Unknown what the approximate power limitations are and how these combine with different hardware • Unclear which operations are safe and what are the risks if the power dies during a task • Harder to debug/test than a standard microcontroller • Doubt about use cases: typically device would either run all the time or it would not matter if it completely resets
Do you have any *remarks* about BFree?	• Stress to audience how it is different from any solar-powered computer • Better documentation on how to make sure power cuts are handled safely • Have smaller form factor BFree boards • People will be able to come up with cool applications for BFree

[1] User reported an issue with device resetting when connected to USB port of a PC, which was corrected with the latest revision of BFree board; refer to explanation in Section 2.7.2.

2.8. Related Work

This chapter merges two visions: the future of embedded computing —*battery-free intermittent computing*—and *sustainable and novice-oriented* programming environments. This work, built on expert-oriented intermittent computing systems, is the *first interpreted runtime for intermittent computation*, and the first such system targeting novice and hobbyist developers.

Intermittent Computing: Devices like the WISP [210], computational RFID tags which harvest energy from RFID reader transmissions, were the first attempts to enable battery-free, energy harvesting embedded computing. Programming for these devices was incredibly difficult and inefficient, so runtime systems that instrumented C programs with checkpoints [136, 203] or transformed these programs into tasks [250, 86, 47] were created. Platforms that leveraged these new runtime systems were developed to increase energy-efficiency and dependability of intermittent computing applications [49, 84]. Tools that enabled deeper introspection into the energy environment [82] and brought command line debugging support [46] further enhanced the ability of *experts* to deploy battery-free systems. None of these systems have addressed novice developers, focusing on performance and energy-efficiency. Even platform such as Flicker [84] mentioned above, that proposes a modular hardware system (similar in spirit to BFree) where individual hardware modules (energy harvesters, computation engine, wireless communication modules, displays, sensors, etc.) can be mixed in various combinations to create an application-specific sensor, requires a very good knowledge of embedded C programming. But more importantly Flicker has no dedicated runtime that handles energy intermittency. Also, Flicker modules are not backward compatible with popular hobbyist-grade microcontroller boards (such as in the case of BFree: Adafruit Metro M0) and does not have enough of memory to store the complete BFree runtime.

Recent technology advances in non-volatile memory storage and ARM Cortex platforms [24] enable us to make the first interpreted language for battery-free devices, and the first focused on enabling the *novice* instead of focusing on performance as in previous approaches. Moreover, as the novice is not as concerned with minute optimizations and performance, but mostly with getting something working, the overhead of our interpreted language approach compared to the state-of-the-art is not detrimental.

Battery-free Applications: We can already list single purpose applications that eschew batteries designed by experts such as wearable health [200], environmental monitoring [83], pervasive or wearable displays [73, 56], wearable authentication and haptics [131], gesture recognition [130], rapidly prototype interactive objects [216, 128], making Skype calls [222], enabling smart spaces [60], and others. This set of applications is enabled by the range of energy available to harvest in various forms, gathering from sunlight, radio frequency transmissions, vibrations, heat [188], human movement [249], and even microbial communities [58]. These applications show the potential for our work, as all of these systems are research products made by experts, often combining advanced techniques spanning computing, electrical engineering, and physics. We believe that BFree will enable these applications to be developed by novices, and potentially ease the development process for experts.

Sensing Platforms: Platforms for wireless sensor networks (WSNs) research and deployment have been developed over the past two decades [90], most notably with the TelosB [193] which was one of the most successful general purpose sensing platforms, and Prometheus [105] the first energy harvesting sensor platform. Energy harvesting WSNs [8] have begun to dominate the sensor world because of decreased maintenance costs and longer deployment lifetimes. Building on these works, synthetic sensors for general purpose embedded computing framed towards the HCI space [122] were developed, along with other platforms meant to increase the applicability and generality of sensors like Hamilton [115], and EcoMicro [124]. None of these platforms can run firmware in Python. Also, none of these platforms can enable battery-free deployment, being built on the assumption that power is continuous and reliable, even if constrained. BFree is specifically engineered to handle the various system difficulties when faced with frequent power failures, enabling the development of robust battery-free, untethered applications by novices.

Novice-Oriented Programming: Developing tools and systems to increase access and applicability of computing to novices has a long history and inspires this work. Well known systems like Logo, Scratch, Processing, and Arduino represent programming environments designed with a low learning curve and directed toward makers, artists, designers, and inexperienced programmers. Platforms like Codeable Objects [100] extend programming to the physical world. Bifröst [154], WiFröst [155] and Scanalog [217] help with debugging complex hardware and software embedded systems. Python Tutor [75] and OmniCode [109] and similar work have sought to teach Python programming to the novice with interactive execution. All of these systems focus on an area where novice programmers are under-served, in this spirit, BFree opens up battery-free and energy harvesting programming to the novice. We believe that interactive programming environments are an interesting area of future research to help novice programmers predict how their battery-free application will perform in the wild.

Computing for Conservation and Sustainability: BFree is motivated by the growing ecological concerns associated with climate change and planetary stewardship that has inspired significant work in computing, HCI, and sustainability [192, 116]. Systems that are designed to encourage energy conservation are tangentially related to this work [218]. For example, EnergyBugs [206] materialize the unseen energy into a tangible object. Persuasive displays [120], eco-feedback systems [64], and other battery-free systems are a response to the increasingly devastating ecological impact of a battery-powered Internet of Things. Other work has advocated for sustainable, responsible approaches to building the smart city [79] and more considered HCI in agriculture [133]. We view BFree as an attempt to democratize mobile and wearable computing with an ecologically responsible view, building on the intellectual underpinnings of work in sustainability. Computing with batteries has offered convenience, but has constrained the design space of ubiquitous computing. As a tool, BFree devices offer ways to visualize energy, show the power of responsible computing practice, and assist in novel computing applications.

2.9. Discussion and Future Work

This work is only the beginning, opening up the possibilities of battery-free devices, for *everyone, everywhere*. We anticipate major research directions to be taken that will enhance the workflow and systems presented here, so that novice developers can be a part of a sustainable future of ubiquitous computing.

A General Platform: Future work could allow the BFree shield to be used without Python, and instead with other languages, both compiled, such as C and Rust, to those interpreted such as JavaScript, and even MakeCode [160] block-based languages. This potential for a general platform that supports many languages is a future goal of BFree such that any person, at any skill level and with any past programming experience can engage in battery-free, energy harvesting prototyping and software development. However, enabling this is not trivial, as checkpointing routines, instantiation mechanisms, and memory management would need to be re-evaluated per language. For example, JavaScript has an even more flexible (often confusing) specification than Python, with a more complex bytecode. We leave this for future work.

Alternative Solutions: Potentially one could write a Python program that stored data right after it was generated, and upon start would read that data and try to resume. In essence, this approach amounts to rolling a custom intermittency solution, running into all the problems described earlier in this chapter. Manually keeping progress is not simple, as it is not known when a failure might happen, and in fact, a failure could happen before the Python runtime even starts or a single line of user code is executed. Another alternative is making your program small enough and simple enough that it will likely finish before the storage capacitor depletes. While this is possible, it is not ideal, as one must first guess how much energy one has and how much energy a line of code costs, becoming very constrained in what one can do. BFree allows makers to program just like they always do and not worry about power failures and recovery.

Garbage Collection: The garbage collector in CircuitPython is a black box: during a startup, required fixed size memory regions are allocated, and then the remaining memory is allocated for dynamic memory requested by the Python application. Because of the black box nature of the algorithm used by the garbage collector it is impossible to find the exact occupied memory regions. For this reason, we are forced in BFree to save all the memory available to the garbage collector during the checkpoint procedure, even when most of it is not actually used by the Python program. A future improvement would be to replace the current garbage collector with one that causes less fragmentation, or adapt the current one to both: (i) expose the memory used by the Python application and (ii) have an efficient compacting method which is called before each checkpoint to reduce the checkpoint size.

Performance: The performance of BFree, especially considering the speed of program resumption after restart, requires further work. Also, we acknowledge that Python code hand-instrumented (but optimized) for intermittent operation *might* be faster than our non-optimized BFree implementation. This observation has been echoed by some of the respondents of our survey, quote:

> *(...) programmers are pretty adamant of being able to control every inch of their code if they need efficiency, and handwritten code will almost always be faster.*

Nonetheless, the aim of BFree is to enable hobbyist or maker programmers, for which the speed of execution is of secondary importance to the usability of the whole system and rapid prototyping ability. We hope to increase both usability and performance to ease access to the battery-free computing domain.

Checkpoint Strategies: BFree provides multiple entry points and adjustable settings for creating checkpoint strategies. This work has only explored the surface level of strategies with just-int-time, periodic, and hybrid methods. In many cases and applications, none of these strategies would be ideal. Significant research space exists to explore and understand checkpoint strategies for interpreted languages. Because an interpreted language has access to the bytecode and other information from the program, which is not always available for bare-metal machine code, more intelligent checkpointing could be envisioned. Moreover, with the larger memory space and compute power of the ARM-based BFree system, more sophisticated checkpoint methods could be explored that take into account history, trends, or even energy aware prediction. These methods are bound to increase the performance of BFree and are a rich area to explore.

Platforms: Our BFree shield is the first proof-of-concept of what is possible with battery-free development for hobbyists and makers. Further hardware extensions could include, *emulators for energy harvesting* (to test the performance of the application), *capacitor size adaptation shields* (to test the code under different energy supply reservoirs), or *multi-sensor shields* (to experiment with different battery-free applications without the need to buy new sensors for each new experiment).

Applications: The success of battery-free intermittent systems solely depends on the richness of applications they can execute. As the field matures, more and more interesting applications arise, from smart protective equipment, to space satellites, to wearable devices that never need charging. That said, not all applications will immediately benefit from BFree, and not all users are even aware of the potential applications or power that battery-free operation gives. As one of the respondents of one of our studies (see Section 2.7) said:

> *The only application I can really think of (...) is long term execution (of) programs. Something like a neural network or genetic algorithm which may run for 24+ hours. It seems kind of niche but I can see the appeal.*

Therefore, more work is needed to think about what “killer” applications for battery-free hobbyist micro-controllers could be, and more work is needed to encourage hobbyists to think beyond traditional boundaries of computing and energy. We view this platform as an enabler for interaction research; such as that extending energy materially [191], exploration of novel interactions that engage with energy, engineering persuasive exhibits or displays, and allowing novel wearables. In the future we hope to see interesting and dynamic applications written and deployed with BFree.

Tooling: Finally, a set of tools is needed, helping makers in experimenting with BFree. These tools have not been a focus of this work. This includes user interface enhancements (for code development, code optimization and code debugging), developer community code management system for BFree hardware and software, and energy introspection.

2.10. Conclusions

BFree allows makers and low skill hobbyists to develop computing and sensing platforms that not only run perpetually on harvested ambient energy but also make them free from batteries and a tethered power supply—thereby reducing the ecological and maintenance cost of traditional embedded systems. BFree’s core innovation lies in developing the first power failure resilient runtime for an interpreted language (CircuitPython) that runs on embedded systems. With BFree, novice programmers can develop CircuitPython applications that sense, compute, learn, communicate, and much more—all without needing a battery or access to an electrical socket. BFree invisibly handles the frequent power interrupts caused by scarce ambient energy so that the programmer does not need to account for them. We evaluated BFree against a battery-powered CircuitPython baseline and showed reasonable overhead of BFree. Our system was tested with actual users, confirming the usability of BFree. Further, we open sourced the code and hardware of BFree as a resource to the community. With BFree, we open a new application area for makers and hobbyists and new realms of possibilities to build sustainable IoT devices. BFree unlocks rapid battery-free prototyping and demonstrates the possibility of converting existing systems to operate intermittently using energy harvested from the environment.

3

Differential Checkpoints

In the previous chapter, we developed a fully featured battery-free hobby platform—based on interpretation—with support for computation and peripherals, showing the steps needed to successfully support the intermittent execution of applications without the need to rewrite the application. However, checkpoint times can be relatively long, and the dominating cause of the checkpoint being so significant is the volatile main memory of the system that needs to be copied to non-volatile memory. The checkpointing cost was less relevant in the previous chapter due to the high cost of interpreting Python programs and the focus on quick prototyping rather than performance.

In this chapter, we extend the techniques introduced in the previous chapter by introducing differential checkpoints that only contain changes made to the volatile memory since the previous checkpoint, which greatly reduces the size of checkpoints because often only a limited amount of memory is modified between checkpoints. However, to safely update the changed data in the checkpoint would require double-buffering, as shown in Section 1.2.1, doubling the required number of non-volatile memory writes. Instead, we introduce a novel patch-based differential checkpointing technique that keeps old checkpoints intact, stores the new data elsewhere in the non-volatile memory, and rebuilds the complete volatile memory state from these patches during restoration—effectively double buffering the data without requiring additional non-volatile memory writes.

This chapter is based on the following publication:
Jasper de Winkel, **Vito Kortbeek**, Josiah Hester, and Przemysław Pawełczak. *Battery-Free Game Boy*. Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT), volume 4, issue 3, pages 1–34, September 2020. https://doi.org/10.1145/3411839.
The accompanying archive containing all the software and data can be found at https://doi.org/10.5281/zenodo.7684870.

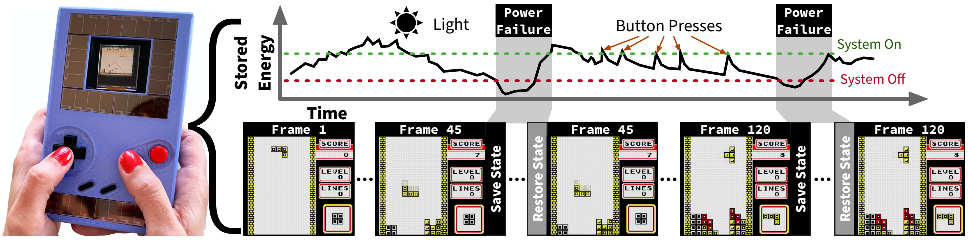


Figure 3.1: Energy harvested from button presses and sunlight powers our custom handheld platform, BFree, running a Nintendo Game Boy emulator which can play classic 8 bit games. BFree efficiently preserves game progress despite power failures, demonstrating for the first time battery-free mobile entertainment.

3.1. INTRODUCTION

This chapter originates from the question: *is it possible to game-on-the-go without batteries*? Batteries add size, weight, bulk, cost and especially inconvenience because of frequent recharging—to any device. Energy can be generated by mashing buttons while gaming, and readily available energy from sunlight is all around us, so why not use this energy for battery-free mobile gaming! Significant challenges in software resiliency and efficiency, hardware operation and energy usage first need to be solved, but would represent a fundamental advancement over non-interactive (and not very fun) battery-free devices that currently exist.

Prototype battery-free devices have been used to make phone calls [223], deployed for machine learning [125], greenhouse monitoring [83], video streaming [163], eye tracking [129] and even built into a robot [250]. However, none of these techniques or prototypes have enabled *interactive battery-free devices*—like a smartwatch, in-place interactive display or even a **handheld video game console**. This is a critical gap in the research around battery-free devices, as these types of *reactive, interactive, and screen-focused* systems are a significant portion of the current and anticipated smart systems.

In this chapter we focus specifically on this ignored part of the battery-free device ecosystem, *mobile gaming*, and use this application to elucidate the essential challenges that must be explored to get us to a future where reactive and user facing applications can also be battery-free. From a market perspective there is a deep need to explore this area. The global gaming industry is massive and generates unprecedented revenues, which already exceeded 100 billion USD in 2016 [166]. Handheld console game sales constitute a large portion of the industry [166].

To enable these types of devices, mobile gaming platforms must be re-imagined at the system and interactivity level. The main challenge is that energy harvesting is dynamic and unpredictable. This is intuitively apparent when considering a solar panel; a cloud, the time of day, weather conditions, movement and orientation of the panel, and even the electrical load all change the amount of harvested energy. Because of this dynamism, these devices run out of energy and lose power frequently, only *intermittently* computing with the device having to wait seconds or minutes to gain enough energy to turn back on. This long recovery process can be energy and resource intensive, causing responsiveness delays. Worse, it can leave the game in an inconsistent state. Naturally, going through this entire re-loading process (from the loading screen of a game to starting play) every time is burdensome, so just blindly replacing batteries in a game console with an energy harvester is not enough to ensure smooth game operation.

To address this challenge this chapter presents a *framework of solutions based around energy-aware interactive computing* and a *reference implementation of a popular game console*—8 bit Nintendo Game boy [171, 50]—as a demonstration, see Figure 3.1. To reduce the unpredictability of energy harvesting, we take advantage of mechanical energy generated by "button mashing" of the console, harvesting this energy generated by actually playing a game on a handheld, and using it, along with solar panels, to power all operations. We design the system hardware and software from the ground up to be energy-aware and reactive to changing energy situations to mitigate the issues caused by frequent power failures. Specifically, we design a technique to create minimal *save games* that can be quickly created, updated, and saved to non-volatile memory before a power failure, then quickly restored once power returns—for example mid-jump in a platform game—all this despite the device fully losing power.

Contributions. In this chapter we present a practical, usable mobile gaming device, *Energy Aware Gaming* (abbreviated as *ENGAGE*). The first intermittently powered interactive gaming platform. Our contributions follow:

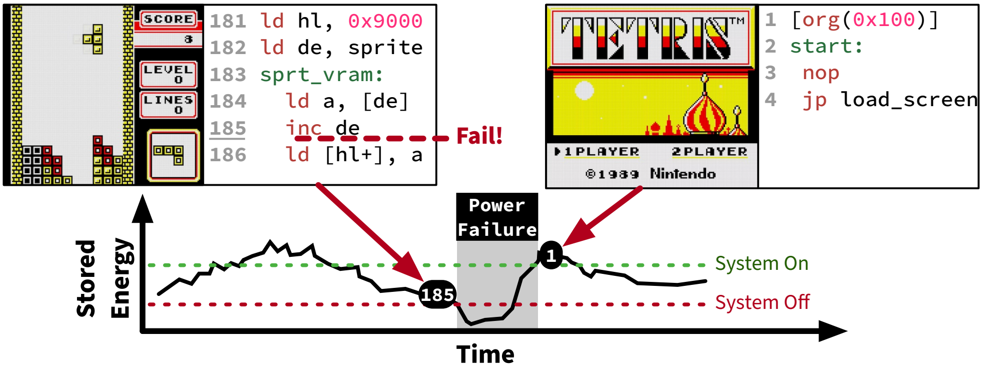
1. We introduce the concept of intermittently powered mobile gaming;
2. We develop an approach to failure resilient, memory-efficient, fast, whole system save games for interactive, display driven devices. A just-in-time differential checkpointing scheme is used based on the concept of tracking changed memory in *patches*;
3. As a stress test and demonstrative exercise of the promise of battery-free gaming, we use these systems and hardware to develop a full system Nintendo Game Boy emulator which plays unmodified Game Boy games despite power failures.

This chapter is a reduced version of the publication *Battery-Free Game Boy* [55]. It focuses on the software support needed to efficiently support intermittent computing on a batteryless handheld gaming device through a novel patch-based checkpointing approach. The remaining components of the original publication, e.g., the hardware, harvesting, emulation, input management, and screen handling, were developed as part of a collaborative project and are *not* part of this thesis. Sections of the original publication focussing solely on these components are excluded from this chapter. However, some components unrelated to this thesis are included in this chapter as they provide the motivation and use case for the introduced checkpointing mechanism. More details on the excluded components can be found in the original publication [55].

The checkpointing mechanism introduced in this chapter was designed as part of the ENGAGE platform, which powers the Battery-Free Game Boy. However, the introduced techniques and the developed framework—MPatch—are general and can be used on other hardware to create efficient intermittently-powered systems.

3.2. CHALLENGES

The goal of this work is to develop the systems and hardware foundations for battery-free mobile gaming. This is motivated by two reasons: (i) the enhanced availability and usability of a platform that never needs to be recharged or plugged in—making the platform more convenient for the typical user, and more accessible for everyone, and (ii) the need for alternative and sustainable forms of entertainment—a nod to the various



```
181 ld hl, 0x9000
182 ld de, sprite
183 sprt_vram:
184    ld a, [de]
185    inc de
186    ld [hl+], a
```

```
1 [org(0x100)]
2 start:
3  nop
4  jp load_screen
```

Figure 3.2: Dynamic energy harvesting causes voltage fluctuations which cause frequent power failures. Shown is what would typically happen if a battery was removed from a Game Boy and replaced with solar panels. The game would play until energy is lost (i.e. at line 185) and then restart at the loading screen. *Intermittent computing techniques* seek to make it such that after the power failure, line 186 is then executed proceeding from the exact system state as before the failure.

industry consortia such as *Playing for the Planet* [197] which aim to reduce the gaming industries ecological impact. A battery-free handheld game console reduces ecological costs and disappointment, as it is always ready to be picked up and played *without needing to be recharged.*

Numerous explorations of battery-free smart devices address the calls for sustainable and carbon-neutral electronic device interaction and electronic design and computing [116, 36, 149, 243, 114] while preparing human-interactive electronics for the "post-collapse society" [228]. None of the existing state-of-the-art intermittently powered systems have yet explored the question of mobile handheld entertainment, going beyond the simple forms of battery-free gaming devices demonstrated commercially in the early 1980's [51]. This is because making such a device is challenging due to complex system difficulties stemming from frequent power failures, listed below.

Challenge 1: Unpredictable Energy Harvesting. Environmental conditions change, and this is exacerbated by mobile gaming. When players move from place to place, most forms of ambient energy change drastically (for instance, by moving from sun to shade), or increasing distance from a radio frequency power source. Without a more predictable source of power, it is hard to envision being able to play continuously without a battery.

Challenge 2: Keeping Track of System/Game State. Maintaining the state of computation, let alone game state, through power failures from intermittent harvested energy is hard [135, 158]. Many software frameworks that support computation progress despite these power failures exist, saving state in non-volatile memory like FRAM and then restoring state after power resumes (see Figure 3.2), such as TICS [119], TotalRecall [242], and many others. Most systems trade memory efficiency for performance, this approach is the opposite of that needed for gaming, where a display buffer and numerous sprites and large game state variables must be saved, requiring high memory efficiency.

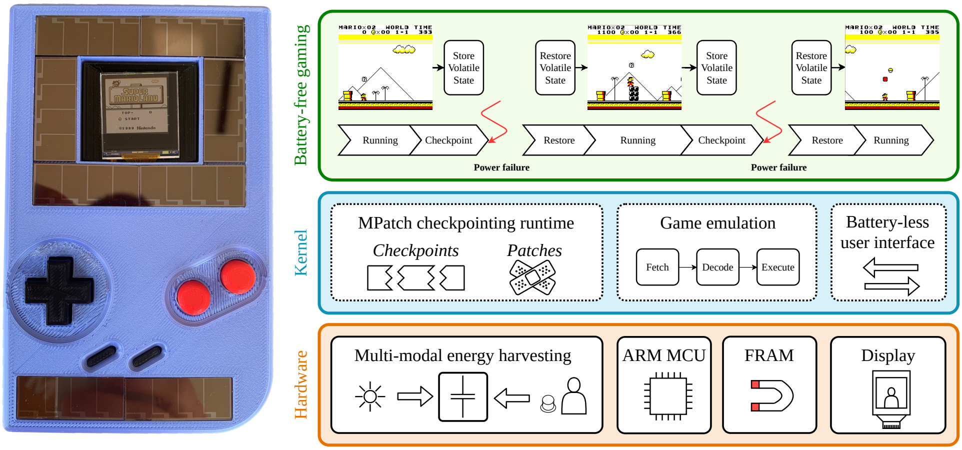


Figure 3.3: ENGAGE hardware platform (left) and its internal architecture (right).

Challenge 3: Enormous Variability of Games. These previous issues are compounded by the huge variability of games, both in terms of memory size, number of sprites, actions, difficulty, and even number of button presses per second. Each game is unique, and could pose difficulties when creating a general battery-free solution.

Challenge 4: Gaming's High Computational Load. To date, no full system emulation of any complex system has been attempted on battery-free, intermittently computing devices. Games and gaming platforms require more performant processors even when running natively—when running in emulation, this is compounded. All existing popular runtimes for intermittent computing are based on Texas Instrument's mixed-memory MSP430 MCU [226], which is an order of magnitude slower than the fastest ARM MCU on the market. To meet the high computational load of games, a practical runtime for ARM microconrollers must first be built.

Challenge 5: Realistic Demonstration. The over-arching goal is to play a real, unmodified, video game on a battery-free console that everyone around the world knows (like Tetris)—in other words to be able to execute preexisting game code (or any existing code for that matter), *not to design a custom game* only to demonstrate the potential of battery-free gaming. This could be possible only when all the above challenges are addressed.

To tackle above *Challenge 1–5* we took one of the most popular gaming consoles of all-time [170]—the original 8 bit Nintendo Game Boy [171, 50]—and redesigned its hardware-software, powering gameplay from the solar panels and button presses of the user, building the first ARM based intermittent computing hadware and runtime system, and doing the first full system emulation of a real world platform (Nintendo Game Boy) with intermittent computing techniques.

3.3. Battery-free Handheld Gaming

We designed the *Energy Aware Gaming* (ENGAGE) platform as proof by demonstration that the discussed challenges could be overcome. The design and architecture of the ENGAGE platforms are shown in Figure 3.3. The *ENGAGE hardware* is the size and form factor of a Nintendo Game Boy, it is built around (i) user input via mechanical energy harvesting buttons (on the A, B, and D-Pad of the original Game Boy), (ii) a display, (iii) a slot for Game Boy game cartridges to be inserted, and (iv) energy harvesting circuitry from solar cells and the buttons which store energy in a small internal capacitor. The *ENGAGE kernel* consists of (i) a patch-based differential checkpointing system (denoted as MPatch) which handles low level memory movement and automatically saves and restores the state of the entire system by efficiently moving necessary data to non-volatile memory (FRAM) and back (SRAM), and (ii) an extensively rewritten full-system Nintendo Game Boy emulator, which can run unmodified Game Boy games. ENGAGE is the first full system emulation, and the first gaming platform built for battery-free, energy harvesting, intermittently powered computing devices.

Usage and Impact. We released the hardware designs, firmware and software as open-source repositories on Github [2]. We target a broad audience with our platform.

3.3.1. Key Ideas

Existing handheld gaming devices rely on large batteries because they need continuous high power to support high compute load, energy cost, and reactivity. We want to enable playing retro 8 bit console games, such as *Tetris* and *Super Mario Land*, on a battery-free console that is similar in user interface and gameplay to the original Nintendo Game Boy. Removing the battery and only using harvested energy causes intermittent operation, which leads to the challenges discussed in Section 3.2. The ENGAGE platform design navigates these challenges based on four key ideas.

Track and Checkpoint Minimal State at the System Level. We must handle intermittent power failures to maintain the state of play. Unfortunately, in games large amounts of memory is moved back and forth to the display, often in the form of sprites, with computation happening in between. Naively checkpointing the entire system state would be impractical, significantly increasing the latency of operation. We note that while large memory movements happen, the changes in these memories are often small, meaning we can reduce checkpoints to only the changed memory, save that state just in time before a power failure, and then restore that state and resume game play. This *addresses* **Challenge 2**, **Challenge 3** and **Challenge 4**.

Use Processor Emulation to Play Retro Games. While ENGAGE could be used for custom gaming libraries made specifically for intermittent operation, the more challenging and interesting problem is full system emulation enabling the play of thousands of existing games, and even home-brewed games. This also allows us to explore and understand the variability of real world games. This *addresses* **Challenge 3** and **Challenge 5**.

Speedup Intermittent Computing. We embrace ultra low powered, high performance ARM Cortex microcontrollers, and external FRAM memory to speed up computation. While a seemingly trivial technology advancement, with this approach we increase compute speed but increase our I/O burden for checkpointing, as the traditional MSP430 FRAM-enabled MCUs have internal FRAM memory accessible at CPU speeds. This is a different tradeoff space than any other intermittent hardware platform [54, 84, 49]. This *addresses* **Challenge 2** and **Challenge 4**.

3.3.2. ENGAGE Full System Nintendo Game Boy Emulator

A key part of our approach is running a full system emulation on ENGAGE hardware. To be able to run Nintendo Game Boy games an emulator is used to emulate the instruction set of the Game Boy processor, i.e. 8 bit 4.19 MHz custom-built Sharp LR35902 MCU with a processor closely based on the Z80 instruction set [50]. An emulator reads bitcode instructions and executes them in native code, mimicking the emulated CPU as closely as possible to ensure it executes in an identical fashion to the emulated CPU. With the restrictions of battery-free systems additional scenarios are introduced that normally do not exist, such as the loss of power while running a game and then attempting to restore the system to the state it lost power. Additionally, emulation efficiency is of critical importance in regards of power consumption.

The emulator allocates non-volatile and volatile Game Boy game memory within the memory space of ENGAGE, removing the need to keep cartridges continuously powered. Only upon loading a new game is the cartridge interface used to retrieve the non-volatile game data.

3.3.3. Gaming Through Power Failures

ENGAGE is protected from the loss of progress by the custom-designed runtime that guarantees data consistency despite power interrupts. The goal of this runtime is to save (i.e. to checkpoint) the current state of the emulator. This entails the current volatile memory content and the registers of both the host processor and the emulated system. Doing this will allow the system to continue execution from this point as if a power failure never happened.

There are multiple intermittent runtime systems which can be broadly divided into two classes: (i) those that use a *special (C program) code instrumentation* to guarantee the correctness of computation despite power interrupts and (ii) those that use a *special version of the checkpointing*, of which a subset is designed for systems that use volatile memory—such as SRAM—as their main memory, and use a separate non-volatile memory that contains the checkpointed data. While designing ENGAGE we chose to use a checkpoint based system to allow emulation of arbitrary game code. We did not consider task-based runtimes simply because they are too complex to comprehend by a programmer and more difficult to design than a checkpoint-based system; see related discussion on this topic in [119]. But first and foremost, task-based system cannot execute a binary (machine) code, which ENGAGE is mostly executing.

The main requirement for ENGAGE is responsiveness. Hence the checkpointing system needs to be as lightweight as possible. Naturally all of the checkpoint systems have some overhead, so when searching for a good solution we would like to minimize checkpoint size as much as possible—resulting in minimum overhead from data restoration. Checkpointing the entire system state, including game, and emulator, would be impossible. One core idea, proposed first by the DICE runtime [12], is *to checkpoint only*

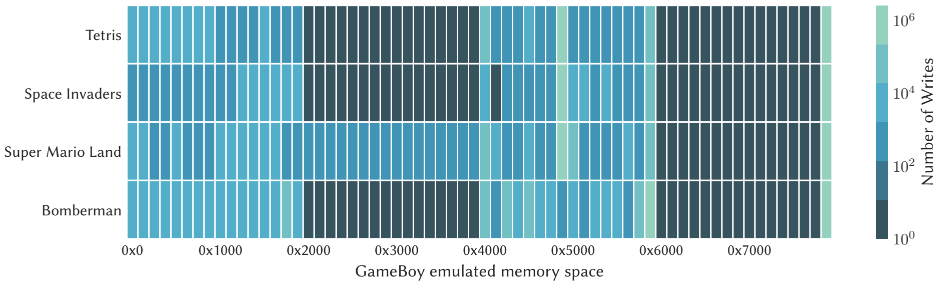


Figure 3.4: Memory writes heat map of four popular 8 bit Game Boy games for one minute of play. Writes tend to cluster in a few large regions; tracking and checkpointing these regions would allow for performant intermittent execution. Note the log-scale of the number of writes.

parts of device memory that have been changed since the last checkpoint. To check whether this idea applies to battery-free handheld gaming system we have performed a simple experiment. For four example Nintendo Game Boy games: (i) *Tetris*, (ii) *Space Invaders*, (iii) *Super Mario Land*, and (iii) *Bomberman*, we have measured to which memory regions of an MCU each game was writing during one minute of game play. The result is presented in Figure 3.4. Indeed, we see that memory writes are very unevenly distributed for each game, hinting that such approach, which we broadly denote as *differential* checkpointing, is well suited for our ENGAGE needs.

The checkpoint runtimes, including differential ones, can be further divided into two unique classes: (i) *corruptible* and (ii) *incorruptible*, as discussed before in Chapter 1.

- **Corruptible Checkpoint:** Such systems copy the current state of the MCU (memory, registers, etc.) to a predetermined location in non-volatile memory. This location is the same every time, as this eases the runtime development and reduces the non-volatile memory requirements. However, it is required that a checkpoint operation must guarantee to complete, otherwise part of the previous checkpoint may be overwritten with the current checkpoint[1]. Often these corruptible runtimes include a check whether a checkpoint was completed successfully, otherwise they start the program execution from the beginning. Such systems require exact prediction of the energy (required to perform a checkpoint) and the energy currently consumed by the complete system (to be able to guarantee that a checkpoint is only performed when its completion can be guaranteed). Such a requirement is *unrealistic* for a computing platform, such as ENGAGE, that includes many peripherals and components all connected to the same energy buffer, as correctly predicting the required energy—even the CPU alone—is difficult;
- **Incorruptible Checkpoint:** Such systems take a different approach: at all times they *guarantee* that there is a valid checkpoint which can be restored. This means that a new checkpoint will never overwrite part of the previous checkpoint in non-volatile memory. Such a guarantee is often implemented through double-buffering.

[1] If the system were to run out of power during the creation of a checkpoint, with the next checkpoint restoration a corrupt state will be restored leading to undefined behavior—thus to a corrupt system.

Table 3.1: Comparison of MPatch with state-of-the-art intermittent checkpointing runtimes.

System	Incorruptible	Differential	Just-in-time	Volatile main memory	ARM support
Mementos [203]	Yes ✓	No ✗	Yes ✓	Yes ✓	Yes ✓
Hibernus++ [30]	No[1] ✗	No ✗	Partially —	Yes ✓	No ✗
QuickRecall [103]	No ✗	No ✗	Yes ✓	No ✗	No ✗
Chinchilla [143]	Yes ✓	N/A	No ✗	No ✗	No ✗
Rachet [235]	Yes ✓	N/A	No ✗	No ✗	Yes ✓
HarvOS [35]	No[1] ✗	No ✗	Yes ✓	Yes ✓	Yes ✓
TICS [119]	Yes ✓	N/A	No ✗	No ✗	No ✗
TotalRecall [242]	No[1] ✗	No ✗	Yes ✓	Yes ✓	No ✗
Elastin [44]	Yes ✓	N/A	No ✗	No ✗	No ✗
DICE [12]	No [1] ✗	Yes ✓	Yes ✓	Yes ✓	Yes ✓
MPatch	Yes ✓	Yes ✓	Yes ✓	Yes ✓	Yes ✓

[1] These systems require perfect energy prediction to not get corrupted. Any changes in, for example, capacitor size [44], power consumption due to peripheral use, or harvested energy, **will** lead to incorrect predictions and therefore **corruption**.

As of the time of publication, there were no known incorruptible differential checkpoint systems, and just one corruptible differential checkpoint system, DICE [12], also refer to Table 3.1 where existing intermittent runtimes are qualitatively compared from ENGAGE requirements point of view. Therefore, to realize a working ENGAGE we developed a new checkpointing runtime, denoted as MPatch, that performs incorruptible differential checkpoints. The proposed runtime is aided by a new concept of *patch checkpointing*, discussed below.

MPatch—a Patch Checkpointing Intermittent Runtime. Memory is constantly being modified during the execution of a program. However, as Figure 3.4 clearly illustrates, it is unlikely that during an on-period of any intermittently powered embedded system, including ENGAGE, all memory is modified. Therefore, when creating a checkpoint containing all the known or active memory regions of the system, one will inevitably copy memory locations that have not changed since the last checkpoint.

It is thus desirable to copy as little of the (embedded) system state as possible while keeping the checkpointing incorruptible. The most fundamental method to do this efficiently is to track which memory regions have been changed since the last checkpoint, in other words, to see memory modification *differences* in-between checkpoints. As mentioned earlier, the only checkpoint runtime that has employed this form of differential checkpoint so far was DICE [12], see again Table 3.1. It is, however, difficult to apply the techniques used by DICE while maintaining an incorruptible system (that uses double buffering). Specifically, assuming that only one of the buffers is active, if part of the checkpoint resides in the previous buffer, and yet another checkpoint occurs, then it is impossible to keep the incorruptibility trait with DICE without still copying all checkpoint data between the two buffers. Therefore, to achieve differential checkpointing that is incorruptible, a new system has to be designed, which resulted in MPatch.

MPatch Just-in-Time Checkpoints. As we have shown in Figure 3.4 not all of the emulator and display memory is written to at every MCU clock cycle. Hence we only checkpoint

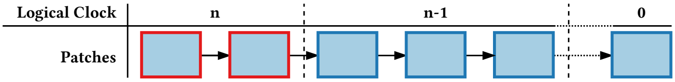


Figure 3.5: MPatch stage operation. Patches outlined with red are staged, but not committed. Patches outlined with blue signify committed patches.

the modified memory regions, which we denote as *patches*. Then, we monitor the voltage level of the storage capacitor, as in other existing runtimes, e.g. [30, 12, 242, 103] and only checkpoint the state when nearing a power failure—we call this *just-in-time checkpoint*. We purposefully do not perform checkpoints at an interval timer: game players are susceptible to lagging in a game. Hence interval-based checkpointing (which introduces frequent fixed-interval delay) is not desirable.

Patch Handling. A patch is a non-volatile copy of a *consecutive* region of volatile memory that has changed since the last successfully created checkpoint. As different memory regions are modified during execution, multiple patches of different memory sections might be required for a complete checkpoint. During the restoration, the most recent patches (in combination with the pre-existing patches) are used to restore the volatile memory to the state it was in during the last checkpoint. By only storing the modified regions the checkpoint time is significantly reduced, as often only a small part of the memory is changed between the two consecutive checkpoints (we will investigate this further in Section 3.5).

As with traditional checkpoint-based systems that use double-buffering, an atomic variable n, determines which of the two buffers should be used to restore the system in case of a power failure [203, 119]. This variable n is changed—often incremented—to mark the completion of a checkpoint. The requirement on n is that for its increment, $n+1$, it holds that $(n \bmod 2) \neq (n+1 \bmod 2)$. MPatch patch management is also built around the atomic variable. However, MPatch extends the function of this variable to act as a *logical clock*, with the additional requirement that $n \neq n+1$.

We now define three fundamental patch operations (i) *Patch Stage*, (ii) *Patch Commit*, and (iii) *Patch Restore*.

- *Patch Stage:* When a patch is created, the required amount of non-volatile memory is allocated and the volatile-memory is copied to the patch. Next, the patch is *staged* by signing it with the current logical clock n added to the front of the *patch chain*, i.e. the list of patches, ordered from newest to oldest, that will be applied during restoration. Staged patches are outlined in red color in Figure 3.5. While a patch is staged it will be discarded if a power failure (and thus a restoration procedure) occurs.
- *Patch Commit:* When the logical clock n is incremented, all previously staged patches will become *committed*. These patches are outlined in blue in Figure 3.5. Committed patches will be considered during the *patch restore* procedure.
- *Patch Restore:* When ENGAGE inevitably fails due to a lack of energy, it should be restored to the last completed checkpoint. Patches hold copies of consecutive volatile memory regions and are linked together to form the patch chain. This moves the complication of deciding what *part* of the patch to apply, if any, to

the restore operation. To reconstruct the state of the most recent checkpoint the (partial) content of multiple patches has to be combined. This reconstruction, due to the implicit ordering in the patch chain, starts from newest to oldest. For each patch, only the parts that were not already applied during the current restore operation are copied to volatile memory, as illustrated in Figure 3.6. In contrast, for a traditional incorruptible checkpoint runtime, restoring a checkpoint means reading the logical clock n and copying the checkpoint content from the selected buffer to the corresponding volatile memory and registers.

3.4. ENGAGE IMPLEMENTATION

We proceed with the implementation details of ENGAGE. Additional information regarding the hardware implementation can be found in the original publication [55]. All hardware, software and tools, as well as documentation for ENGAGE are publicly available via [2].

3.4.1. ENGAGE HARDWARE

We built a handheld, energy harvesting, battery-free hardware platform to enable the development and testing of our approach to battery-free mobile gaming. ENGAGE is built using the following components.

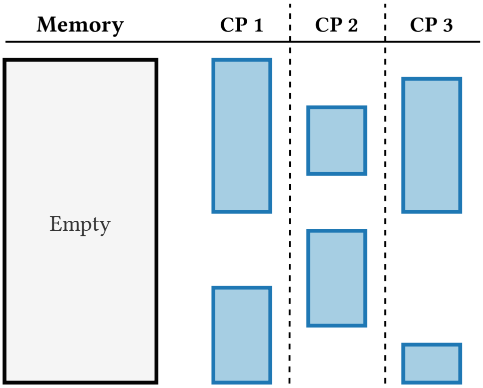
PROCESSING AND MEMORY

Stemming from the requirements (Section 3.2), for compatibility and popularity reasons, we build our ENGAGE around an ARM MCU architecture. However, none of the ARM architecture MCUs we are aware of contain on-chip fast, byte-addressable non-volatile memory—such as FRAM—serving as main memory. Only slow and energy-expensive FLASH memory is present. Therefore we equip our battery-free console with external dedicated FRAM. Central to ENGAGE is the *Ambiq Apollo3 Blue ARM Cortex-M4 MCU* operating at a clock frequency of 96 MHz [16], chosen for its good energy efficiency. The Apollo3 runs the Game Boy emulator and MPatch software. External *Fujitsu MB85RS4MT 512 KB FRAM* [65] is connected through SPI to the MCU providing a fast and durable method of non-volatile storage for patch checkpoints.

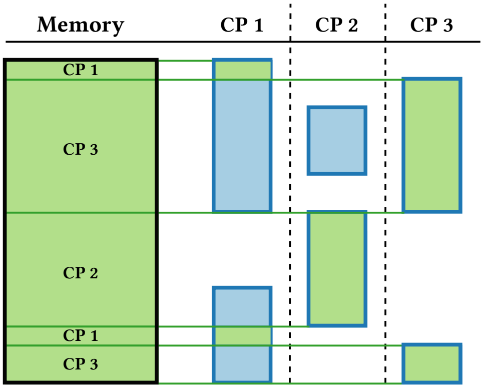
3.4.2. ENGAGE EMULATOR IMPLEMENTATION

As many Nintendo Game Boy emulators have already been written we have decided not to build yet another one and relied on the existing emulator implementation that targets a different MCU. Specifically, to run with ENGAGE we extensively modified and rewrote a pre-existing freely-available implementation of original Nintendo Game Boy emulator targeting a STM32F7 MCU [33]. All the modifications to this emulator, enabling to reproduce our work, are part of our open-source repository freely available to download from [2].

ENGAGE Memory Configuration. The Apollo3 ARM Cortex-M4 features flash and SRAM as on-board memory, where the Flash memory contains all the code (MPatch and Game Boy game emulator code) and non-volatile game data copied from the Game Boy game cartridge. SRAM contains the memory of the whole ENGAGE platform and the volatile



(a) MPatch memory **before** restoration.



(b) MPatch memory **after** restoration. The restore sequence applies only the parts of patches that are required to reconstruct the memory.

Figure 3.6: MPatch patch restore procedure after three successful checkpoints (CP). For the ease of illustration, we assume that the memory is initiated as empty; blue rectangles depict patches that have been successfully committed to non-volatile memory and green rectangles signify the parts of the patches that are applied during restoration.

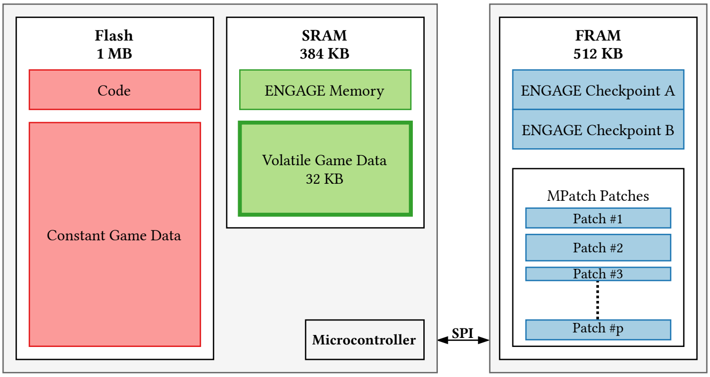


Figure 3.7: ENGAGE physical memory structure. Constant game data is executed from Flash with its volatile memory in SRAM, avoiding overhead from accessing the external FRAM. Only checkpoints and patches are stored in external FRAM.

game memory—both separated from each other. Two buffers, *Checkpoint A* and *Checkpoint B*, for double buffering the core content during checkpointing, as well as all patches created by MPatch reside in external FRAM. The complete memory map of ENGAGE is presented also in Figure 3.7.

3.4.3. MPATCH IMPLEMENTATION

CORE CHECKPOINTS

MPatch is built upon a basic double-buffered checkpoint scheme which we denote as the **core checkpoint** system. The core checkpoint encompasses all the emulation management logic of ENGAGE, *except for the emulated game memory*, which is checkpointed using patches as described in Section 3.4.3. Specifically, the core checkpoint system checkpoints the .data, .bss and active stack sections of the MCU's volatile memory as well as the registers of the MCU, as can be seen in Algorithm 1. All this is double-buffered in the external non-volatile memory of ENGAGE. Naturally, this means that for every byte of volatile memory in the checkpoint, we need twice as many bytes in non-volatile memory. We remark that not all memory of ENGAGE is checkpointed. Specifically, we do not checkpoint memory buffers required for peripherals (as the peripheral state needs to be re-initialized every ENGAGE reboot). The restoration of a checkpoint will restore the state of the system to that of the last successful checkpoint. If the system does not experience a *first time boot*, the default memory initialization step (which traditionally runs before any user code) will be skipped. After this, the steps listed in Algorithm 2 are performed to continue executing as if no power failure had occurred. In line 3 of Algorithm 2 the MPatch patch restoration process is started to restore the emulated game memory which will be discussed further in Section 3.4.3.

We designed the core checkpoint system from the ground up, implementing special keywords enabling the exclusion of certain volatile memory parts from a checkpoint. Also, the core checkpoint provides hooks for every stage of the checkpoint for ease of extension, which is required to incorporate patches from MPatch.

Algorithm 1: Checkpoint Creation

```
Procedure CheckpointCreate():
    CoreCheckpoint()                    // Checkpoint memory not manged by MPatch
    PatchesCreate()   // Create and stage patches; see Section 6 and Algorithm 3
    RegisterCheckpoint()                               // Checkpoint the CPU registers
    RestorePoint()               // Continuation point after a restore operation
    if isNotRestore() then
        CheckpointCommit()              // Call function that commits the checkpoint
```

Algorithm 2: Checkpoint Restoration

```
Procedure CheckpointRestore():
    CoreCheckpointRestore()                  // Restore memory not manged by MPatch
    PatchesRestore()               // Restore committed patches; see Algorithm 5
    PeripheralRestore()                                     // Restore peripherals
    RegisterCheckpointRestore()                       // Restore the CPU registers
    RestorePoint()              // Continue at the restore point; see Algorithm 1
```

PATCH CHECKPOINTS IMPLEMENTATION

The emulated memory, i.e. the memory used by the Game Boy games, is a region in SRAM accessed only by emulated read and write instructions from the emulator. Leveraging this fact makes tracking modification to the emulated memory straightforward, and doing so has little impact on the overall performance. ENGAGE tracks these modifications, and when a checkpoint is created, this information is used to create the required patches as can be seen in Algorithm 3. Tracking of these modifications is done using the memory protection unit of the MCU. Upon writing to a region of emulated game memory, the memory protection unit triggers an interrupt allowing the memory region to be marked as modified. After a region is marked as modified the interrupt for the region is disabled. This results in an efficient method of tracking memory writes since the introduced overhead is only present during the first write after a reboot. The memory protection unit features eight regions which each have eight sub-regions, for a total of 64 sub-regions. We equally divided the memory space of the emulated Game Boy memory between these sub-regions resulting in patches containing 32 kB / 64 = 512 B of emulated memory.

Content of a Patch. In addition to the copy of a volatile memory region, a patch contains accompanying metadata required to successfully manage and restore a patch. This metadata is: (i) the *value of the logical clock n* from when the patch was staged, (ii) the *interval of the volatile memory* that is stored within the patch, (iii) the *next patch* in the patch chain, (iv) the *metadata to build an augmented interval tree* to speed up the restoration procedure, which will be discussed later in this section.

Patch Allocation. Patch sizes are allowed to differ. Therefore some form of dynamic memory allocation is required. This brings challenges, as dynamic allocation leads to fragmentation, which is undesirable in an embedded system. Therefore patches are allocated using a *fixed-size block allocator* [113]. These allocated blocks are chained together to create enough room required to store the volatile memory within the non-

Algorithm 3: Patch Creation

```
Procedure PatchesCreate():
    while $p \leftarrow ModifiedMemory$ do        // For each of the modified regions of memory
        PatchStage($p.address_{start}, p.address_{end}$)    // Create and stage the patch; see Algorithm 4
```

Algorithm 4: Patch Staging

```
Procedure PatchStage($address_{start}, address_{end}$):
    $patch \leftarrow$ AllocatePatch($address_{start}, address_{end}$)    // Allocate memory for a patch
    PatchCreate($patch$)  // Copy the volatile memory region into the non-volatile patch
```

volatile blocks. Each block contains: (i) a link to the *next block* in the chain, and (ii) a link to the *next free block* in the chain. All blocks are stored and managed in non-volatile memory. This creates challenges when trying to synchronize its non-volatile and volatile state. If these are not kept in sync, blocks will be lost, and the system may become corrupt. Additionally, write-after-read (WAR) violations [52] should be avoided when interacting with the non-volatile state. These two separate links in a block are required to eliminate one of these WAR violations, and this violation could also be eliminated by introducing forced checkpoints, as inserting a checkpoint will break a WAR violation [52]. The total memory overhead of a patch in ENGAGE as it is currently implemented is 29 B. By excluding the interval tree required for the metadata, this can be reduced further to 17 B, but this would require an additional dynamic memory allocator to allocate this memory in volatile memory during a restoration (e.g. standard heap). For the final version used in ENGAGE, this was deemed undesirable, and therefore we integrated the interval tree metadata within non-volatile patches.

Patch Restoration. Restoring patches involves first discarding all staged—but not yet committed—patches, and then iterating through the patch chain while applying only the regions of a patch that were not previously applied during the restoration process. To keep track of the regions of volatile memory that were already restored we maintain an augmented *interval tree* during the restoration process. After a patch is applied, its range is added to the interval tree, and when a patch is applied, the interval tree is queried to detect overlaps. If there are no overlaps, the path is applied (i.e. written to the corresponding region in volatile memory). However, if the patch region overlaps with any region in the interval tree, the patch is split-up and all sub-patches are attempted to be applied. The complete algorithm for patch restoration is shown in Algorithm 5, with its accompanying patch apply algorithm shown in Algorithm 6.

Memory Recovery. One of the features of MPatch is its constant time patch creation while being incorruptible. However, patches that are no longer useful, i.e. that will not be applied during restoration, should be deleted. To avoid WAR violations, removing a patch (reclaiming its memory), consists of two operations. Firstly, the patch is freed, and secondly, the patch is deleted. Between these two operations, a checkpoint of only

Algorithm 5: Patch Restoration (note: $low(p)$, $high(p)$ denote the low, high component of range p, respectively)

```
Procedure PatchesRestore():
    DiscardUncommitted()        // Call function that discards uncommitted patches
    while $p_{apply} \leftarrow next(PatchChain)$ do        // Extract next patch from patch chain
        PatchApply($p_{apply}, low(p_{apply}), high(p_{apply})$)        // Apply patch; see Algorithm 6
        IntervalInsert($low(p_{apply}), high(p_{apply})$)    // Insert the patch range into the
         interval tree
```

Algorithm 6: Patch Apply (note: $low(p)$, $high(p)$ denote the low, high component of range p, respectively)

```
Procedure PatchApply():
    if $p_{overlap} \leftarrow$ IntervalOverlap($low, high$) then    // Check for overlapping region in
     interval tree
        if $low < low(p_{overlap})$ then
            PatchApply($p_{apply}, low, low(p_{overlap}) - 1$)    // Recursively apply patch with a
             new, partial, range
        if $high > high(p_{overlap})$ then
            PatchApply($p_{apply}, high(p_{overlap}) + 1, high$)    // Recursively apply patch with
             a new, partial, range
    else
        Write($p_{apply}, low, high$)  // Write patch content between low and high to the
         volatile memory
```

the MPatch management state is made containing patch and block allocation related metadata. During the deletion of a patch special care is taken to avoid WAR violations when modifying non-volatile memory in the patch chain. Memory recovery is not needed during every time a checkpoint is created or restored, is automatically done when there is no more non-volatile memory available to allocate a patch.

3.5. ENGAGE EVALUATION

We built ENGAGE as proof by demonstration that battery-free mobile gaming was possible. In this section we demonstrate that the system can play unmodified retro games despite intermittent power failures. We analyze the real-world execution of the platform while playing *Tetris* in different lighting scenarios (i.e. with different energy scarcity) to show the effect of energy availability. We then benchmark the ENGAGE hardware platform for power consumption and, investigate the performance of the MPatch system. We find that in well-lit environments playing games that require at least moderate amounts of clicking, play is only slightly interrupted by power failures (less than one second of failure per every ten seconds of play). Our measurements of MPatch across four different games show that checkpoints are fast (less than 50 ms and restoration time after a power failure is not noticeable (average of 140 ms).

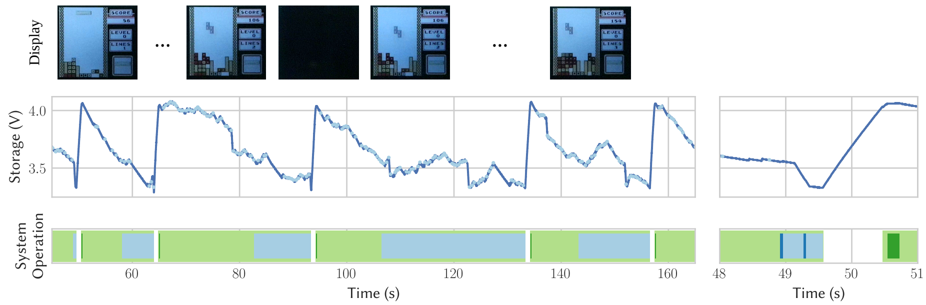


Figure 3.8: End-to-end evaluation of ENGAGE operating in 'daylight' (approximately 40 klx during *Tetris* gameplay using harvested energy only. The storage capacitor voltage is shown, overlaid by unique button presses (marked as light blue dots). Additionally, the following system events are shown at the bottom of the figure: initialization time (marked in dark green), system on time (marked in light green), low energy state (marked in light blue, denoting moments of ENGAGE periodically checkpointing due to critical system voltage) and checkpoint time (shown in dark blue in the separate zoomed-in window on the right). The actual game frames are shown on top, taken from recording the ENGAGE display during the evaluation scenario. The scenario shows that user interaction prolongs the on time of ENGAGE, by pressing buttons during gameplay—achieving ten seconds or more of on time with small off times. We consider this to be a playable *Tetris* scenario.

3.5.1. End-to-End ENGAGE Performance

First, we look at the typical play of ENGAGE executing an example Nintendo Game Boy game *Tetris*, chosen due to its requirement for moderate/high button presses and a small number of cut-scenes. We show how the system operates only on harvested energy. We execute two experiments, each in different lighting conditions: (i) 'daylight' with approximately 40 klx and (ii) 'shade' with approximately 20 klx, where a gamer plays ENGAGE fully untethered, operating on harvested energy only. In the experiment the voltage of the main supply capacitor of ENGAGE is recorded together with various debugging signals indicating different system states. The system state and button presses are recorded using a Saleae logic pro 8 logic analyzer [208]. The ENGAGE platform was placed in a light box with two remotely controllable lights generating the two different light exposure conditions. The luminance of both scenarios was verified using a UNI-T UT383 lux meter [231].

In the first scenario ('daylight', Figure 3.8) we show a period of execution with both little and many button presses. Here clearly the contribution of the energy harvesting by the switches is shown, significantly prolonging the on time of the device (marked in green). The figure shows the complete sequence from startup until the ENGAGE reaches a critically low energy level when it starts checkpointing. Due to the variability in the incoming energy pattern, ENGAGE can spend some time in this state, since it always needs to account for the worst-case scenario of no additional incoming energy. This scenario results in on times of ten seconds or more with small off times of less than a second, making it a very playable experience.

In the second scenario ('shade', Figure 3.9) we halved the amount of light the solar panels are exposed to compared to 'daylight', a more challenging condition for ENGAGE. This reduces on times to around 3.5 s with off times of more than a second. Despite

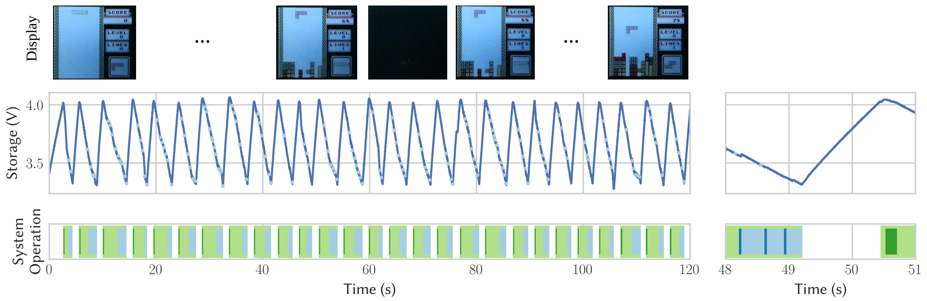


Figure 3.9: End-to-end evaluation of ENGAGE operating in 'shade' (approximately 20 klx. Description of figure elements is the same as in Figure 3.8. With less energy available to ENGAGE as in the scenario in Figure 3.8, on times are reduced to around 3.5 s, with off times of more than a second. This scenario creates a noticeable impact to the user experience.

the system still functioning correctly the lack of incoming energy becomes noticeable and even button mashing cannot compensate for the lack of energy. As with any energy harvesting platform, the limits of operation are defined to a major degree by the available energy in the environment. Full-system emulation is challenging and energy-intensive, but the game is still playable and functional; just with longer intermittent outages. We note that the downward peaks of storage voltage in Figure 3.8 and Figure 3.9 are caused by the energy harvester: during maximum power point tracking no energy is harvested causing the quick drop in the storage capacitor voltage.

Full-System Restoration Time. We have also measured end-to-end time of ENGAGE restoration: from the moment of applying power to the MCU to the moment of executing game code within the Game Boy emulator. In the case of *Tetris* this is 264 ms. The other games we tested resulted in comparable restore times, the main difference resulting from MPatch operations, as is further described in Section 3.5.2.

3.5.2. MPatch Performance

To better understand and quantify the effect of patches on the checkpoint and restore time, we evaluate MPatch against a *naive* approach—comparable in operation to Mementos [203]—where all active memory in the system is copied to non-volatile memory during a checkpoint, even if it was not modified since the last checkpoint. We compare these two strategies, MPatch and *naive*, by running multiple different games on ENGAGE. These games include: (i) *Tetris*, (ii) *Super Mario Land*, (iii) *Space Invaders*, and (iv) *Bomberman*. These games represent a wide variety of play styles, developers, and even release dates.

MPatch Checkpoint Time

To measure only the impact of the MPatch patch checkpoints, we disable the just-in-time checkpoints—used in Section 3.5.1—and run the system on constant power during these measurements. Instead, we perform a checkpoint every *c* execution cycles of the emulator and chose three different values for *c*, which correspond to different *on times*, i.e. 1 s, 5 s, and 10 s. During normal operation checkpoints will only be created when the voltage

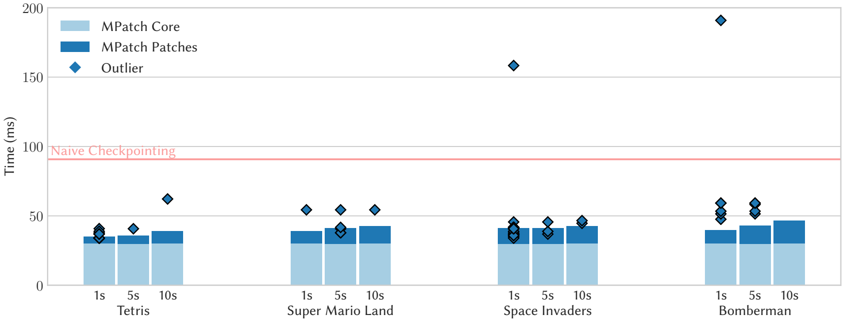


Figure 3.10: MPatch checkpoint time comparison of approximately two minutes of game play per game using three different on times (1 s, 5 s, and 10 s) between successive checkpoints. ENGAGE has noticeably better performance than naive system, across all on times and games.

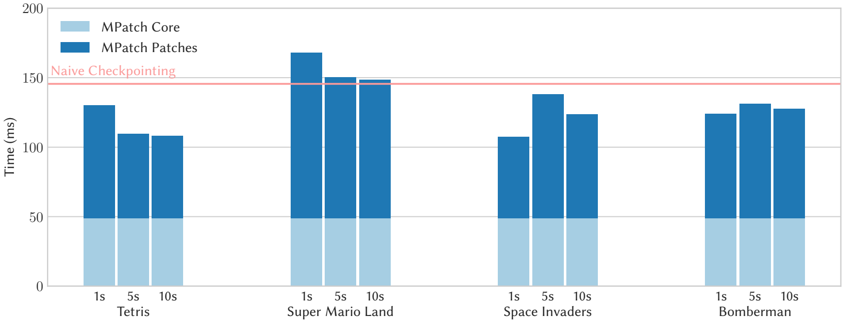


Figure 3.11: Restoration time comparison of after approximately two minutes of game play per game using three different on times (1 s, 5 s, and 10 s) between successive checkpoints. ENGAGE has comparable or better performance than naive system, across all on times and games.

reaches a critical threshold, as seen in Section 3.5.1. These fixed on times represent a simplified scenario where the critical voltage threshold is reached after the specified on time. The *on time* affects the number and size of the checkpoints, as it allows for more memory writes between two consecutive checkpoints. The on time does not affect the *naive* checkpoint, as it always checkpoints all memory, with the only variable size being the system stack of ENGAGE. However, because of the way ENGAGE works—as an emulation loop—the system stack size is virtually constant.

During the emulation of each game, with the three different on times, we measured the cost of each component of the checkpoint using the same logic analyzer as used in experiments in Section 3.5.1. A checkpoint of ENGAGE consists of a core checkpoint of ENGAGE (Section 3.4.3) and additionally patches created by MPatch. The core checkpoint includes the management of both the emulator and the emulated memory, but excludes the emulated memory itself. This emulated memory is the largest memory component of the system, and therefore also the largest component of a naive checkpoint. For this reason we checkpoint this part of the system using MPatch, as the other components of ENGAGE are virtually constant in the amount of memory that is modified and are thus covered by the core checkpoint.

Figure 3.10 illustrates the naive checkpoint time as the horizontal line, the average checkpoint time of a core checkpoint (light blue bar), the differential component of a checkpoint using MPatch (dark blue bar), and the outliers (blue diamonds). As can be seen, the cost of the core checkpoint is around 30 % of the complete *naive* checkpoint, the rest being the emulated memory. However, when using MPatch to checkpoint the emulated memory, the core checkpoint dominates the total checkpoint time. In total MPatch is on average *more than two times faster* than the *naive* approach. This confirms our hypothesis that only a small amount of emulated memory is modified during execution. This reduction in checkpoint time directly leads to a lower energy requirement for each checkpoint and leaves more time for game emulation. Interestingly this assumption seems to hold even when the on time approaches 10 s, which is substantial for intermittent devices. Some outliers take longer than a *naive* checkpoint, and this is due to a periodically performed memory recovery procedure (Section 3.4.3)—which was introduced to keep the creation of patches constant while keeping the system incorruptible.

MPatch Restoration Time

We also evaluate the restoration time of patch checkpointing of MPatch, in a similar manner as in the previous section (i.e. the same set of games, comparison against three other reference mechanisms). The results are presented in Figure 3.11.

Restoring a patch-based checkpoint requires more time than the creation of a patch, as described in Section 3.4.3, due to the need to apply only the parts of the patches that are required, and because all the volatile memory has to be restored. Additionally, the restoration procedure must take into account all the committed patches when trying to restore the volatile memory, as each of these might hold some region that was only checkpointed using that specific patch. Therefore it is not directly influenced by the on-period, but influenced by the time since a *memory recovery*. Nevertheless, as can be seen in the figure, MPatch often *reduces the restoration time* compared to *naive* restoration. We can also conclude from this that tested games often only modify a portion of their memory (in this case the emulated memory), as can also be seen in Figure 3.4.

3.6. Discussion and Future Work

Our evaluation of ENGAGE has shown that retro games are playable without batteries, making the next step in self-sustainable gaming made first decades ago by e.g. Bandai Corporation's LCD Solarpower game series [51]. Although the core gameplay mechanisms of the mobile handheld gaming have been successfully implemented, i.e. interaction with screen-displayed data (for the original Nintendo Game Boy), other forms of interaction that make game experience complete are waiting to be researched and implemented.

3.6.1. Limitations, Alternatives and Future Work

Of course ENGAGE is just a first step in the direction of battery-free gaming and the proposed platform still has many limitations that need to be addressed. First, our battery-free platform *plays no sound*. We agree that no sound play is the main hurdle of complete game immersion. How to make sound enjoyable despite power supply intermittency is the core research question, but at the same time (in our opinion) an exciting research area. Some approaches to the sound problem we anticipate as worth-considering are (i) to include separate storage for sound buffering and play, following the architecture of [49, 83], (ii) introduce superficial pauses in the original game tone—effectively making the game sounds identical to the original battery-based game but punctured by silence at pre-selected moments—to make sound interrupts less irritating during gameplay, or (iii) to create intermittent system-specific game sounds—sounds that inform the user that the system is about to die or has just become operational again—to enrich battery-free gameplay.

Additionally, we cannot claim that all games will have the same playability when ported to the intermittently-powered domain. Only when the off-times are negligible for the player we can safely assume that any existing game could be played intermittently. Negligible off-times will cause no irritation to the person who is accustomed to always-on style of play. This observation would hold for any game system—not only classical (but old) Nintendo GameBoy we used as a basis for ENGAGE, but also recent systems such as PlayStation Portable or Nintendo Switch. An open research question is to find how long this off time is (less than a second or maybe less than a millisecond)? Our intuition says that this time is game-dependent and the longer the off times are present in a battery-free console, the set of games that can be ported to the battery-free platform gets smaller. Games that do not need frequent button pushes intuitively would be less irritating to play intermittently (e.g. *Chess*) or *Solitaire*); refer also to qualitative comparison of 8-bit Nintendo Games portability in Table 3.2. However, this creates an interesting paradox of button-based interaction. More button presses during the game result in more energy being supplied to the game console. and Section 3.3.2 (in extreme case games that are based on button bashing, such as classical *Track & Field* arcade game from Konami Corporation, gamer would be able to continuously generate energy purely from gameplay). At the same time less button presses result in less energy being created, causing a reduction in continuous duration of play. To verify the above claims *detailed user studies considering a large pool of gamers and games* need to be performed, where users play different games with artificially-induced intermittent operation (varying duration of on and off times).

Table 3.2: This table describes the difficulty (or irritability) of playing types of Nintendo GameBoy games on intermittent power, assuming the intermittent effect is noticeable to the player and that enough energy is available for some level of play.

Game name	Type	Button presses	Intermittent play	Comments
Baseball	Sports	Very High	Hard	Reaction time is part of the game
Super Mario Land	Platformer	High	Hard	Button press order is crucial
Tetris	Puzzle	High	Medium	Tile rotation is often infrequent
Solitaire	Cards	Low	Easy	No penalty for missing a press
WordZap	Puzzle	Low	Easy	Easy with “no solving time” penalty
Chess	Strategy	Very Low	Easy	Most time spent on thinking

Behavior Nudges to Generate More Energy. Many types of games have natural gaming mechanics that could be leveraged to increase energy harvesting actions. *Dance Dance Revolution*, *Bop-It*, and others, exploring this gaming induced behavior change for increasing energy is an interesting research direction. For example, a specific rapid button pressing sequence can trigger new game events (new levels, extra game points, etc.). Then, there are great user interfaces for battery-free interaction, for instance a crank[2], that can be researched further.

Native Execution. We chose the hard path: running a game emulator on an intermittent platform. This was to demonstrate the range of capabilities available to intermittent computing, and to leverage the vast amount of pre-built games that can play unchanged on the platform. However, one could imagine that native gameplay would significantly increase the performance of the platform, by orders of magnitude, since a single emulated instruction has significant overhead over native code for the platform. This could be accomplished by compiling game binaries to native ARM code, or by leveraging a bespoke gaming API from bare-metal C code. The latter is intriguing as an exercise to take advantage of the unique aspects of intermittently-powered and battery-free gaming, where the situation and context, as well as the gameplay, will affect how much energy is harvested. Game mechanics leveraging this system attribute might increase engagement.

3.7. RELATED WORK

Battery-free Sensors. Long before our idea of a batter-free gaming console, non-gaming embedded platforms were realized in a battery-free manner—making these sensors more environmentally-friendly. The first such battery-free platforms were wireless sensors [190]. First battery-free sensors were based on the idea of computational RFID tags: programmable RFID tags with on-board sensors (such as accelerometers or temperature sensors) communicating with the outside world by radio frequency backscatter to a RFID reader. WISP [234, 210] and Moo [233] are the first realization of such RFID tags. Since the introduction of WISP and Moo many research groups have focused on making battery-free backscatter communication more efficient [244], for instance, by making it free from dedicated energy sources [189], by enabling communication with non-backscatter networks

[2] Which is already used in the upcoming post-retro *Playdate* console [187], which sadly is not used for internal battery charging.

such as IEEE 802.11 [112] or LoRa [221], or by improving backcatter-based networks—either based on standard RFID protocols [141], or based on dedicated backscatter network stack [81]. A separate line of research focused on introducing camera-based image processing to backscatter-based sensors. First, a backscatter-based battery-less cameras, as an extension to WISP platform, has been demonstrated in [164, 165], later followed by a dedicated (non-WISP) backscatter-based system [163, 207]. Additionally, non-radio frequency backscatter systems based on passive visible light communication backscatter, such as PassiveVLC mote [245], have also been demonstrated. It is important to remark that the biggest drawback of backscatter-based systems is the reliance on external energy sources (itself powered by batteries or power lines) that downscales the benefit of removing battery from a complete system.

Additionally, battery-free sensors that communicate using non-backscatter, i.e. active, communication techniques also become actively researched. These include simple sense and transmit sensor powered by ambient temperature differences [255], UFoP [83] and Capybara [49]—energy-harvesting storage-adaptive sensors, Battery Free Phone [223], SkinnyPower—wearable sensor powered by intra-body power transfer [213], Camaroptera—image-inferring sensor [167], SoZu—battery-free activity detector [254], or Botoks—time-aware wireless sensor [54]. Non-wireless/non-communicating battery-free sensors include CapHarvester—local energy monitor powered by harvesting stray voltage from AC power lines-[74], self-powered step motion counter [107], Saturn—battery-free microphone [28], and active radio battery-less eye tracker [129].

Battery-free Interactive Devices. It is imperative to extend battery-less devices beyond a simple ‘sense-and-transmit’ functionality (as summarized above) demonstrating simple forms of user interaction. The same RFID technology that laid the foundation for battery-free sensing was also used to demonstrate battery-less interaction. Such systems include RFID-based tags displaying external information [176], elderly monitoring based on embedded-in-clothes RFID tags [104], surface shape detection [106], speech recognition [237], augmented reality with (i) unmodified RFID tags [128] and (ii) modified RFID tags (to enable touch sensing) [93], interactive building block system with augmented RFID tags[3] [131, 94] or finger gesture measurement [111]. It needs to be emphasized that any RFID tags-based interaction is very sensitive to interference and signal mis-matches as demonstrated in [238].

Separately from RFID-based battery-free interactive devices, non-RFID counterparts are also actively researched. Most of these devices focus on remote device control through touch. Examples of such devices are capacitance-based touch sensors (although communicating with FM radio receiver through backscatter) [236], Ohmic-Sticker—force-to-capacitance sensors attachable to laptop touchpad [95], aesthetically pleasing self-powered interactive surfaces based on photovoltaic cells [156] and self-powered gesture recognition based on (i) photovoltaic panels [137][4], (ii) photodiodes [130] and (iii) capacitance sensing [229]. E-ink battery-free wearable displays embedded in clothes, energized by NFC-enabled smartphones were demonstrated in [56].

Another approach for battery-free embedded devices is to equip the area where the sensor resides in some form of wireless power transfer system. Many end-to-end

[3] A similar concept for NFC-based tags has been presented in [38].

[4] System claims to be battery-less, while in evaluation a battery-based version was used.

wireless power solutions can be found in the literature, including recent systems built on top of capacitive power transfer [253], magnetic resonant coupling [220], quasistatic cavity resonance [211], lasers [99] or distributed RF beamforming [61]. As in the case of backscatter-based sensors, wirelesly-powered sensors require external (complex, bulky and still having not fully resolved safety issues) infrastructure. This limits applicability of this approach to ubiquitous battery-free gaming.

Battery-free Gaming. An ultimate form of interaction is through a gaming system. The first commercial battery-free/solar-powered gaming platform was Bandai's LCD Solarpower [51], released already in 1982, which enabled the manipulation of hard-coded elements on a liquid crystal display. Unfortunately, Bandai's console and modern existing academic-grade battery-free gaming systems are limited to a simple game forms, such as attachable touch pad extenders for better (but still battery-powered) mobile game experience [43, 251] (similar to an earlier referred design [95]), extra controllers for smartphones based on its front/rear cameras [246], or based on RFID technology that requires heavy-lifting of battery-less features by an expensive RFID reader using either (i) computational RFID tags [233, 234] as for instance in [151], or (ii) using commercial off-the-shelf RFID tags as in [127]. Battery-free non-RFID touch pad extender for the introduction of physical manipulation into touch screen-based games was prototyped in [169]. Battery-free gaming aimed at children includes a system based on rubbing/-touching electrostatic surfaces to power simple electronics [110, 42, 41] and an attachable energy harvester mote for learning and understanding concepts of energy generation and consumption [206].

Sustainable Design of Interactive Devices. Design of any future interactive devices must consider sustainability and reuse, as advocated already a decade ago in [36, 149]. The same plea, but in the context of pervasive devices, was presented in [101]. For almost a decade many studies call for sustainable 'upstream' HCI by making conscious choices in HCI design process in selecting materials that are sustainable, recyclable and reusable [114] or using post-apocalyptic terms— HCI "designed for use after the industrialized context has begun to decay" [228]. We are unaware of any studies on whether the (handheld) gaming community considers sustainable gaming as important, let alone existing, problem. A loosely related study to our posted problem is the study on the motivations behind leading green households [243].

Intermittent Computing Systems. As discussed in Chapter 1, the goal of intermittent computing frameworks is to guarantee the correctness and completion of the computation of battery-less energy harvesting embedded platforms *despite* frequent power interrupts. Such framework is essential for the usability of battery-free gaming platform.

From the publication of the first framework supporting intermittently-powered devices, Mementos [203]—voltage threshold-triggered checkpointing system, more efficient checkpoint systems are being published. These include Hibernus++ [30] and QuickRecall [103] (just like Mementos, both hardware-activated checkpoints), DICE (differential checkpoints) [11, 12] and WhatsNext (checkpointing augmented with approximate computing) [66].

A separate stream of work targets peripheral support for intermittently-powered devices, such as Restop (through dedicated middleware) [29], Samoyed (through just-in-time checkpoints) [145] and Karma (supporting parallel or asynchronous peripheral operations) [37], or targeting handling of dedicated peripherals such as e-displays (to improve their update rate) [157].

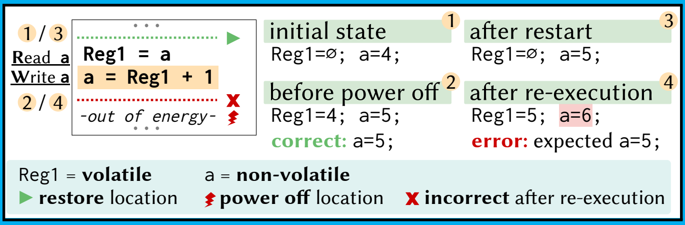
3.8. CONCLUSIONS

This chapter presented a first working example of a battery-free gaming console and the first full system emulation on intermittent power: ENGAGE. We demonstrate we can port existing battery-based gaming platforms—such as in our case 8 bit Nintendo Game Boy—to the battery-free domain. To achieve this, we developed MPatch, a new system for persistent computation across power failures based on a novel concept of patch checkpointing of the volatile memory state into non-volatile memory regions.

Part Two: Non-Volatile Main Memory

In the previous part of the the thesis, we focused on enabling intermittent computing on embedded systems with volatile main memory (SRAM). However, despite our best efforts, a significant amount of data is still copied to and from non-volatile memory to keep a consistent copy of our volatile memory, therefore still introducing considerable overhead to support intermittent computing. We can mask this overhead using larger buffer capacitors, requiring checkpoints less frequently. But increasing the capacitor size also increases the charge time and the system's total size as the capacitor's physical size increases.

In this second part of the thesis, we target systems with non-volatile main memory, e.g., MRAM, FRAM, or ReRAM, which are specialized non-volatile alternatives to SRAM. Using non-volatile main memory eliminates the need to store the main memory during a checkpoint, significantly reducing the cost of checkpoints. However, these non-volatile memories consume more energy and are slower than their volatile counterpart. Additionally, using non-volatile memory requires more complicated techniques to keep the memory synchronized with the checkpoint of the registers, introducing additional overhead.



An example of a memory sequence in which a ***read*** *operation (**R**) is followed by a* ***write*** *(**W**) can result in corruption after a power failure.*

Corruption in intermittent systems with non-volatile main memory is caused by re-executing Write-After-Read (WAR) dependencies, shown in the figure above. Whenever data in the non-volatile memory is read and later written to, re-executing this section of code will re-execute the original read from memory. However, this time the read loads a different value from memory, namely the value written during the write operation just before the power failure, which will cause the re-execution to be incorrect. This WAR phenomenon is actually the same underlying reason for corruption in Chapter 1 (Section 1.2.1). However, it can be handled more easily in systems with volatile main memory, as the memory content must be copied regardless. When using non-volatile memory, we do not want to copy the entire memory state, as this would undo all the benefits of using non-volatile main memory.

4

Avoiding Checkpoints Using Stack Segmentation

In this chapter, we consider an embedded device with non-volatile main memory in the form of FRAM. Having non-volatile main memory eliminates the need to store all the memory during a checkpoint. However, to maintain a consistent memory state, special care must be taken to restore the non-volatile memory to the condition it was in when the checkpoint was created before computation can continue. To address this synchronization problem, this chapter introduces a segmented stack approach where one active stack is included in the checkpoint and can be modified freely. Access to memory outside this active stack must be logged and restored to maintain memory consistency in case of a power failure.

This chapter is based on the following publication:
Vito Kortbeek, Kasım Sinan Yıldırım, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. *Time-sensitive Intermittent Computing Meets Legacy Software.* In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 85–99, March 2020. https://doi.org/10.1145/3373376.3378476.
The accompanying archive containing all the software and data can be found at https://doi.org/10.5281/zenodo.3563081.

4.1. INTRODUCTION

As referred to in Chapter 1, using existing approaches that support intermittent computing on systems with non-volatile main memory introduces numerous challenges. Task-based programming requires significant developer effort to transform a program to fit the programming model [48]. On the other hand, checkpointing systems remove the cognitive burden of porting, but have high memory overhead and performance penalties due to frequent checkpoints [235]. Moreover, some of these systems cannot execute all C-programs: in particular, pointers [250] and recursion [143] might lead to incorrect checkpoints. Additionally, not all existing techniques (e.g., [235]) are effective for microcontroller architectures with instructions that can directly read and modify memory in the same instruction, such as the MSP430FR [224], an often used microcontroller series within the intermittent computing community due to its onboard FRAM. These instructions would need to be separated into multiple instructions to create idempotent regions, which introduces additional overhead and requires significant changes to the compiler. Furthermore, existing checkpointing systems do not allow semantics to handle *elapsed time* and in turn they cannot handle time-sensitive data that might be expired after a long power failure. Developers have no way to easily inject decision points into legacy software based on the time elapsed since failure can occur in-between any lines of the code.

These issues beg the question: is there a way to bridge the gap between time-sensitive intermittent computing and legacy software designed for continuously-powered systems? As of now we are still far from an *ideal* intermittent computing system that (i) removes the cognitive burden of porting legacy software and enables unaltered C programs (with standard programming constructs and any typical compiler optimizations enabled) to be executed on intermittent power; (ii) provides semantic and syntactic mechanisms to handle data freshness (and passing of time in general) for *timely execution* of the application; and (iii) introduces low memory impact and little performance penalty. These requirements are necessary to enable the widespread adoption of intermittent computing.

In this chapter, we propose **TICS** (Time-sensitive Intermittent Computing System), an intermittent computing system designed with the goal of running time-sensitive code on intermittent platforms via *automatic* checkpoints. TICS enables programmers to (i) execute any kind of *unaltered C program* (including pointers and recursion) by greatly reducing, as well as bounding, the overhead of checkpoint/restore times—eliminating system starvation, and (ii) optionally annotate the program with structures to specify custom *timing requirements*—protecting against timing errors that are never seen in continuously-powered programs. The core scientific contributions of this work are:

- *Time sensitivity semantics* for checkpoint-based intermittent systems—enabling, for the first time, declarative annotations for intermittent applications to handle the passing of time in-between power failures and to eliminate *time consistency violations* particular to intermittent systems;
- *Memory consistency management* for checkpoint-based intermittent systems by combining *data versioning* and *stack segmentation* to bound checkpoint/restore times—enabling the execution of unaltered C-programs–including pointers and recursion–*without system starvation* and endangering memory consistency, and providing a foundation for memory isolation and interrupt handling;

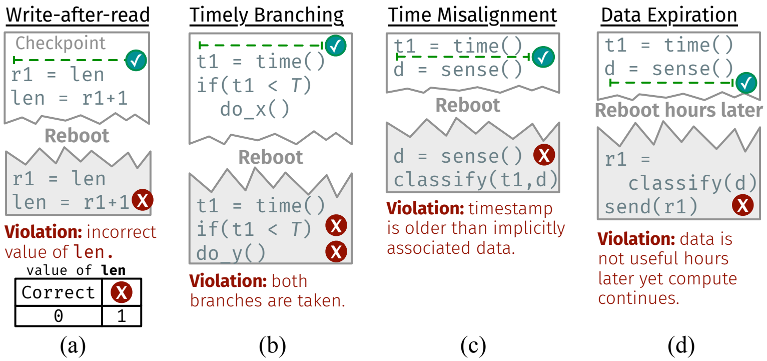


Figure 4.1: **Four types of consistency violations encountered with automatic checkpointing.** These violations occur because of incorrect execution caused by bad checkpoint placement, leading to an execution that is not possible on a continuously powered device. With this work we introduce a new class of violations, i.e. *time-based violations*, that have not been previously explored in checkpointing systems—refer to figures (b)–(d).

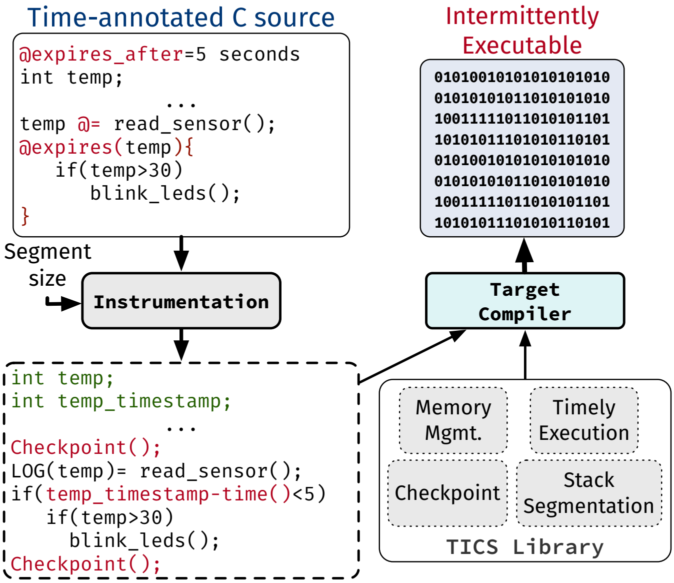
TIME CONSISTENCY.

Consistency violations identified in previous work [136] include only memory consistency violation; see Figure 4.1(a): after a checkpoint, non-volatile global variable `len` is changed, but these actions are not included in the checkpoint. When the checkpoint is restored, `len` is again updated, leading to an incorrect value of `len` due to the Write-After-Read (WAR) dependency. We identify *three other types of consistency violations*, all having to do with time. The errors stem from the fact that clocks internal to the MCU are reset after each power failure, meaning that devices have difficulty tracking how long they have been off [87, 199]; even when using external timekeepers, time-sensitive portions of a program must be handled differently in checkpointing systems by careful checkpoint placement or time management.

1. **Timely Branching.** If a checkpoint is placed in a line of code before a timestamp is gathered, and that timestamp is used in a predicate statement, execution can execute both branches if the timestamp elapses; see Figure 4.1(b);
2. **Time and Data Misalignment.** Often in embedded programs, a timestamp is gathered every time sensor data is obtained. If a checkpoint is placed between the timestamp and the data gathering, the timestamp will be inaccurate. After a power failure recovery at that checkpoint, *new* data will be gathered associated with an *old* timestamp—causing incorrect execution of the program; see Figure 4.1(c);
3. **Data Expiration.** Data gathered in one power cycle may not be fresh enough for the next power cycle. This phenomenon [86] has not been handled by any automatic checkpointing systems to date; see Figure 4.1(d).

4.2. TICS: SYSTEM DESIGN

TICS consists of a runtime combined with code instrumentation for the C language—Figure 4.2 presents the logical flow and the main components of the TICS system. The main motivation behind TICS is to provide the view of a continuously-powered system



```
@expires_after=5 seconds
int temp;
                  ...
temp @= read_sensor();
@expires(temp){
   if(temp>30)
      blink_leds();
}
```

```
0101001010101010101O
```

```
int temp;
int temp_timestamp;
                  ...
Checkpoint();
LOG(temp)= read_sensor();
if(temp_timestamp-time()<5)
   if(temp>30)
      blink_leds();
Checkpoint();
```

Figure 4.2: **TICS overview:** A runtime combined with code instrumentation ensures memory consistency via *data versioning* and *stack segmentation*; progress of computation via *checkpointing*; and timely execution via *time annotations*.

to the programmer—so that legacy C code can be run without any modification to the program source. TICS allows the programmer (i) to focus on the correct and timely execution of the application—eliminating the explicit need for intermittency handling, and (ii) to perform a few modifications to the original program, specifically, to annotate their code *only* to define timing constraints.

Task-based versus Checkpointing. Conversion of a C program into a task-based program requires significant manual labor; automatic transformation of a pointer-based C program is incredibly difficult due to memory burden created by a multitude of versions of memory locations/variables. Therefore, instead of task-based transformation, TICS uses checkpointing in order to get rid of manual code transformation and its limitations.

Building an Efficient Stack. As the amount of state that is checkpointed grows, the checkpointing overhead increases, potentially leading to overheads that may exceed the device's capabilities and energy budget. Since functions often manipulate local variables in their stack frame, there is no need to checkpoint the whole stack. TICS employs a novel strategy by segmenting the stack into fixed and predetermined size blocks. The stack segment that is directly manipulated at a time instant by the program is called the *working stack* and it will be the only one among others that needs to be logged into a *segment checkpoint*—since other segments are not modified. By segmenting the stack TICS can provide a fixed worst-case checkpoint time, as the variable stack size is fixed to the size of a stack segment. It is worth mentioning that the programmer is completely unaware of the underlying stack segmentation but the desired size of stack segments can be chosen at compile time for the sake of performance—see Section 4.4.

Pointer Handling. As pointer access cannot be determined at compile time, existing systems need to checkpoint the whole main memory in order to keep memory consistent—leading to huge checkpoints and in turn *system starvation* due to limited energy reservoir. TICS implements a data versioning scheme to handle pointers and ensure memory consistency: it keeps track of only manipulated memory locations by keeping the original values in a non-volatile *undo log*. The undo log is cleared upon a successful checkpoint, otherwise TICS restores the original contents of the memory using the undo log—ensuring memory consistency despite power failures.

Memory Impact. Checkpointing the device's volatile state requires an atomic *two-phase commit* operation to ensure its consistency [142]: in the first phase the checkpointed data is copied to a temporary buffer in non-volatile memory; then in the second phase the buffered data is committed to the original location. Existing checkpointing systems double buffer the stack, `.bss` and `.data` sections—their memory requirements increase with the volatile state. On the other hand, TICS only requires the segment checkpoint and the modified memory locations in `.bss` and `.data` sections to be double buffered—significantly reducing the memory impact.

Timely Execution. C does not provide any keyword/statement to express time constraints of the data and handle it—programmers must explicitly timestamp data and handle data expiration. This complicates application development as well as might lead to bugs due to manual timing and expiration checks, control flow delivery and recovery due to data expiration—as given in Section 4.1. TICS provides annotations to relate data and time as well as special statements to change control flow and perform recovery upon data expiration—all underlying time management is performed at run-time without programmer intervention.

4.2.1. Efficient Automatic Checkpoints

Existing works, e.g. [235, 88, 35] exploit architectural support and ensure constant and scalable checkpointing overhead. For example, Ratchet [235] uses non-volatile memory as the main memory so that stack and global variables are already persistent—leading to constant checkpoint time since only the volatile registers of the processor are checkpointed. This requires decomposing programs into idempotent code sections via the compiler using *static analysis* at the instruction level and gluing them together with checkpoints. However, dynamic memory manipulations that cannot be determined at compile time, e.g. write operations via pointers, require a checkpoint after each instruction, leading to a considerable checkpointing frequency and, in turn, overhead. TICS targets devices with non-volatile main memory—a checkpoint operation logs *only* the registers and the stack in a dedicated double-buffered area in non-volatile memory via a two-phase commit. Since the stack grows/shrinks dynamically, checkpointing overhead grows with the size of the stack. Moreover, recovery time, i.e. restoring the state after a power failure, is not fixed and might exceed the device's energy budget—leading to *system starvation*. TICS remedies this with *stack segmentation* and *data versioning*.

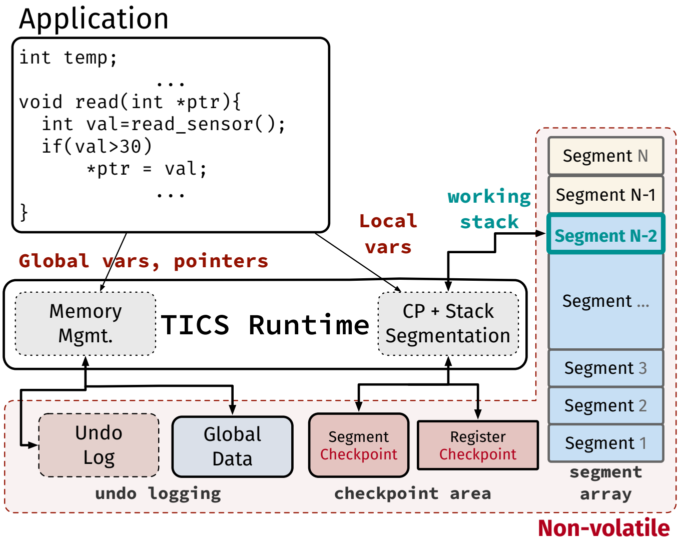


Figure 4.3: **TICS architecture.** With TICS only the working stack and the registers during a checkpoint are logged—ensuring deterministic worst-case overhead. Previously checkpointed segments belonging to the lower parts of the stack are maintained in a segment array. The global variable and pointer access are handled by the memory manager which implements undo logging to keep memory consistent.

STACK SEGMENTATION.

The stack allocation within the execution of the applications might vary significantly, in particular when a lot of memory space is allocated/deallocated at function entries/returns. The stack size requirement depends on dynamic program flow (that might be unknown at compile time) and in turn, it is not possible to guarantee a worst-case checkpoint size. TICS segments the stack into blocks of fixed size selected at compile time—the maximum stack frame in a program (determined during compilation) dictates the minimum block size. TICS maintains the segmented stack of a program as a *segment array* in non-volatile memory—see Figure 4.3. The size of the stack array is fixed at compile time by considering the stack requirements and exceeding the size at runtime leads to a stack overflow. The program interfaces with the top segment of the segmented stack, the so-called the *working stack*: the program modifies only the working stack, and upon a checkpoint, only the working stack is two-phase committed into the double-buffered *segment checkpoint*—this enables a fixed checkpoint time. Moreover, recovering from a power failure only requires the working stack to be restored from the segment checkpoint, instead of restoring the whole stack.[1]

During program execution, the stack grows/shrinks making the working stack point to different segments in the segment array. When a function is entered, the stack pointer is adjusted: TICS inserts a check before the modification of the stack pointer to determine whether there is enough space in the working stack to execute the function. When enough

[1] Differential checkpoints [11] log only modified part of the stack—but they can still be large for nested function calls each using a lot of stack.

space is in the working stack, the execution resumes and the function interfaces with the working stack. Contrary, if there is not enough space left on the working stack, a *stack grow procedure* is initiated so that the working stack points to the next segment in the segment array. It is worth mentioning that a checkpoint after this point requires only the new working stack to be saved into the segment checkpoint since the previous segments remain unmodified. When a function that triggered a stack grow returns, a *stack shrink* is initiated so that the working stack points to the previous segment in the segment array. TICS can enforce an implicit checkpoint if the current working stack was not saved into the segment checkpoint yet. This is because if the currently checkpointed segment is out of the program stack, the working stack should be checkpointed first so that the modifications can be rolled back upon power failures and the stack consistency is ensured.

It is worth mentioning that no special attention is needed when TICS executes *recursive functions*. However, as in general embedded systems, the depth of the recursive calls is limited by the size of the stack memory, which is represented by the fixed size of the segment array in TICS architecture.

Memory Management and Pointers.

TICS maintains global variables; i.e. `.data` and `.bss` sections in non-volatile memory. Intermittent execution might create inconsistencies if the application modifies non-volatile memory directly and the modified locations are not versioned, i.e. double buffered [202, 136]. TICS instruments non-volatile memory write operations and enables on-demand versioning: *undo logging* is employed so that if any memory location outside of the working stack has been modified, the original version is saved in an undo log. After a successful checkpoint, the undo log is cleared. Upon power failure, the contents of the undo log are written back to the original locations. Since the undo log is also fixed in size, TICS forces a checkpoint when the undo log is full to eliminate the overflow and ensure forward progress.

In TICS, pointer writes to global variables within the `.data` and `.bss` sections in non-volatile memory are managed at runtime. Additionally, *pointers* to the stack can manipulate memory locations, in particular, stack segments other than the working stack. A pointer to the working stack can directly modify its contents since the working stack is checkpointed separately. Conversely, if it points to other segments in the segment array or global variables in `.data` and `.bss`, the memory manager employs undo logging.

4.2.2. Semantics for Timely Execution

TICS provides annotations; i.e. `@expires_after`, to denote the expiration constraints of the data and necessary keywords for checking if time constraints are met—see Section 4.4. A timestamp value is associated with each programmer annotated variable and the write operations on these variables are instrumented by the compiler. TICS can update the value of the timestamp automatically upon writes using a persistent timekeeper which keeps track of time across power failures [86]—see Section 4.3 for details. Programmers can check the expiration of the data using `@expires` block—TICS compares the current time with the timestamp to identify if programmer-defined timing constraints are met. Programmers can also use `@expires_after=0s` statement for any variable that

```
@expires_after=1s /* data expires in 1 second */
int temperature[WINDOW_SIZE];
...
/* data & timestamp alignment (assign timestamp) */
temperature[i] @= read_sensor();
...
/* catch data expiration */
@expires(temperature[i]){
  if(temperature[i] > max) {max = temperature[i]};
}
...
/* branch in time (before the send deadline) */
@timely(SEND_DEADLINE){ send(max); } else {...}
...
```

Figure 4.4: **An overview of TICS annotations for timely execution of intermittent applications.** TICS supports timely branches, ensures data and time alignment and catches data expiration.

requires a timestamp associated with it but does not have any expiration constraint. It is the responsibility of the programmer to provide necessary logic within these syntactic structures.

SUPPORTING TIMELY BRANCHES.

In order to prevent timely branch violations as depicted in Figure 4.1(b), TICS introduces `@timely/else` block that takes a time value as an input. This block disables automatic checkpoints, reads the *current time* using the (persistent) timekeeper and checks if the given time value is greater than the current time. If this is the case, the branch is taken, a checkpoint is placed at the end of the branch and automatic checkpoints are enabled. Otherwise, the branch is not taken and automatic checkpoints are enabled.

ENSURING DATA AND TIME ALIGNMENT.

As depicted in Figure 4.1(c), if a checkpoint is placed between the timestamp assignment and the data gathering (or vice versa), the timestamp can be inaccurate after a power failure. In particular, this issue is problematic if checkpoints are done automatically, e.g. with a periodic timer. To remedy this, timestamp assignment and data gathering operations should form an atomic block. TICS ensures the atomicity by (i) disabling automatic checkpoints so that timestamp assignment and data gathering cannot be split; and (ii) placing a checkpoint right after these operations (and enabling automatic checkpoints thereafter, if needed) so that the consistency of timestamp and data is guaranteed despite a power failure.

TICS introduces operator `@=` for the atomic assignment of the data and timestamp—see Figure 4.2. TICS makes this assignment explicit via `@=` since there is no need to update the timestamp of the associated data per each write, e.g. the sensed temperature value can be converted from the raw ADC value to the degree in Celsius and this conversion should not lead to the update of the associated timestamp.

CATCHING DATA EXPIRATION.

In TICS, `@expires` and `@expires/catch` blocks are used to work with the data within a certain time frame and to catch data expiration— Figure 4.1(d). For the sake of implementation simplicity, we remark that these blocks consider only one variable.

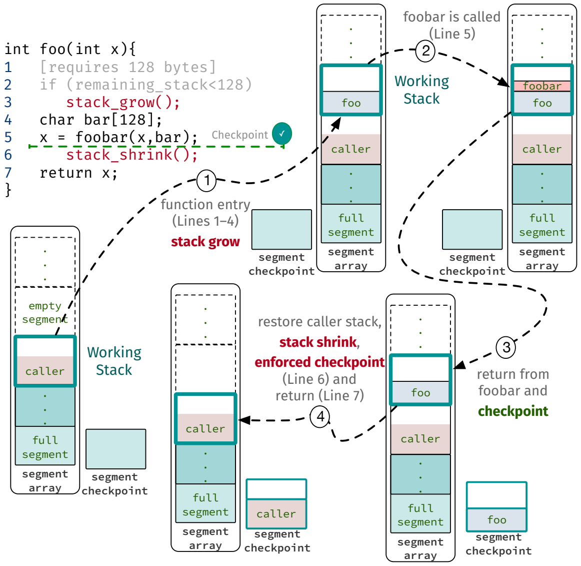
Conditional-based @expires. TICS implements `@expires` block by using an `if` statement at the beginning that checks if the data is still valid; see Figure 4.2. If the condition is met, the rest of the operations will be executed within this block. Due to automatically-inserted checkpoints and arbitrary power failures, `@expires` block might not be atomic. If a checkpoint is placed inside an `@expires` block, a power failure might lead to data expiration—TICS disables automatic checkpoints at the beginning of the `@expires` block so that computation starts from the `if` statement after each power failure. TICS places a checkpoint at the end of `@expires` block and enables automatic checkpoints thereafter. It is worth mentioning that these operations ensure atomicity, but data can still expire since the instructions within the `@expires` block can be long enough to violate data freshness constraints.

Exception-based @expires/catch. In order to catch data expiration while executing an `@expires` block, TICS sets a timer at the beginning that fires when the data expires. Upon timer fire and in turn data expiration, TICS restores the original contents of the modified variables inside the `@expires` block by using the original values in undo log. TICS delivers the control flow to the `catch` block that handles specific logic to handle data expiration. Since undo logging is required for exception-based implementation, its implementation is parallel to the rest of TICS for the sake of memory consistency.

4.3. TICS: IMPLEMENTATION

TICS is built around the MSP430FR5969 [224] MCU with 64 KB non-volatile (FRAM) and 2 KB volatile (SRAM) memory. The compiler back-end *instruments* the assembly to support stack segmentation. The code instrumentation is done via the LLVM utility library LibTooling [177], which is intended for both static analysis and code transformations. We employed *code transformation* rather than compiler support, to allow for portability, enabling the use of multiple compilers, and in turn eliminating the need for re-implementing the instrumentation. In order to produce the target binary, we used MSP430-GCC version 7.

Stack Segmentation. In TICS, stack segmentation is employed at function entries and exits. Before the stack grows or shrinks, TICS checks the stack frame size of the corresponding function (known at compile time) to determine if the function can be executed by using the current working stack. If there is not enough space in the working stack, a stack grow procedure is initiated so that the working stack points the next segment in the segment array. Since the arguments of the function remain in the previous segment, these arguments are copied from this segment to the empty working stack. If a stack shrink is needed, the caller stack is restored, the working stack is changed and a segment checkpoint is performed if the previously checkpointed data belongs to a segment lower than the current working stack. All these operations are depicted in steps 1–3 in Figure 4.5. To enable these operations, we modified the compiler back-end to insert the required stack availability check and argument copying operations—the size of a stack segment is determined at compile time and its minimum size depends on the minimal stack requirements of the functions in the source.



```
int foo(int x){
1    [requires 128 bytes]
2    if (remaining_stack<128)
3       stack_grow();
4    char bar[128];
5    x = foobar(x,bar);
6       stack_shrink();
7    return x;
}
```

Checkpoint ✓
① function entry (Lines 1–4) stack grow
② foobar is called (Line 5)
③ return from foobar and checkpoint
④ restore caller stack, stack shrink, enforced checkpoint (Line 6) and return (Line 7)
Working Stack
empty segment
caller
foo
foobar
full segment
segment array
segment checkpoint

Figure 4.5: **TICS stack segmentation and checkpointing.** In pseudocode: lines in light gray and red colors represent the code inserted during the compiler pass; the red lines only execute when the working stack needs to grow or shrink.

Memory Consistency Management. To implement undo logging so that the changes in non-volatile memory locations (other than the working stack) can be undone, global variable and pointer manipulations are instrumented. Since pointers can point not only to global data but also to the working stack or segment checkpoint in non-volatile memory, the instrumentation is done by checking if the physical address is in the working stack or not. If so, the memory manager logs the contents of the memory cell in the undo log[2].

Automatic Checkpoints. To keep the consistency of checkpointed data—as the system can die while performing a checkpoint—the checkpoint data is double buffered in non-volatile memory. A flag is used to provide an exact barrier after which the checkpoint is ready to be used as a restore point. These enable checkpoint operations to be atomic. Checkpoint restoration happens when the system reboots due to a power failure. The current implementation supports: (i) *timer-driven* checkpointing; where the runtime interrupts program execution and checkpoints the system state periodically at a given frequency; (ii) *hardware-assisted* checkpointing, e.g. [30] where a voltage level based interrupt triggered upon a low-energy state to perform a checkpoint; and (iii) manual checkpoints. It is worth mentioning that TICS disables (automatic) checkpoints before *interrupt service routines* and places an implicit checkpoint right after *return-from-interrupt* (ISRs) instruction. This is sufficient to prevent memory inconsistency while servicing interrupts—if a power failure prevents the completion of an ISR, the system will continue as if the interrupt did not occur right after the recovery from the power failure.

[2] Memory management is implemented fully in software as microcontrollers, e.g. MSP430FR59* [224], do not have a memory management unit.

		'Greenhouse monitoring' routines				
Intermitt.		**Sense Moisture**	**Sense Temp.**	**Compute**	**Send**	**Consist.**
4%	**plain C**	9	9	9	0	✗
	plain C + TICS	0	0	0	0	✓
	TinyOS	0	0	0	0	✓
	TinyOS + TICS	0	0	0	0	✓
48%	**plain C**	29	29	29	20	✗
	plain C + TICS	20	20	20	20	✓
	TinyOS	29	29	29	20	✗
	TinyOS + TICS	20	20	20	20	✓
100%	**plain C**	47	47	47	47	✓
	plain C + TICS	45	45	45	45	✓
	TinyOS	47	47	47	47	✓
	TinyOS + TICS	44	44	44	44	✓

Table 4.1: **Real-world program with TICS on intermittent power (4%, 48% and 100% intermittency rate).** We ran four applications implementing greenhouse monitoring (GHM): in C and in TinyOS (with and without TICS instrumentation). We measured how many times each GHM routine executed. Only these programs that consistently executed the same number of routines were considered correct.

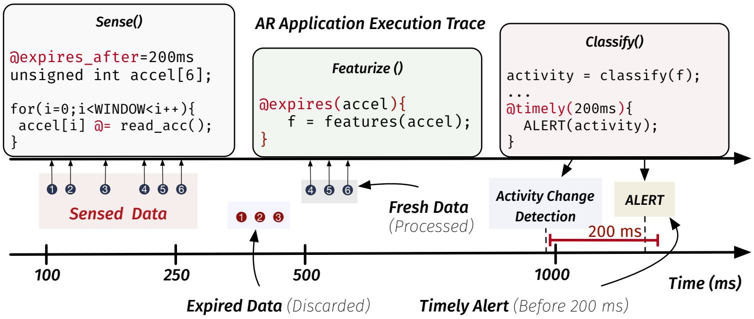
Time Annotations. Each write to a time-annotated variable is instrumented so that the timestamp value associated with the variable is updated. To implement exception-based time annotations, we instrumented `@expires/catch` block so that a timer is set considering the data expiration constraints. Moreover, we instrumented the necessary instructions for undo-logging the memory modifications and changing the control flow upon data expiration. TICS with time-sensitive programs requires the ability to measure time across power outages using a remanence-based timer [87, 199] or a Real-Time Clock with a small capacitor [84]—persistent timekeeping is mandatory to update timestamps and to handle time annotated source files.

4.4. EVALUATION

We investigate the execution overhead of TICS for various applications, comparing to the state-of-the-art intermittent runtimes. We demonstrate how TICS enables porting of arbitrary C programs as well as TinyOS code—for the first time we demonstrate the *successful execution of legacy code for sensor networks into the intermittently-powered domain.* We also show results from a user study we conducted comparing TICS to task-based programming. We found that TICS has comparable overhead to state-of-the-art runtimes while providing **a complete set of features available to the regular C programmer**.

4.4.1. PORTING LEGACY CODE: TINYOS TO INTERMITTENT WORLD

To prove the claim that TICS enables automatic porting of existing/legacy C code for non-intermittently powered systems, we instrument an unmodified TinyOS program for Greenhouse Monitoring (GHM). GHM executes in an infinite loop to *sense the moisture* of soil, *sense the temperature* of ambient, *Compute* measurement averages and *Send* over a wireless interface. We compare Plain C and TinyOS [126] versions of GHM with and without TICS instrumented checkpoints. Both apps were executed on the same microcontroller as before (MSP430FR5969 [224] evaluation board) with artificially generated power *intermittency traces*, i.e. the microcontroller was brought to a hardware reset following a



```
@expires_after=200ms
unsigned int accel[6];

for(i=0;i<WINDOW<i++){
 accel[i] @= read_acc();
}
```

```
@expires(accel){
   f = features(accel);
}
```

```
activity = classify(f);
...
@timely(200ms){
  ALERT(activity);
}
```

Figure 4.6: **Timely execution of the sample AR application:** TICS catches data expiration, discards stale data and ensures timely branches by following the programmer annotations.

pre-programmed pattern. We compare the results of executing the Plain C and TinyOS versions of GHM in Table 4.1 for varying levels of intermittency. We measured how many times each GHM routine was executed successfully. Only these programs that consistently executed the same number of all routines were considered correct.

Results. We observe that TICS allows to work at *any* intermittency conditions and it executes legacy code correctly. This shows that TICS can run semi-sophisticated legacy TinyOS programs without any manual program porting needed. It is worth mentioning that TinyOS is an event-based operating system and porting event-based legacy code might require some manual modifications for the sake of the semantically correct execution of the application—in particular, timely-sensitive handling of the events should be implemented by the time annotations provided by the TICS in order to guarantee semantically correct results. However, if the programmer omits such manual modifications, TICS still guarantees the forward progress of the computation as well as the memory consistency of the event-based applications. In Section 4.4.3, we also demonstrate the porting of existing computation-based benchmarking applications. Apart from injecting time annotations (if required), all porting is handled by TICS automatically without any manual intervention. Therefore, the evaluation results later on support and complement this result.

4.4.2. TIME-SENSITIVE INTERMITTENT COMPUTATION

For the evaluation of time-sensitive execution of intermittent programs, we considered an existing activity recognition (AR) application used in prior work [47, 136, 142] (this application is also used for benchmarking in Section 4.4.3). The AR application obtains a window of three-axis accelerometer sensor readings and determines whether the device is moving or stationary. In the training phase, the mean and standard deviation features of a window of samples are extracted. Then, in the recognition phase, the activity is determined by performing a nearest neighbor classification. In order to observe the time consistency violations described in Section 4.1, we provided two versions of the AR application: (i) manual management of time (and using MementOS-like checkpoints); and (ii) TICS annotated application. We run these applications by powering our MSP430FR5969 [224] wirelessly with 915 MHz Powercast TX91501-3W transmitter [194]. The microcontroller

Time Consistency Violation	Potential Count (during experiment)	Observed Violations w/o TICS	Observed Violations w/ TICS
Timely Branch	256	32 ✗	0 ✓
Time Misalignment	870	78 ✗	0 ✓
Data Expiration	870	173 ✗	0 ✓

Table 4.2: **Time consistency violation statistics for the AR application running intermittently.** Our results indicate that TICS eliminates these violations by demanding little modifications on the legacy software.

was connected to a Powercast P2110-EVB receiver (with on-bard 10 μF storage capacitor). We tested the execution of these applications at the same distances resulting in almost the same (i) power failure rates, (ii) charging and (iii) off-time. We observed the number of time consistency violations.

The lines of code where the accelerometer is sampled and the corresponding timestamp is assigned are the potential points for time misalignment violation. Specifically, a timestamp can be assigned to the sensed data a relatively long time after the sensor sampling, due to a power failure and long charging time—in both applications there were 870 accelerometer sampling where time misalignment violations could potentially occur. The obtained samples are also subject to data expiration violation while they are consumed for training and classification. In these applications, we considered data to be fresh and useful if it is consumed within 200 ms time window—it is considered to be stale otherwise (see Fig. 4.6). In order to keep track of the duration of the recognized activities, both applications maintain timestamp. A timely branch that uses this timestamp is required to alert about activity changes; e.g. if the duration of the activity is less than 200 ms this indicates an activity switch. There were 256 points in the execution where a potential timely branch violation could occur.

Results. Table 4.2 summarizes our results. We observed that TICS prevents all time consistency violations, thanks to easily injected time annotations, whereas the other application led to 32 timely branch violations, 78 time misalignment violations and 173 data expiration violations. Our results indicate that TICS ensures timely intermittent execution by providing little modifications on the legacy software via its time annotations.

4.4.3. TICS System Efficiency

TICS supports all C language features—including pointers and recursion— thanks to its memory consistency manager. This implementation eliminates system starvation by allowing porting any kind of legacy software to the intermittent computing world—breaking the limitations of the prior work. Here we provide a performance comparison of TICS with the prior work to explore its execution overhead.

We have compared TICS against three state-of-the-art task-based systems: InK [250], MayFly [86] and Alpaca [142]. In addition, we compared TICS against naïve checkpoint-based system that logs the complete stack and all global variables (which closely resembles what MementOS [203] does) and Chinchilla [143]—state-of-the-art checkpoint-based system that promotes all variables to global data and statically logs these. Chinchilla was re-compiled from its GitHub source [143] with LLVM version 3.8 (the strict requirement for Chinchilla). InK, MayFly, and Alpaca were compiled with the standard GCC compiler (msp430-gcc version 6.2.1.16). Finally, for completeness, we compare all systems to plain C.

Application Benchmarks. We chose three representative applications, used earlier by most studies on systems for intermittently-powered devices: (i) *bitcount* (BC), (ii) *Cuckoo filter* (Cuckoo) and (iii) *Activity Recognition* (AR) (as indicated in Section 4.4.2) [76]. BC implements bit counting in a random string with seven different methods (including recursion), later cross-verifying for correctness; Cuckoo implements cuckoo filter over a set of pseudo-random numbers, then performs sequence recovery using the same filter; AR implements physical activity recognition based on machine learning with locally stored accelerometer data. For a fair comparison, the experiments were conducted using a continuously-powered TI MSP-EXPFR5969 evaluation board [224]. Each application was verified for correctness at the end of each execution. Cuckoo cannot be implemented in MayFly since loops are not allowed in a MayFly task graph. Also, BC used for the evaluation of Chinchilla, see e.g. [143, Fig. 8–10], was not the original one, as the authors have *manually removed the recursion* to make it work with their system.

TICS against Chinchilla.

Chinchilla converts each local variable of a function to a corresponding global variable in non-volatile memory at compile time. This conversion prevents stack manipulation via pointers and in turn checkpointing the whole stack due to pointer manipulations. Chinchilla must know in advance the local variables in order to allocate corresponding global variables in non-volatile memory—recursive function calls and in turn, existing applications that exploit recursive implementations cannot be supported. Moreover, due to the local-to-global conversion via bypassing stack allocation of local variables, there is an explosion in the number of global variables—decreasing the scalability of memory requirements. Inline functions further complicate this issue: the corresponding global variables are needed to be allocated per every line where the function is inlined. As an example, if an inline function of one local variable is called 100 times, then 100 different global variables need to be created. These issues are the major limitations of Chinchilla, making it an incomplete system. Inspecting our results presented in Figure 4.7, TICS is able to execute *all* benchmarks, while Chinchilla cannot run recursion-based code, i.e. BC. Due to the dynamic memory logging employed by TICS, the execution time overhead will vary per benchmark. Additionally, the compiler optimization level has a significant effect because the runtime code is also affected by the lack of optimization.

Micro-benchmarking TICS.

The execution time overhead of TICS with the number of checkpoints for different working stack sizes is given in Figure 4.7. As the working stack size gets bigger, the number of working stack change driven checkpoints decreases since the on-demand stack requirement of the applications are fulfilled—S2 configuration did not lead to a working stack changes and in turn checkpoints and S1 led to a considerable number of working stack changes and therefore also more checkpoints. On the other hand, increasing the working stack size also increases the overhead of a single checkpoint since the logged data is bigger—there will always be a trade-off. Among benchmarking applications, AR led to a considerable amount of working stack change driven checkpoints with configuration

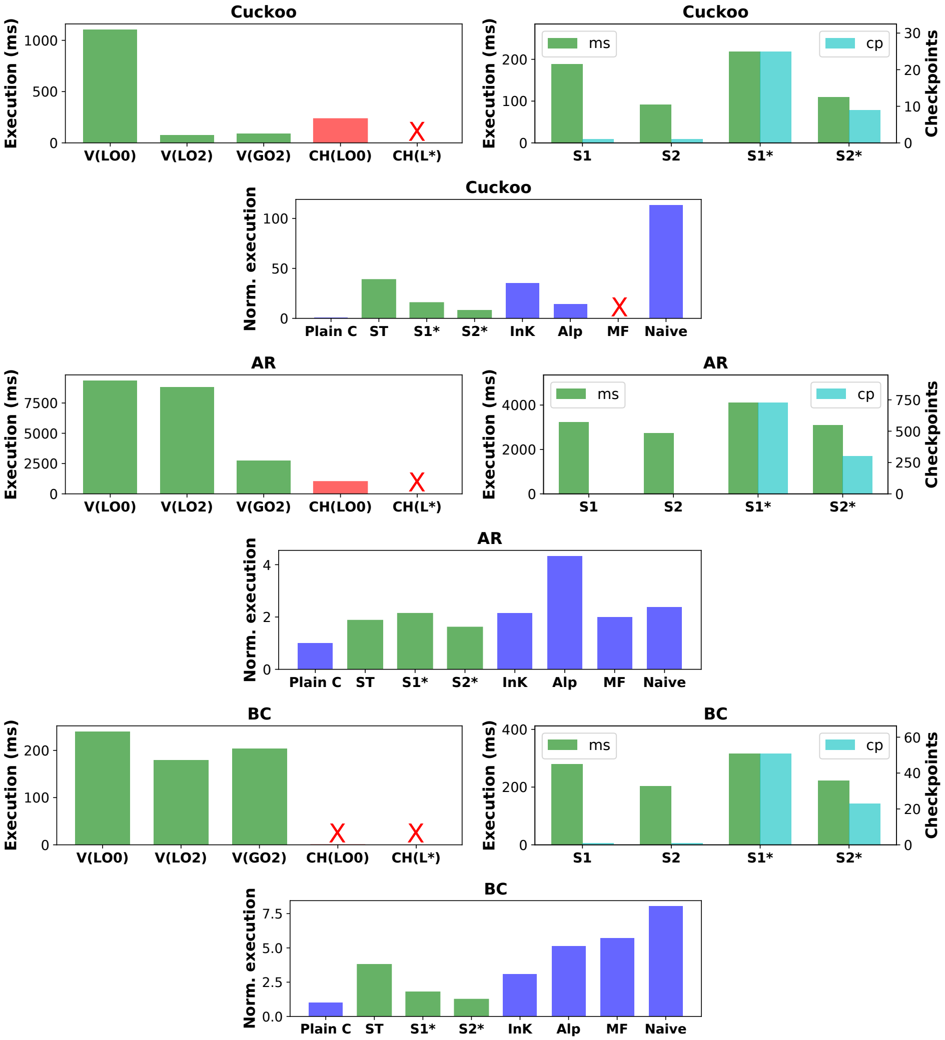


Figure 4.7: **Benchmark performance.** *Firs*: TICS to Chinchilla comparison; *Second*: micro-benchmarking of TICS; *Third*: TICS to task-based systems comparison. **V**, **CH**, **Alp**, **MF**: TICS, Chinchilla, Alpaca, MayFly; **LO0, LO2**: LLVM-compiled code with –O0, and –O2 optimization. **L***: all compilations options of GCC and all of LLVM except –O0. **GO2**: GCC-complied code with –O2 optimization. **S1, S2**: configurations with different working stack sizes imposing a different checkpoint frequency and checkpoint time—**S1**=50 B, **S2**=256 B; **S1*****, S2***: the same stack configurations as **S1, S2** but with an additional timer checkpointing every 10 ms if there was no checkpoint due to the working stack. **ST** denotes **S2** with checkpoints at the task boundaries. Red cross (✗) denotes the code did not compile with the chosen compiler/optimization.

	InK		Chinchilla		TICS	
	.text	*.data*	*.text*	*.data*	*.text*	*.data*
AR	3442	4459	12870	8986	6878	1364
BC	2922	4433	10902	8658	5944	1488
CF	2648	4693	12128	9050	7178	1948

Table 4.3: **The memory consumption (in B) for three applications written in InK, Chinchilla and TICS.**

Operation	**Configuration Variables**	**Duration** (µs)
Stack grow/shrink		max 345
Checkpoint logic	0 B seg. | 64 B seg. | 256 B seg.	264 | 464 | 656
Restore logic	0 B seg. | 64 B seg. | 256 B seg.	273 | 475 | 664
Pointer access	no log | log 4 B (64 B)	13 | 308 (371)
Roll back from undo log	4 B | 64 B	234 | 294

Table 4.4: **TICS overhead, split per runtime operation.** Results obtained with GCC (optimization –O2) at 1 MHz.

S1 due to its varying stack size requirements. We also enabled timer-driven checkpoints with a frequency of 10 ms that ensure the forward progress—configurations S1* and S2* indicate the configurations S1 and S2 with timer-driven checkpoints enabled. TICS checkpoints do not introduce significant overhead since only the working stack and registers are logged.

TICS Against Task-based Systems.

We selected configurations S1* and S2* to asses the execution time performance of TICS considering the task-based runtimes—right column of Figure 4.7. For the fairness of comparison against task-based systems, apart from timer-driven checkpoints in S1* and S2*, we placed checkpoints to configuration S2 at task-boundaries for TICS (shown as ST) and our naïve MementOS-like [203] implementations. We observed that by selecting a reasonable working stack size, TICS reaches almost the performance of existing task-based systems.

TICS Memory Overhead.

Table 4.3 presents a comparison of memory overhead of the benchmarking applications implemented in InK (task-based system), Chinchilla (checkpoint-based system) and TICS. The `.data` section overhead of TICS depends on the size of the configurable stack segment array (which was 2048 B) and undo log (which was 2048 B). Both are excluded from the `.data` section. The code size in selected applications is dependent on not only the application source but also on the stack segmentation and memory consistency management implementations in TICS. Overall, we see that for all benchmarks TICS has *significantly lower* memory overhead than Chinchilla—more than twice `.text` and more than six times for `.data`. compared to InK, TICS `.data` is also significantly lower, except for `.text`.

TICS Point-to-point Overheads.

Table 4.4 presents the detailed overhead of TICS runtime operations. The checkpoint and restore operations include saving registers and working stack in non-volatile memory using a two-phase commit operation—the working stack size has a direct impact on the checkpointing overhead. The constant checkpoint overhead without saving the working stack segment is depicted as 0 B size in the table. The *stack grow/shrink* operations update the working stack to point to another segment in the segment array. During pointer manipulation, TICS checks the pointer address to see if the working stack is targeted. If this is the case, there is no need for the undo logging and the working is stack directly manipulated. Otherwise, TICS logs the original value in undo log—the overhead of different variable sizes is depicted in the table. The time it takes to recover the original value of a variable from the undo log depends on the variable sizes.

4.4.4. User Study and Developer Effort

We have designed a large online user study. The goal was to objectively assess the time to design a TICS application.

Methodology. At the beginning of an online survey each participant was given an introduction to intermittent execution and to TICS and InK [250]. Then, we have then asked participants to find bugs in three simple programs: (i) *swap of two variables (with no use of a temporary variable)*, (ii) *bubble sort*, and (iii) *program that considers variable expiration based on time*. Each program was written separately in TICS and in InK and had *exactly the same type of bug*, at *exactly one line of the program*. Users were asked to point to a line that contained that bug and specify the correct statement.

Each program with a bug was presented to a user on a separate page. Additionally, *time* spent on finding a bug in each of the programs was measured. No corrections of the given answers were possible once the answer was submitted. We randomized the order in which each program appeared at the respondent's screen in order to remove presentation bias against one language and objectify bug finding time.

User Pool. A total of 90 responses were collected. 78% of all respondents had at least two years of programming experience. Almost 83% of respondents had average or below average knowledge of embedded systems powered by energy harvesting technologies.

Result. Results are shown in Figure 4.8. We observe that in all cases it was (i) *harder to find a bug* and (ii) *users were more prone to error* when exposed to a task-based language. Statistically, Wilcoxon T Test on all programs' bug search time rejected the hypothesis that TICS/InK results were the same with p-value below 0.001. In other words, TICS is a *more user-friendly system than a task-based one*. As the complexity of a program increased users had difficulty finding a bug in an InK program (for *Bubble Sort* in half of the cases users were wrong). Regarding the subjective evaluation of TICS against InK, participants considered TICS to be *more intuitive, easier and more concise* than InK.

4.5. Related Work

In Table 4.5 key characteristics of TICS are compared to those of Mayfly [86], Alpaca [142], Ratchet [235], Chinchilla [143] and InK [250]. In this section we compare some of these characteristics from the state of the art to TICS.

Checkpointing Systems. Systems that automatically determine checkpoint placement at compile-time like [235, 35, 143] are most closely related to this work. HarvOS parses

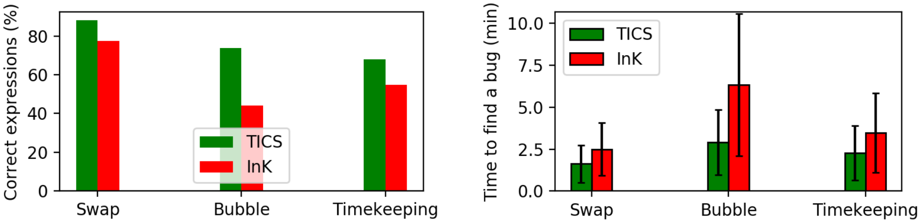


Figure 4.8: **TICS user study results.** For all three test programs, *Swap*, *Bubble* and *Timekeeping*, users found that it is easier with TICS to identify a bug and were more accurate in correcting the TICS program than that of InK [250]. Whiskers in the right-hand side figure denote standard deviation.

Runtime	Pointer Support	Recursion Support	Timely Execution	Porting Effort	MSP430 Support
Mayfly [86]	No ✗	No ✗	Yes ✓	High ✗	Yes ✓
Alpaca [142]	No ✗	No ✗	No ✗	High ✗	Yes ✓
Ratchet [235]	Yes ✓	Yes ✓	No ✗	Low ✓	No ✗
Chinchilla [143]	Yes ✓	No ✗	No ✗	Low ✓	Yes ✓
Ink [250]	No ✗	No ✗	Yes ✓	High ✗	Yes ✓
TICS (*this work*)	**Yes ✓**	**Yes ✓**	**Yes ✓**	**Low ✓**	Yes ✓

Table 4.5: **State of the art programming models.**

the control flow graph of a program and instruments with energy-aware checkpoints, requiring a small amount of programmer intervention to place effectively. Ratchet functions by placing checkpoints at the boundaries of idempotent sequences of instructions. Chinchilla over-instruments programs with checkpoints by storing some variables in non-volatile memory, and disabling/enabling checkpoints heuristically. Apart from the aforementioned studies, Mementos [203] was the first checkpointing scheme, using intermittent voltage checks to decide when to save state. QuickRecall [103] and Hibernus [31] extended this work with newer non-volatile memories. DINO [136] laid out the memory consistency problems that will arise with intermittent computing for mixed-volatility processors. TICS builds on these early techniques. However, these systems do not consider timely execution of the applications.

Task-based Programming Models. Alpaca [142] and related works [47] focus on providing control flow and data flow mechanisms while reducing the memory footprint from multi-versioning. Mayfly [86] provides explicit semantics for specifying timing constraints on sensor data in a task-based language. InK [250] provides a way to handle events and interrupts from clock sources, sensors, and energy in the environment, despite power failures. Task-based systems require a custom programming model, which leads to added programmer intervention and complexity. Task decomposition is a manual process that is error-prone and not resilient to changes in the availability of energy in the environment.

Non-volatile Processors. Integration of non-volatile components, e.g. non-volatile registers, to the processor architecture provides automatic management of forward progress and memory consistency [139, 138]. This eliminates the need for handling these properties explicitly by the programmer. However, non-volatile architectures consume more power, and they have increased area and decreased frequency as compared to general-purpose volatile processors with SRAM-based flip-flops [88]. TICS targets off-the-shelf processors with hybrid volatile and non-volatile memory in the market.

4.6. Discussion and Future Work

In the future, we anticipate exploring ways to automatically import or infer timing semantics and rules from legacy code in TinyOS or other systems. Additionally, virtualizing the I/O interface across power failures could also lead to better ported applications.

4.7. Conclusions

TICS is a runtime for intermittently powered systems that enables the full use of C features like pointers and recursion through a memory consistency management scheme (data versioning and stack segmentation) and provides semantics for easily porting time-*sensitive* programs to the intermittent domain while maintaining correctness. Guarantees on worst case checkpointing time are provided, ensuring TICS scales as applications become more complex. We evaluated TICS against the state of the art, showing reasonable overhead nearly matching the performance of task-based systems. We conducted a user study, where participants found TICS more intuitive than the task-based approach.

5

Avoiding Checkpoints Using Instruction Rescheduling

In the previous chapter, we introduced a method to undo modifications to the non-volatile main memory by including part of it in the checkpoint through an active stack region and logging all accesses outside of the active stack region, and undoing them when the power fails. This method focused on restoring the main memory after a power failure and creating fewer checkpoints. In turn, the approach in the previous chapter increased the checkpoint time as more data needed to be copied to and from the non-volatile memory, as the checkpoint also included the active stack segment in addition to the registers.

In this chapter, we introduce and improve upon an alternative approach. Instead of restoring the non-volatile memory to match the state it was in when the checkpoint was created upon a power failure, we place the checkpoints so that the memory never needs to be restored. We do this by introducing checkpoints between each Write-After-Read (WAR) operation to the non-volatile memory to create idempotent sections. Traditionally, this technique introduces many more checkpoints compared to alternative approaches, but the checkpoints only consist of the registers and thus are small. In this chapter, we significantly reduce this overhead by introducing multiple compiler optimizations that reorder memory accesses in such a way as to reduce the number of required checkpoints.

This chapter is based on the following publication:
Vito Kortbeek, Souradip Ghosh, Josiah Hester, Simone Campanoni, and Przemysław Pawełczak. *WARio: Efficient Code Generation for Intermittent Computing*. In Proceedings of the International Conference on Programming Language Design and Implementation (PLDI), pages 777–991, June 2022. https://doi.org/10.1145/3519939.3523454.
The accompanying archive containing all the software and data can be found at https://doi.org/10.5281/zenodo.6413018.

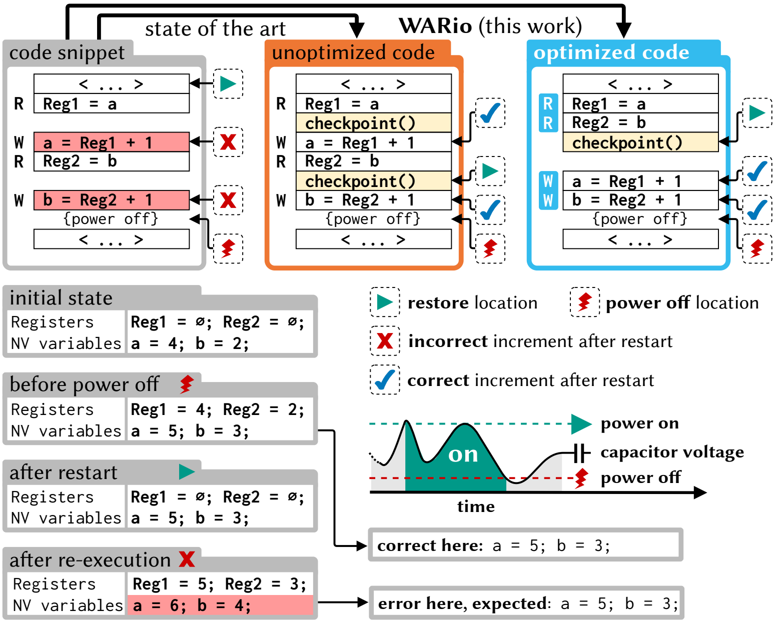
5.1. Introduction

As discussed in Chapter 1, capacitors hold orders of magnitude less energy than batteries, which means that their energy supply is intermittent as they must recharge. Therefore, power failures are common, causing computational intermittency [135]. The intermittent operation causes the computational state to be lost unless explicitly saved in Non-Volatile Memory before a power failure and restored afterward.

As discussed in the beginning of part two of this thesis, relying on Non-Volatile (NV) memories (such as FRAM or MRAM) significantly reduces the cost of a single checkpoint by saving only the (live) registers. However, state-of-the-art static solutions using NV main memory require frequent checkpoints, often at the basic block level. Moreover, selective (instruction-level) checkpoint placement must account for the unique problem present in contemporary (and future) Microcontroller Unit (MCU) architectures that use non-volatile main memory: *Non-Volatile memory corruption in variable manipulations with WAR dependencies caused by re-execution*. This problem, often referred to as a WAR violation [148, 235], was first observed in [202] and is schematically presented in Figure 5.1. Throughout this chapter, we use the term *WAR violation* (or simply WAR) to refer to these possible memory corruption locations which are caused by re-execution following a power failure. When we refer to *resolving a WAR*, we refer to the placement of a checkpoint between its 'Read' and 'Write' to create two distinct idempotent regions.

Problem Statement. A state-of-the-art approach is to detect idempotent regions by looking for instruction sequences that perform a WAR to the same memory address, and placing checkpoints at the boundaries of these regions [235], Figure 5.1 (middle). Nonetheless, strategic checkpoint placement of [235], which performs this task automatically at compile time, does not perform any transformations to *reduce the number of introduced checkpoints* (which are often over-instrumented). Our fundamental insight is that in a code region with many consecutive WAR violations, moving the 'Write' operations belonging to these WARs to a later stage in the code, i.e., *clustering* them, will reduce the number of checkpoints needed, thereby increasing the performance of intermittent computing. The more consecutive (unrelated) WAR operations—the more benefit from checkpoint reordering, which reduces the execution time. This reduction is clearly seen in Figure 5.1 (right), *halving* the number of checkpoints inserted by [235] (Figure 5.1 (middle)). To implement the transformations mentioned above, we use NOELLE [152], an LLVM [180] plugin that uses alias analysis [219, 19] to compute a PDG (among other information).

Our Contributions. We present WARio, **W**rite **A**fter **R**ead **I**ntermittent-computing **O**ptimizer, a set of compiler transformations for intermittently-executed programs to reduce checkpoint overhead. WARio builds upon the techniques introduced in Ratchet [235] and operates both in the middle end and the back end of the compiler. In the middle end, ① WARio introduces *two novel algorithms that cluster the 'Write' operations* of several WARs to reduce the required number of checkpoints. In the back end, ② WARio reduces the number of checkpoints by introducing a *hitting set algorithm to select the checkpoint locations to resolve back-end WARs* (in addition to the existing hitting set in the middle end [235]) and by ③ *protecting stack pointer modifications* in a novel way that requires fewer checkpoints.



code snippet

```
  < ... >
R Reg1 = a
W a = Reg1 + 1
R Reg2 = b
W b = Reg2 + 1
  {power off}
  < ... >
```

unoptimized code

```
  < ... >
R Reg1 = a
  checkpoint()
W a = Reg1 + 1
R Reg2 = b
  checkpoint()
W b = Reg2 + 1
  {power off}
  < ... >
```

optimized code

```
  < ... >
R Reg1 = a
R Reg2 = b
  checkpoint()
W a = Reg1 + 1
W b = Reg2 + 1
  {power off}
  < ... >
```

initial state

Registers	Reg1 = ∅; Reg2 = ∅;
NV variables	a = 4; b = 2;

before power off

Registers	Reg1 = 4; Reg2 = 2;
NV variables	a = 5; b = 3;

after restart

Registers	Reg1 = ∅; Reg2 = ∅;
NV variables	a = 5; b = 3;

after re-execution

Registers	Reg1 = 5; Reg2 = 3;
NV variables	a = 6; b = 4;

correct here: a = 5; b = 3;

error here, expected: a = 5; b = 3;

Figure 5.1: Three versions of the same code snippet demonstrating Non-Volatile Memory corruption, its mitigation, and our optimization. A checkpoint records only the registers (Reg1 and Reg2). The variables (a and b) are in NVM and *not* restored after a power failure. The unprotected code (left) executes until the power failure, reading from and writing to the NV variables. A restart does not undo any modifications to NVM, resulting in incorrect re-execution caused by a Write After Read to the NV variables. By placing a checkpoint of the registers between the *read* (R) and *write* (W) of a WAR, state-of-the-art systems such as Ratchet [235] (center figure, unoptimized code) avoid this memory corruption caused by re-execution. WARio (our work) aims to reduce the number of required checkpoints by *clustering* writes to NVM, reducing the number of required checkpoints (right figure, optimized code).

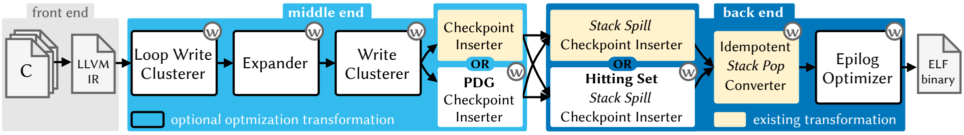


Figure 5.2: WARio architecture. Input plain C code is transformed, through a set of middle and back end compiler transformations described in Section 5.2, to an output Executable and Linkable Format (ELF) binary file that can be executed (guaranteeing no NVM corruption caused by WARs) on an intermittently-powered system. The complete WARio system consists of all the transformations marked with a Ⓦ; other combinations are used to evaluate performance of individual transformations in Section 5.4.2. The transformations marked as *existing* where introduced in prior work [235]. The transformations marked as optional are not needed to avoid WAR violations, but improve the performance by reducing the number of inserted checkpoints.

The transformation steps ①–③ of WARio reduce checkpointing overhead for continuous-checkpointing-based intermittent computation. Compared to Ratchet [235], a state of the art system, WARio reduces the checkpoint overhead by up to 88%, and on average by 58%, considering a broad set of software benchmarks.

5.2. WARio System Design

Addressing the problem presented in Section 5.1 we present WARio. During compilation, WARio performs multiple optimizations targeted at reducing the number of WAR violations in the C code. WARio possesses the following features.

❶ **Support for General Purpose C Programs:** WARio takes a regular embedded C code and automatically transforms it to a WAR-protected executable;

❷ **Oblivious to Energy Conditions:** No prior information on the battery-free system's energy use or input harvested energy is needed prior (and during) compilation into a WAR-protected executable;

❸ **Support for Short Device Activity Times:** WARio guarantees forward progress at short device activity times, i.e. in the order of tens of milliseconds;

❹ **No Programmer Involvement:** WARio does not expect to restructure the program manually to help resolve any WAR dependency; and

❺ **Interrupt Support:** During checkpoint placement WARio makes sure that there can be no WAR violations caused by interrupts pushing information to the stack.

5.2.1. WARio Architecture

WARio targets the following platform: (i) a single processor embedded system (MCU); (ii) direct physical memory access, i.e., no virtual memory; (iii) no data cache, (iv) register access/'bare metal', i.e., no operating system; and (v) non-volatile byte-addressable main memory.

WARio's architecture consists of a set of Intermediate Representation (IR)-based compiler transformations executed in a specific order, as presented in Figure 5.2. All of the components of WARio are described in detail below.

WARio Front End

WARio takes the C code of a project aimed to be run on an intermittently-powered device and converts it to LLVM IR, per each C source file. Subsequently, WARio merges individual IR files into a single (combined) IR of the whole project. We note that both these steps are standard front end compiler transformations (marked as the gray area in Figure 5.2).

WARio Middle End

The core tasks performed by WARio are executed in the middle end. Each of the steps (listed within the light blue area in Figure 5.2) is explained below.

Loop Write Clusterer. This transformation aims at reducing the number of checkpoints in a loop that contains one or more WAR violations. Algorithm 7 shows the pseudocode of this transformation, and Figure 5.3 the resulting IR after each step. Both figures are used to explain the transformation in detail and provide a visual example.

Let us take as an example the unmodified loop code snippet in Figure 5.3. Directly inserting checkpoints, represented by the orange box, results in one checkpoint per iteration i. After applying the `Loop Write Clusterer` transformation, the loop requires only i/N checkpoints when executing, where N is the unroll factor used during the transformation, provided during compilation[1]. It does so by postponing write operations to NVM until the end of the unrolled loop—essentially combining the checkpoints required for N iterations of the loop into a single checkpoint. First, the transformation analyzes the input code using a PDG analyzer, such as [152], to collect all the memory dependencies in the program. The transformation then collects all loops in the program (denoted as L_{all} in Algorithm 7). For each input loop $L \in L_{\text{all}}$ the `Loop Write Clusterer` checks whether the loop is a candidate to be unrolled.

► ***Candidate Selection:*** Not all loops are candidates to have their writes clustered. Most notably, to be a candidate (Line 3 in Algorithm 7), the loop must contain at least one WAR violation to cluster (Line 11). Otherwise, the loop will not have any checkpoints to remove. Additionally, the write cluster insertion point (the destination of the to-be-moved WAR `store` instructions, i.e., the *loop latch*) must post-dominate all the relocated `store` instructions in order not to change the semantics of the loop (Line 13). The final requirement is that the loop does not contain any function calls, as those implicitly cause checkpoints hindering our ability to cluster the writes (Line 11).

► ***Loop Unrolling:*** If a loop is a candidate, Loop Write Clusterer unrolls it N times (Line 4 in Algorithm 7). The IR resulting from the unroll step is shown in Figure 5.3—`UnrollLoop` for an unroll factor of $N = 3$.

► ***Loop Analysis:*** Loop analysis is a necessary operation of `Loop Write Clusterer`, as simply moving all the writes to the loop latch is insufficient to retain the loop semantics. Let us therefore proceed with introducing the analysis steps (Line 17 in Algorithm 7) that will be needed to perform correct code transformation (Line 24 in Algorithm 7). The first step is the obtainment of a loop dependency graph (Line 18 in Algorithm 7), from which

[1] The effect of N on the unrolling effectiveness will be a part of WARio evaluation presented in Section 5.4.2

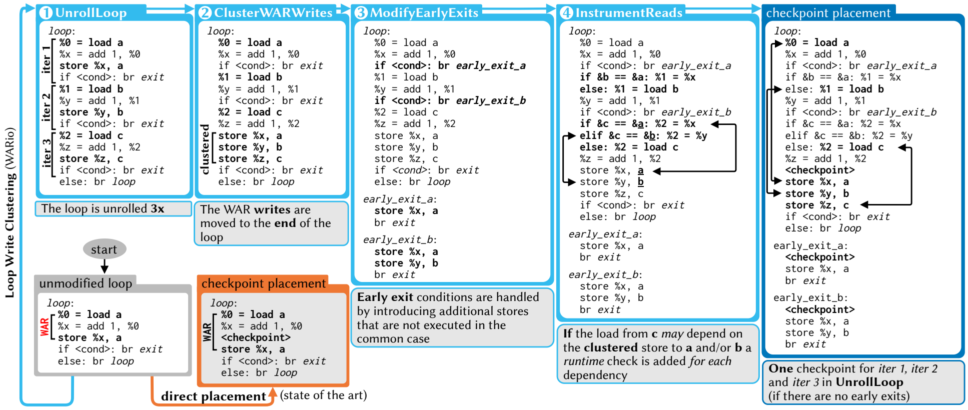
the WAR and Read After Write (RAW) dependencies are obtained (Line 19 and Line 20 in Algorithm 7, respectively).

► ***Clustering WAR Writes:*** Unrolled loops, denoted as L', are passed for analysis using the PDG information (Line 5 in Algorithm 7), which are later on transformed (Line 6 in Algorithm 7) resulting in a set of WARs that are postponed, resulting in moved WAR writes (`store` instructions) shown in Figure 5.3—`ClusterWarWrites`.

► ***Early-exit Handing:*** When moving all writes to the insertion point, i.e., the loop latch, WARio potentially skips writing those values to NVM due to early exits, e.g., exits introduced due to unrolling. The transformation must guarantee that any early exit (`ModifyExits` in Algorithm 7) that follows a postponed write contains a copy of that postponed write. Otherwise, exiting a loop early (by reaching the desired number of iterations during execution before the end of the unrolled loop) would not execute the postponed write to NVM, invalidating the program execution. Figure 5.3—`ModifyEarlyExits` shows the addition of these postponed writes (`store` instructions) to the early exits.

► ***Dependent Read Handling:*** When postponing all the writes to the loop latch, WARio might attempt to move write instructions past reads that depend on them, for example, due to unrolled loop-carried dependencies. First, the transformation collects the read instructions that might depend on a preceding write instruction, using RAW dependency information collected earlier through the PDG. If any of the reads depend on one or more of the postponed writes that are now no longer dominating the read (i.e., they now happen after the read), they would result in reading incorrect information. Therefore, if the read *may* depend on a postponed write, a runtime check is inserted that compares the source address of the read (`load`) instruction and the destination address of the postponed write (`store`) instruction (Line 37 in Algorithm 7). If these are equal, the read is skipped (i.e., the value is not retrieved from memory), and the register containing the content of the postponed write is copied into the read destination (Line 38 in Algorithm 7). On the other hand, if the addresses are not equal, the original read is performed. It might be the case that a read instruction may be dependent on multiple writes. In this case, the transformation adds checks for each of the writes, passing its output as input to the next check as shown in the `InstrumentReads` procedure in Algorithm 7. Figure 5.3—`InstrumentReads`, shows an example where the `load` of variable *c* *may* depend on the `store` to variables *a* and *b*, which were postponed (worst case). Adding a runtime check introduces overhead, but it is minimal compared to the time it takes to perform a complete checkpoint. However, there is a break-even point as the number of checks added to each read instruction grows depending on the number of aliasing writes before it.

► ***Checkpoint Placement:*** To illustrate the effect of the `Loop Write Clusterer`, the last (dark blue) box in Figure 5.3, shows the final loop IR with the addition of checkpoints. When the loop is executing, the three iterations from the original loop (now unrolled), containing the **three WAR violations**, are resolved with only **one checkpoint** instead of three.



```
loop:
  %0 = load a
  %x = add 1, %0
  store %x, a
  if <cond>: br exit
  %1 = load b
  %y = add 1, %1
  store %y, b
  if <cond>: br exit
  %2 = load c
  %z = add 1, %2
  store %z, c
  if <cond>: br exit
  else: br loop
```

The loop is unrolled **3x**

```
loop:
  %0 = load a
  %x = add 1, %0
  if <cond>: br exit
  %1 = load b
  %y = add 1, %1
  if <cond>: br exit
  %2 = load c
  %z = add 1, %2
  store %x, a
  store %y, b
  store %z, c
  if <cond>: br exit
  else: br loop
```

The WAR **writes** are moved to the **end** of the loop

```
loop:
  %0 = load a
  %x = add 1, %0
  if <cond>: br early_exit_a
  %1 = load b
  %y = add 1, %1
  if <cond>: br early_exit_b
  %2 = load c
  %z = add 1, %2
  store %x, a
  store %y, b
  store %z, c
  if <cond>: br exit
  else: br loop

early_exit_a:
  store %x, a
  br exit

early_exit_b:
  store %x, a
  store %y, b
  br exit
```

Early exit conditions are handled by introducing additional stores that are not executed in the common case

```
loop:
  %0 = load a
  %x = add 1, %0
  if <cond>: br early_exit_a
  if &b == &a: %1 = %x
  else: %1 = load b
  %y = add 1, %1
  if <cond>: br early_exit_b
  if &c == &a: %2 = %x
  elif &c == &b: %2 = %y
  else: %2 = load c
  %z = add 1, %2
  store %x, a
  store %y, b
  store %z, c
  if <cond>: br exit
  else: br loop

early_exit_a:
  store %x, a
  br exit

early_exit_b:
  store %x, a
  store %y, b
  br exit
```

If the load from **c** *may* depend on the **clustered** store to **a** and/or **b** a *runtime* check is added *for each* dependency

```
loop:
  %0 = load a
  %x = add 1, %0
  if <cond>: br early_exit_a
  if &b == &a: %1 = %x
  else: %1 = load b
  %y = add 1, %1
  if <cond>: br early_exit_b
  if &c == &a: %2 = %x
  elif &c == &b: %2 = %y
  else: %2 = load c
  %z = add 1, %2
  <checkpoint>
  store %x, a
  store %y, b
  store %z, c
  if <cond>: br exit
  else: br loop

early_exit_a:
  <checkpoint>
  store %x, a
  br exit

early_exit_b:
  <checkpoint>
  store %x, a
  store %y, b
  br exit
```

One checkpoint for *iter 1, iter 2* and *iter 3* in **UnrollLoop** (if there are no early exits)

```
loop:
  %0 = load a
  %x = add 1, %0
  store %x, a
  if <cond>: br exit
  else: br loop
```

```
loop:
  %0 = load a
  %x = add 1, %0
  <checkpoint>
  store %x, a
  if <cond>: br exit
  else: br loop
```

Figure 5.3: Code blocks of a simplified version of the IR of the loop, where variables starting with a '%' denote registers, '<cond>' is the condition that terminates the loop, 'br' branches to a label. The 'exit' label and IR (not important for the transformation) are omitted. The unmodified loop is directly instrumented with checkpoints (orange box) by Ratchet [235], i.e. the state of the art system. In this example WARio applies the `Loop Write Clusterer` transformation (light blue) to reduce the required checkpoints from one per iteration to one every three iterations, as shown in the last code block (dark blue).

► ***Correctness:*** When clustering—and therefore moving—the WAR writes, we need to take appropriate steps to maintain correctness. First, when moving a write to a later unrolled loop iteration, we must ensure that the postponed writes are written to NVM when the unrolled loop terminates early. The *Early-exit Handling* step guarantees that all writes are executed by adding writebacks to every loop exit. Second, when moving a write, we have to resolve all reads that *may* depend on it, i.e., attempt to read memory from the same address. The *Dependent Read Handling* step assures that no incorrect read will occur by canceling the write rescheduling or adding runtime checks and handling to aliasing reads. Together, these steps force the `Loop Write Clusterer` transformation to be conservative and semantically correct.

Expander. A large number of checkpoints are caused by function calls. Each function call must perform a checkpoint if it can modify any data on the callee stack. However, more significantly, each function (regardless of the number of arguments) needs at least one checkpoint when returning from a function that uses stack memory. The reason for this is that an interrupt might trigger at any time, and the Interrupt Service Routine (ISR) will automatically push (write) information on the stack causing a WAR violation (Section 5.2.1—Paragraph `Epilog Optmizer`). Strategically inlining functions more aggressively than usual results in fewer checkpoints caused by function calls and returns. In addition, it aids the succeeding transformation by not having a forced checkpoint location due to the function call.

Write Clusterer. The goal of the `Write Clusterer`, similar to that of the `Loop Write Clusterer`, is to reduce the number of checkpoints inserted by clustering write operations belonging to WAR violations together. Doing so will cause the `Checkpoint Inserter` to resolve more WAR violations using a single checkpoint. Instead of the

Algorithm 7: Loop Write Clusterer

```
1  Algorithm LoopWriteCluster():
2      for $L \in L_{all}$ do                                  // Go through all program's loops
3          if IsCandidate($L$) then                            // See Line 7
4              $L' \leftarrow$ UnrollLoop($L, N$)              // See Section 5.2.1
5              $W_s, R_s, E \leftarrow$ Analyze($L'$)          // See Line 17
6              Transform($W_s, R_s, E$)                        // See Line 24

7  Procedure IsCandidate($L$):
8      $D \leftarrow$ FindDependencies ($L$)                   // Use the PDG
9      $W \leftarrow$ FindWARs ($D$)                           // Find initial WARs
10     $C \leftarrow$ FindFunctionCalls ($L$)                  // Find any function calls
11     if $W \neq \varnothing$ and $C = \varnothing$ then      // If loop has WARs and no calls
12         for $w \in W$ do                                    // For each WAR violation
13             if $L_{latch}$ not post-dominates $w_{write}$ then
14                 return false                                // Loop is not a candidate
15         return true                                         // Loop is a candidate
16     return false                                            // Loop is not a candidate

17 Procedure Analyze($L$):
18     $D \leftarrow$ FindDependencies ($L$)                   // Use the PDG
19     $W \leftarrow$ FindWARs ($D$)                           // Extract WAR violations
20     $R \leftarrow$ FindRAWs ($D$)                           // Extract RAW dependencies
21     $R_s \leftarrow$ ReadsToResolve ($W, R$)                // Reads dependent on WAR writes
22     $E \leftarrow$ ExitsToModify ($W_s$)                    // Exit edges in the loop
23     return $W_s, R_s, E$

24 Procedure Transform($W_s, R_s, E$):
25     PostponeWARs ($W_s$)                                    // Move the WAR writes to loop latch
26     ModifyExits ($E, W_s$)                                  // Handle early exits (Line 28)
27     InstrumentReads ($R_s$)                                 // Handle dependent reads (Line 32)

28 Procedure ModifyExits($E, W_s$):
29     for $w \in W_s$ do                                      // For each WAR violation
           // Exit edges that follow the original write location
30         for $e \in$ ExitEdges $(E, w_{write})$ do
31             copy $w_{\mathrm{write}} \rightarrow e$         // Insert copy of write in exit

32 Procedure InstrumentReads($R_s$):
33     for $r \in R_s$ do                                      // Go through all the dependent reads
34         $r_{\mathrm{final}} \leftarrow r$                   // Track the last instrumented read
35         for $w \in$ AliasingWrites $(r)$ do                 // Writes that alias read
36             if $r$ depends on $w$ then
                   // Create new instructions to handle the read
37                 $\mathrm{cmp_{inst}}$ = NewCompareInstruction($r_{addr}, w_{addr}$)
38                 $\mathrm{sel_{inst}}$ = NewSelectInstruction($sel_{inst}, w_{src}, r_{final}$)
39                 $r_{\mathrm{final}} = \mathrm{sel_{inst}}$  // Track the last read select
40         for $u \in$ usages $r$ do
41             replace $u$ with $r_{\mathrm{final}}$           // Replace with final read select
```

aggressively clustering used by the `Loop Write Clusterer`, the write cluster does not insert any runtime checks. The `Write Clusterer` analyses the individual basic blocks of the IR and looks for instances such as in Figure 5.1 (left), where multiple WAR violations are not dependent on each other. If this is the case, the `Write Clusterer` clusters the writes of the WAR violations as in Figure 5.1 (right). Doing so reduces the number of required checkpoints by handling multiple WAR violations with one checkpoint.

PDG Checkpoint Inserter. After transforming the IR during the previously described transformations, the next and final step is to insert checkpoints to break all the remaining WAR violations. The goal of a checkpoint is to save the current volatile state of the system in a way that it can continue operation after a power failure at that point. A checkpoint saves all the volatile-state of the system in NVM. For WARio, a checkpoint contains only the state of the registers, as the main memory is completely NV. Doing this is a multi-step process similar to that of [235]. For each function in the program, the transformation collects all the WAR dependencies. Next, the transformation collects all the locations of forced checkpoints, e.g., at function calls, and removes WAR violations resolved by these forced checkpoints. The remaining WAR violations are resolved by inserting checkpoints between the read and the write of a WAR violation. Where to place a checkpoint is a crucial decision, as a single checkpoint can resolve multiple WAR violations if placed correctly. The transformation converts each of the remaining WAR violations to a set of locations that resolve that WAR violation. Next, a cost is associated with all the potential checkpoint locations, primarily depending on the loop depth. The resulting sets of potential locations are used in a greedy minimal hitting set algorithm [53, Section 4.2.1] to find a set of checkpoint locations that resolve all the WAR violations. This technique was also used by Ratchet [235]. Both write postponing transformations discussed before are effective because they reschedule the write instructions so that the hitting set algorithm can resolve multiple WAR violations with a single checkpoint. Therefore, the hitting set algorithm would result in fewer overall checkpoint locations and is integral to the system's performance.

WARio Back End

The final steps of the code transformation are performed by the back end. All steps (listed within the dark blue area in Figure 5.2) are explained below.

Hitting Set *Stack Spill* Checkpoint Inserter. Up to this point, WARio targeted memory dependencies in the middle end of the compiler. However, to safely support intermittent execution, all WARs to NVM must be handled with a checkpoint, including those that arise in the compiler's back end. During the register allocation phase, the back end may run out of empty registers and move (spill) some of these registers to a stack slot on the stack. The accesses to the NV stack can introduce new WAR violations. These WAR violations are resolved by first forcing the compiler not to reuse any stack slot during the register allocations phase; after this, only a loop can cause a write after a read to one of these slots. Instead of placing a checkpoint before a write to a stack slot that causes a WAR, as is the case in Ratchet [235]. WARio's `Hitting Set` *`Stack Spill`* `Checkpoint Inserter` handles inserting checkpoints by applying the same algorithm as the middle end. A minimum hitting set algorithm [53, Section 4.2.1] selects the checkpoint locations, not using memory information provided by the PDG, as this information is not available during this compilation stage, but by using the known stack slot locations. Strategically placing the checkpoints to handle more WAR violations per checkpoint dramatically reduces the number of checkpoints introduced in the back end, caused by the register pressure increases following `Write Clusterer` and `Loop Write Clusterer` transformations. It is, therefore, a vital component of WARio that allows the checkpoint reduction achieved in the middle end to propagate through the back end.

Idempotent *Stack Pop* Converter. The remaining WAR violations caused in the back end are due to pop instructions. When executing a pop instruction, the stack variables are first loaded (read) into registers, and then the stack pointer is adjusted. Assuming an interrupt happens, the processor will automatically push some registers on the stack and jump to the interrupt service routine. The act of pushing (writing) data to the stack causes a WAR violation concerning the stack. Resolving these WAR violations is done the same way as in Ratchet [235], breaking all pop instructions into (i) first loading the memory into registers, then (ii) performing a checkpoint, and finally (iii) adjusting the stack pointer.

Epilog Optimizer. Because of the aforementioned checkpoints required to absolve all the pop instructions from WAR violations, the epilog of any function contains at least one checkpoint whenever it uses stack memory. However, often the stack pointer is not adjusted in one go when a function returns. Factors such as the use of a frame pointer and other back end implementation-specific causes can induce more stack pointer adjustments, leading to an equal number of additional checkpoints. As a final transformation just before the code generation phase, WARio analyzes the epilogs of all the functions and will reduce the required number of checkpoints during the epilog to just one, whenever possible. It does so by temporarily postponing any incoming interrupts until after the stack adjustment, eliminating the chance of an interrupt allocating on the stack and therefore eliminating WAR violations. Doing this will result in a longer delay between the interrupt arrival and handling. However, the delay consists of only a small amount of instructions.[2] This epilog optimization results in only one inserted function epilog checkpoint before the last stack pointer adjustment to avoid interrupt-related WAR violations, **instead of up to three** in [235], reducing the penalty of function calls.

5.3. WARio Implementation

We now proceed with the implementation details of WARio's architecture presented in Section 5.2.

5.3.1. Target Architecture

We implemented WARio for the popular 32-bit ARM Cortex-M processor architecture [26], but with on-chip mixed (volatile and non-volatile) main memory, such as the recent Ambiq Apollo4 Blue [17]. WARio's main memory resides in the NVM, including all global- and stack-allocated variables. Only the processor configuration, e.g., peripheral configurations, and the registers, are volatile. Therefore, only the register's state is being stored

[2] As WARio targets intermittent computing, where the device might power off at any time, this delay in interrupt handling is not a concern.

during a checkpoint.[3]

5.3.2. Selected Compiler and PDG Analyzer

We chose LLVM version 9.0.1 [180] as the compiler on top of which WARio is built. For the PDG analysis and loop transformation abstractions WARio uses NOELLE [152] (commit `fc36051`).

5.3.3. WARio Middle End Transformations

We proceed with the description of all IR transformations performed by WARio.

Loop Write Clusterer. Using abstractions provided by NOELLE this transformation iterates over all loops in the program. For each loop, it performs the algorithm described in 5.2.1—Paragraph `Loop Write Clusterer`. The unrolling factor N is a compile-time flag provided to WARio. The default unroll factor used to assess WARio performance is $N = 8$, which we found experimentally—refer to Section 5.4.2.

Expander. This transformation goes over all the functions in the input program twice. Firstly, it creates a list of functions containing pointers. These functions are candidates to be inlined, as they might aid in the later transformations. Secondly, the `Expander` goes through all the calls in every function. If a function call is in a loop without any sub-loops—and appears in the list of candidate functions—the `Expander` inlines the function call into the caller.

Write Clusterer. This transformation uses the WAR violation results from the PDG to collect potential WAR clustering candidates. The WAR writes (`store` instructions of LLVM) of these WAR violations are then clustered as described in Section 5.2.1—Paragraph Write Clusterer.

PDG Checkpoint Insterter. Finding the checkpoint locations happens as described in Section 5.2.1—Paragraph PDG Checkpoint Inserter. The transformation uses PDG information provided by NOELLE to find the WAR violations in the program. Next, the transformation inserts checkpoint intrinsics, i.e., special placeholder instructions that signify the back end to insert a checkpoint at that location, at all the checkpoint locations selected by the hitting set algorithm.

5.3.4. WARio Back End Transformations

We need to stress that inserting checkpoints to avoid WAR violations to physical NVM is a task 'close' to the actual hardware, which can only be handled by the back end. Not all WAR violations can be discovered and resolved in the middle end of the compiler. Therefore, the final step is to resolve all the WAR violations in the compiler's back end. The reason for not resolving all WARs directly in the back end is that information on, e.g., detailed memory dependency from the PDG, is accessible only in the middle end.

Hitting Set Stack Spill Checkpoint Inserter. The first cause of WAR violations in the back end is handled by the `Hitting Set Stack Spill Checkpoint Inserter`, which occurs after the register allocation. During the `LLVM` register allocation, as in Ratchet [235], the `-no-stack-slot-sharing` option is used to disallow the reuse of stack

[3] We emphasize that peripherals are not addressed in this work. We refer to Section 5.5 for further discussion.

slots. The remaining stack spills can only cause a WAR violation *if* they occur in a loop, caused by re-executing a basic block that re-uses the stack slot. The transformation goes through all the stack slot accesses in the LLVM Machine IR and checks for non-handled WAR violations, i.e., violations not already handled by checkpoints inserted in the middle end. Instead of inserting checkpoints before the stack slot writes of remaining WARs, as in Ratchet [235, Section 4.1], WARio implements a minimum hitting set algorithm similar to what is used in the middle end (Section 5.4—Paragraph PDG Checkpoint Inserter) to reduce the required checkpoints needed to eliminate all WARs.

Idempotent *Stack Pop* Converter. The other cause of WAR violations in the back end occurs during the final frame lowering step as discussed in Section 41—Paragraph Idempotent Stack Pop Converter. WARio implements this step for the Thumb-2 [23] back end in LLVM, instead of the Thumb back end used in Ratchet (as we found out in its source code [89]), in order to support the Cortex-M [26].

Epilog Optimizer. The Thumb-2 back end in LLVM inserts up to three different stack pointer modifications during the epilog of a function to restore (i) callee saved registers, (ii) the frame pointer, and (iii) other allocated stack memory. To handle all these potential WAR violations with a single checkpoint instead of three, we exploit a trait of target Cortex-M architecture. Namely, (i) temporarily disabling the global interrupts before the stack-pointer adjustment, and (ii) and re-enabling them afterwards. During the period where the interrupts are disabled, which usually lasts a few instructions, interrupts are not lost but set as pending. After the interrupts are re-enabled, any pending interrupt will trigger.

5.3.5. Checkpoints

All the previously discussed transformations do not actually insert checkpoint calls directly. Instead, they insert checkpoint intrinsics which happens just before the code generation in the compiler's back end. The checkpoints themselves are assembly routines. As the main memory is NV, the checkpoint only includes the current state of the (live) registers. However, one can not simply copy the content of the registers to a reserved location in NVM, as this would lead to a corrupt checkpoint if the power fails during the creation of said checkpoint. Instead, in order to be incorruptible, the checkpoint has to be double buffered in NVM, as in other software support systems for intermittently-powered devices, e.g. [119, Section 3], [250, Section 3.4].

5.3.6. Compilation Process

Creating the intermittently-executable code is as simple as replacing LLVM's `clang` [178] with our dedicated WARio compilation script, denoted as `iclang`. `iclang` orchestrates the different compiler transformations without any user intervention. Within `iclang` the programmer can also specify a compilation path that can be selected from all possible ones shown in Figure 5.2. `iclang` compiles the C program without any transformations using `gllvm` version 1.3.0 [184]. This compilation stage creates the whole-program IR file from multiple C project files which is then used as an input to the WARio. Additionally, before the `Loop Write Clusterer`, a basic inlining transformation (using LLVM-specific `opt -always-inline -inline` command) is executed. Also, before the `Expander` transformation the user-specified optimization level (e.g., `-O2`, `-O3`) is applied.

After all needed transformations the WARio generates the ELF program binary, which can be then executed on an intermittenlty-powered device.

5.4. WARio Evaluation

We now proceed with the evaluation of WARio vis-á-vis state-of-the-art compiler-based software systems for intermittently-powered devices. WARio, together with all supporting code to gather and process the evaluation results is available via an open-source repository [174] and as an artifact [173].

5.4.1. Evaluation Setup

We begin with the outline. We will justify implementation choices aimed at the correct assessment of WARio.

Target Processor Platform

WARio performance was measured using a custom-built emulator for ARM Cortex-M processors with on-chip byte addressable NVM. During WARio's development, the only such processor announced commercially was the Ambiq Apollo4 Blue [17], which was not yet available at the time of writing this article due to the ongoing chip shortage that started in 2021 [32].

Why Processor Emulation is Needed The reason for using emulation is threefold. First, an emulator enables us to collect detailed information about the processor status without inserting additional code for data collection (such as variable increments at a traced event). Such code inserts would alter the evaluation results on actual hardware. Simply, these new variables manipulations would introduce additional WAR dependencies to resolve, which were not part of the input benchmark code and should therefore not be counted. Second, emulation enables us to verify the absence of WAR violations during execution by checking all memory accesses in the emulator. Finally, it allows us to evaluate WARio without requiring the delayed Ambiq Apollo4 Blue. We emphasize that processor emulation is a common assessment strategy in many works targeting software systems for intermittently-powered devices. Examples are [235, Section 4.2], [88, Section 6], [148, Section 5.1], and [147, Section 6.1].

Emulator Architecture The developed emulator is based on the Unicorn [183] CPU emulator version 1.0.3, which itself is based on the QEMU emulator [181]. Unicorn was selected for reasons of (i) native support of the ARM Cortex-M family [26], (ii) support of the Thumb-2 instruction set [23, Section 1.2.1] (which is needed for ARM Cortex-M) and (iii) ability to extend the emulator with new features. Specifically, the features we built on top of Unicorn are as follows.

► ***Performance Statistics Collection:*** The emulator enables to collect information on (i) the number of executed clock cycles, (ii) the number and cause of checkpoints, (iii) the number of clock cycles between two consecutive checkpoints, and (iv) where checkpoints occurred in the code. For the pipeline refill-based instructions of ARM Cortex-M4 [15, Section 3.3.1] we calculate the approximate number of executed clock cycles using our implementation of the three-stage instruction pipeline used by Cortex-M processors.

► ***WAR Violation Absence Verification:*** Our emulator performs the same verification of the absence of WAR violations as in [148, Section 5.2] with one main modification. The work of [148] checked only the middle end code, excluding the processor specific back end. Our WAR violation verification is built into the emulator, which allows us to detect WAR violations also in the back end and in any assembly code.

Software Benchmarks

The first software benchmark used in the evaluation is CoreMark [59], an industry-grade benchmark for measuring CPU performance in embedded systems. Additionally, we have used the following programs from the MiBench [77] suite: CRC, SHA, and Dijkstra. We have also used picojpeg [67] and Tiny AES [182] to represent two real-world libraries for embedded platforms.

As described in Section 5.3.6, all benchmarks use the same compilation pipeline: from plain C to complete WARio. When a certain transformation is disabled for a specific benchmark compilation (see Section 5.4.1), the IR passes through this specific transformation without any modifications. All benchmarks are compiled using the `-O3` optimization level of LLVM. Furthermore, the loop unroll factor in the `Loop Write Clusterer` transformation is $N = 8$, which we empirically found, as will be presented in Section 5.4.2.

Software Environments

We evaluate all benchmarks, listed in Section 5.4.1, in the following software environments. Justification for our selection of these environments is outlined in Section 5.6.

WARio and its Components Benchmarks are evaluated by a WARio and by WARio with `Expander`. We also evaluate individual transformations of WARio, as listed in Figure 5.2, i.e. `Loop Write Clusterer`, `Expander`, `Write Clusterer` and `Epilog Optimizer`. Note that the `Checkpoint Inserter`, the basic version of the *`Stack Spill`* `Checkpoint Inserter`, and the `Idempotent` *`Stack Pop`* `Converter` transformations are always required to create a program that can execute intermittently and are included in all the other WARio transformations. In addition, the `Hitting Set` *`Stack Spill`* `Checkpoint Inserter` includes optimized checkpoint placement algorithm that uses a minimum hitting set to aid the write clustering transformations (Section 5.2.1—Paragraph `Stack Spill Checkpoint Inserter`). This advanced version is enabled during all WARio benchmarks, except for the `Epilog Optimization` (not to impact its results).

Ratchet Ratchet [235] is the only completely compiler-based software environment for intermittently-powered devices, i.e. operating fully in the middle and back end of the compiler, without runtime memory logging as e.g. [119, 143], or source instrumentation, as e.g. [250, 142, 119]. Ratchet also addresses all features (❶–❺) listed at the beginning of Section 5.2.1. During the evaluation we used an unaltered version of the Ratchet middle end available via [89], and re-implemented the back end to support the Thumb-2 instruction set [23, Section 1.2.1] needed for ARM Cortex-M family [26], as we remarked already in Section 5.4.1.

R-PDG Additionally, we designed and implemented a version of Ratchet [235], denoted as R-PDG, that uses the PDG information provided in NOELLE [152] for checkpoint insertion, instead of the built-in aliasing information available in LLVM. This adaptation to Ratchet is made to evaluate only the effect of WARio transformations while excluding the added benefit of using PDG information.

Non-instrumented Plain C Code Finally, plain C (non-instrumented version) of all benchmarks is executed. They will be treated as the ultimate reference to all benchmarks run in all software environments listed above.

Energy Traces

We evaluate WARio considering the following power supply cases.

► ***Continuous Power:*** This case is required to measure execution time overhead from checkpoint insertions and code transformations for all software environments.

► ***Intermittent Power with Predefined Pattern:*** For a single scenario a fixed power on period is repeated until a given benchmark completes its execution.

► ***Intermittent Power with Measured Traces:*** We have run our emulator following two example empirical voltage traces measured at the output of an actual energy harvester of a battery-free embedded device. The preexisting traces used in our evaluation, available via [201], were initially used in the evaluation of Mementos [203]: one of the first software frameworks for battery-free intermittently-powered devices.

5.4.2. Evaluation Results

With the evaluation setup introduced, we are ready to present the evaluation results of WARio.

Execution Time

First, we measured the execution time for all benchmarks (listed in Section 5.4.1) executed by all software environments (listed in Section 5.4.1). All results were normalized to the execution time of non-instrumented plain C code versions of each benchmark.[4] The results are presented in Figure 5.4.

The core message of this evaluation is that the average execution time for all benchmarks with WARio (blue dashed line in Figure 5.4) is reduced by 45.6% compared to average execution time for Ratchet (gray dotted line in Figure 5.4) and 27.7% compared to R-PDG (gray dashed line in Figure 5.4). Average per-benchmark overhead reduction by using WARio was also significant. WARio with `Expander` reduced the overhead of Ratchet and R-PDG by 58.1% and 44.3%, respectively. The above numbers demonstrate that WARio reduces checkpointing overhead on intermittently powered devices.

[4] Note, however, that C-only code is incapable of maintaining forward progress on intermittently-powered device with volatile/non-volatile memory architecture.

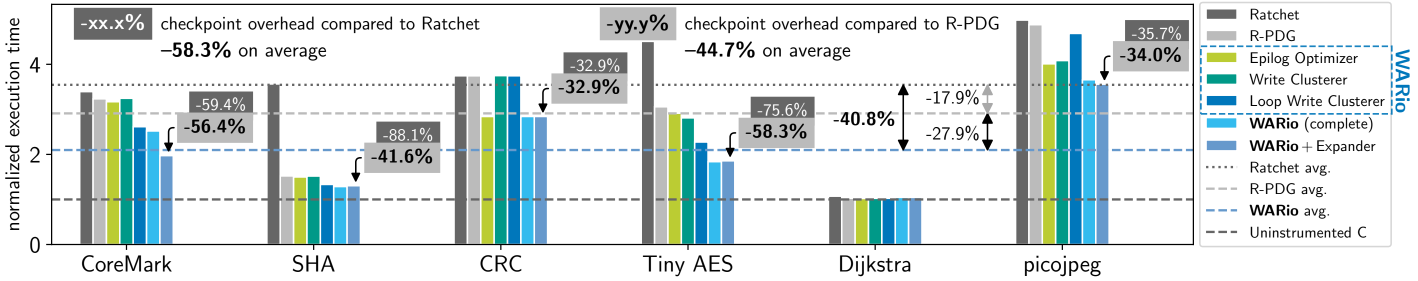


Figure 5.4: Execution time for all benchmarks for Ratchet [89], R-PDG (i.e. improved PDG-based version of Ratchet, see Section 5.4.1) and various components of WARio (per isolated WARio compiler transformation, complete WARio and WARio with `Expander` [see Section 5.2.1]). All results are normalized to the uninstrumented C version of each benchmark, i.e. the lower bound of execution time of each benchmark.

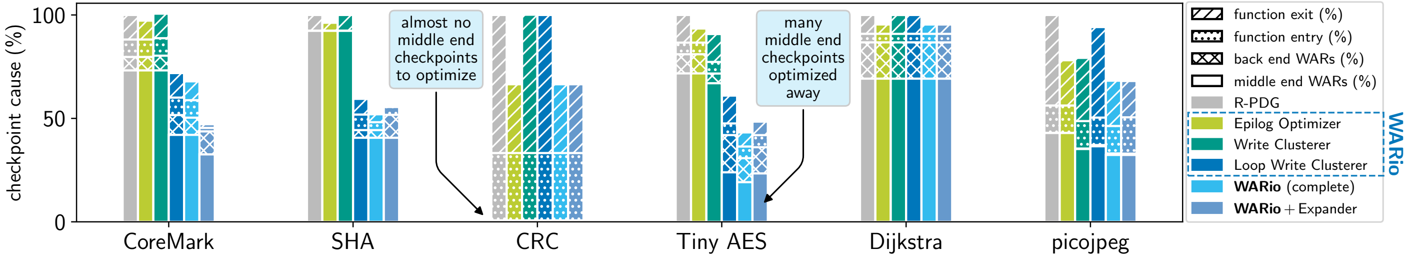


Figure 5.5: Analysis of the checkpoint cause for the corresponding benchmarks presented in Figure 5.4. Each stacked bar, per benchmark, represents the reduction of checkpoints compared to R-PDG, representing 100%. As the bars for Ratchet are excluded from the figure due to the scale difference, the reduction in the number of executed checkpoints compared to Ratchet is shown separately in Table 5.1. Each bar consists of four segments, indicated by the different hashing applied. The different segments depict the checkpoint causes: function exit, function entry, back end, or middle end.

Checkpoint Cause

Figure 5.4 shows also how beneficial each compiler transformation is (see Section 5.4.1). We see that each benchmark benefits differently from each transformation. To shed more light into this observation we gathered more statistics. For the same setup as in Figure 5.4, we recorded the number of inserted checkpoints that were executed and what caused them. The result is presented in Figure 5.5. Specifically, we gathered how many checkpoints were caused by the (i) back end WAR dependency, (ii) middle end WAR dependency, (iii) function entry, and (iv) function exit. Ratchet is not present in Figure 5.5 because the number of checkpoints compared to other software environments listed in Section 5.4.1 is disproportionately high. In other words it is far worse than its improved version R-PDG. Therefore we have used R-PDG as a reference point for the evaluation. In Figure 5.5, R-PDG represents the starting point for each benchmark, i.e., it represents 100% of the checkpoints. Each WARio transformation aims to reduce the number of executed checkpoints relative to R-PDG, represented by the total height of each stacked bar.

Inspecting individual benchmarks, Dijkstra execution time is almost non-visible in Figure 5.4. This is because few WAR violations occur in Dijkstra. This is shown by the data gathered for Dijkstra seen in Figure 5.5, where the number of reduced checkpoints (except for function exit) at each WARio transformation is not decreasing. For CRC, on the other hand, there are no middle end checkpoints to optimize—this is the reason for the smallest improvement from WARio with `Expander` compared to other benchmarks. Benchmarks that benefit most from WARio's write clustering are SHA and Tiny AES, because both benchmarks contain many loop operations. Specifically, for SHA and Tiny AES reduction of middle end WAR checkpoints after the `Loop Write Clusterer` is ≈60% and ≈70%, respectively.

Inspecting individual compiler transformation, the gain from the use of `Expander` is not significant, or is even slightly detrimental, as in the case for Tiny AES. The reason is as follows. `Expander` attempts to guess what functions are good to inline and sometimes this guess is inaccurate (see Section 5.2.1—`Expander`). To really benefit from `Expander`, WARio would need a code profiling information. The `Epilog Optimizer` reduces checkpoints for benchmarks with many exits; CRC benefits from this significantly.

The middle end is the main focus of WARio transformations. These, however, can lead to an increase in register spills due to the increased register pressure. However, as we observe, the reduction in the number of middle-end checkpoints heavily outweighs the increased number of checkpoints in the back end. This is seen in Figure 5.5 for CoreMark, SHA and Tiny AES, comparing the number of back end checkpoints with and without the transformations.

Code Size

Next, we measured the overhead in terms of extra `.text` size in the ELF of (i) Ratchet, (ii) WARio, and (iii) WARio with the `Expander` transformation compared to the non-instrumented (original C) versions. These measurements, presented in Table 5.2, show the code-size penalty associated with WARio's speedup demonstrated in Figure 5.4.

The average code-size increase of Ratchet and WARio are nearly identical. Per-benchmark overhead mainly depends on the number of checkpoints inserted in the

Table 5.1: The difference in total number of executed checkpoints by WARio compared to Ratchet.

	WARio	**WARio + Expander**
CoreMark	-36.6%	-56.0%
SHA	-88.6%	-87.8%
CRC	-33.5%	-33.5%
Tiny AES	-74.5%	-71.5%
Dijkstra	-18.7%	-18.7%
picojpeg	-33.6%	-33.7%
average	-47.6%	-50.2%

Table 5.2: Per-benchmark code-size increase compared to the original C version (without intermittent computing support).

	Ratchet	**WARio**	**WARio + Expander**
CoreMark	+39.6%	+38.7%	+67.9%
SHA	+33.2%	+33.4%	+62.3%
CRC	+8.4%	+7.8%	+7.8%
Tiny AES	+16.2%	+12.1%	+37.7%
Dijkstra	+7.9%	+8.2%	+8.2%
picojpeg	+5.2%	+11.9%	+13.4%
average	+18.4%	+18.7%	+32.9%

code and they are rather consistent between Ratchet and WARio (except for AES—advantageous for WARio and for AES—advantageous for Ratchet). This suggests that not only WARio performs better than Ratchet, and attains this without any extra code footprint penalty. The code size is not significantly affected, even though WARio removes many checkpoints (as demonstrated in Figure 5.5) because a checkpoint is a simple jump instruction to the checkpoint routine. Hence, removing a checkpoint only removes a single instruction from the executable. Additionally, WARio sometimes adds additional instructions while executing the `write clustering` transformations. On the other hand, adding the `Expander` transformation to WARio does increase in the average code size. Note that `Expander` does not always translate to an increase in performance, as seen in Figure 5.4. The reason for `Expander` increases the code is because of inlining function duplicates.

Next, we investigate how large the loop unroll factor N should be. The result are presented in Figure 5.6. For this experiment we chose a subset of benchmarks that benefited most from `Loop Write Clusterer`, i.e. SHA and Tiny AES, see again Figure 5.5. We measured the total number of checkpoints (top part of the figure) and execution time overhead reduction (bottom part of the figure) compared to benchmark with $N = 1$, i.e. no unrolling, as a function of N.

Loop Unroll Factor

The first observation is that as N reaches a certain point, the percentage of checkpoint reduction stalls. Simply, there need to be checkpoints for intermittent systems to work correctly. However, unrolling a selected loop for loop write clustering. On average, a steady state (for both number of checkpoints as well as overhead) is reached when the number of checkpoints in the middle end is reduced from ≈80% to ≈40%. These factors also cause the overhead to fluctuate when the unroll factor N becomes large, as these

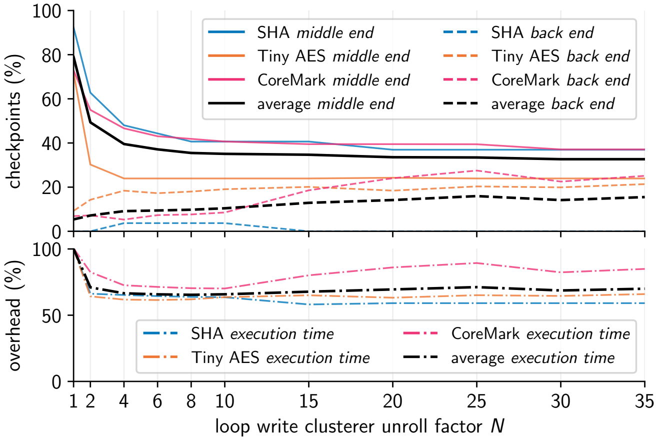


Figure 5.6: The effect of the loop write clustering transformation factor N on the overhead and number of checkpoints for three example benchmarks. $N = 2$ already gives a substantial improvement, while approximately $N = 8$ provides the most benefit.

added checks and checkpoints in the back end will outweigh the reduction of checkpoints in the middle end. The ideal unroll factor for these specific benchmarks appears to be $\approx N = 8$. Therefore, $N = 8$ has been selected for all the other experiments, as we remarked already in Section 5.4.1.

Impact of Power Intermittency

We measured the size of the idempotent sections, i.e., the number of CPU clock cycles between two checkpoints during execution. Figure 5.7 shows the results for Ratchet, R-PDG, and WARio (complete). The median (white line) does not increase significantly. As expected, the 75th percentile (top of the box) and mean (white triangle) increase for most benchmarks. Most importantly, we see that (on average) the maximum idempotent region size is not significantly affected by the removal of over half of the checkpoints. In some cases, e.g., SHA, the maximum idempotent section size did increase dramatically. However, even with this increase, the required power on time is approximately 5.6 ms or 0.9 ms with a processor speed of 8 MHz or 50 MHz, respectively. WARio removes checkpoints at locations where idempotent sections are generally small, e.g., in a loop body or during the epilogue of a function, often leaving the large idempotent sections unmodified. Therefore, WARio does not significantly increase a device's required minimum power-on time to maintain forward progress as compared with Ratchet [235]. We note that additional research is needed to automatically reduce large regions to sustain forward progress for systems requiring even lower minimum power on time. However, the WARs remain protected, preventing inconsistencies due to power failures.

Furthermore, we executed the same benchmarks using different power on/power off patterns, as specified in Section 5.4.1, until completion. The overhead the intermittent execution introduces is composed of three factors: (i) the processor boot procedure execution, (ii) the last successful checkpoint restoration, and (iii) re-execution of the code

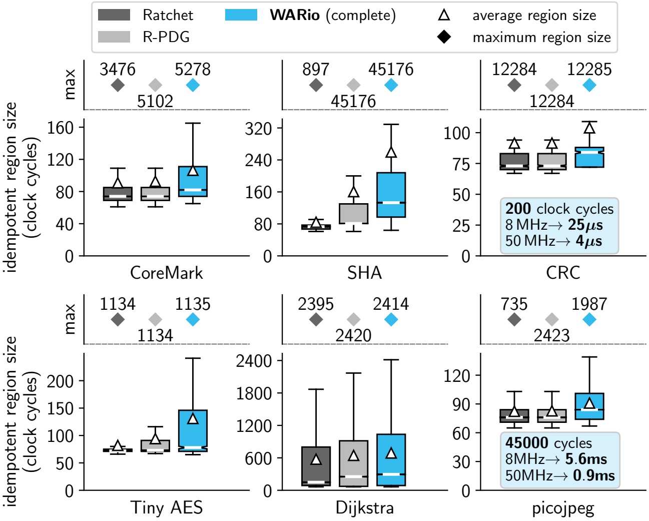
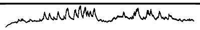
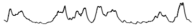


Figure 5.7: Idempotent region size for all considered benchmarks and software environments. Data is presented as a box plot, where maximum values are given at the top of each benchmark's result.

between the last checkpoint and the location of the power failure. The first two factors are constant, but the third factor depends on where the power failure happened in the idempotent region. Table 5.3 shows this overhead as a percentage of the total execution time. For all the benchmarks, this overhead is minimal. Even with very short power on times of 2 ms (at a processor clock speed of 50 MHz), the average overhead is less than 1% compared to continuous power.

Table 5.3: Code re-execution overhead in percentage for WARio with `Expander` compared to the continuously-powered version, $\mathcal{O}$, and number of observed power failures during benchmark execution, $\mathcal{P}$, per different power on cycles.

power *on* duration		**CoreMark**		**SHA**		**CRC**		**Tiny AES**		**Dijkstra**		**picojpeg**	
clock cycles	**time at {8 MHz, 50 MHz}**	$\mathcal{O}$	$\mathcal{P}$	$\mathcal{O}$	$\mathcal{P}$	$\mathcal{O}$	$\mathcal{P}$	$\mathcal{O}$	$\mathcal{P}$	$\mathcal{O}$	$\mathcal{P}$	$\mathcal{O}$	$\mathcal{P}$
50 000	{6.2 ms, 1 ms}	0.24%	127	2.87%	380	7.25%	1	0.23%	7	1.70%	1135	0.18%	2624
100 000	{12.5 ms, 2 ms}	0.14%	63	2.87%	190	0.00%	0	0.09%	3	0.86%	563	0.09%	1310
1 000 000	{125 ms, 20 ms}	0.01%	6	2.78%	19	0.00%	0	0.00%	0	0.07%	55	0.01%	130
5 000 000	{625 ms, 100 ms}	0.00%	1	0.00%	3	0.00%	0	0.00%	0	0.02%	11	0.00%	26
trace $\boldsymbol{\alpha}$		0.00%	3	0.04%	8	0.00%	0	0.00%	0	0.04%	27	0.00%	66
trace $\boldsymbol{\beta}$		0.00%	1	0.00%	2	0.00%	0	0.00%	0	0.01%	5	0.00%	13

✤ Time (for a given processor frequency) is provided as a reference for two example processor clock speeds—8 MHz (i.e. speed at which internal FRAM of TI MSP430 [226] runs on a maximum speed) and 50 MHz.

Table 5.4: WARio compared against state-of-the-art intermittent execution support systems.

system	non-volatile main memory	register-only checkpoint	no runtime memory log	incorruptible	C language support	compiler-based	code-aware	code-transf.	ARM support
Mementos [203]	✗ no	✗ no	✓ yes	✓ yes	✓ yes	✗ no	✗ no	✗ no	✓ yes
MPatch [Chapter 3]	✗ no	✗ no	✗ no	✓ yes	✓ yes	✗ no	✗ no	✗ no	✓ yes
Chinchilla [143]	✓ yes	✓ yes	✗ no	✓ yes	∼ partially†	✓ yes	✗ no	∼ partially	✗ no
TICS [Chapter 4]	✓ yes	✗ no‡	✗ no	✓ yes	✓ yes	✓ yes♠	✗ no	✗ no	✗ no
InK [250]	∼ partially	✓ yes	∼ partially	✓ yes	✗ no✿	✗ no	✗ no	✗ no	✗ no
Rachet [235]	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes	✗ no	✓ yes✤
WARio	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes

† Does not support any form of recursion [119]. ‡ The active stack segment is included in the checkpoint. ♠ Source code instrumentation combined with a segmented stack implementation in the TI MSP430 [226] GCC [179] back end [119]. ✿ A C-style domain specific language for energy-task programming [119, Section 5.4]. ✤ Only Thumb instruction subset, no Thumb-2 support [89].

5.5. Discussion

Location-specific Checkpoints. WARio does not place checkpoints that are user- or application-specific, e.g. to guarantee that inter-checkpoint (idempotent region) time is not larger than certain number of cycles. On the other hand, the number of checkpoints placed by WARio is great enough that extra checkpoints might not be necessary, see Figure 5.7.

Sensing Applications and Use of Peripherals. WARio does not target sensing-based applications, that require interaction with the peripherals. This is a problem that needs to be solved separately, for example using special libraries [117, Section 3.4], which can be used in combination with WARio.

Code Profiling. WARio would benefit from a code profiler. Specifically, code profiling would improve both checkpoint placement and the effectiveness of the `Expander`. We leave the design of code profiling for the future.

Just In Time Checkpoints. Instead of inserting checkpoints to resolve WAR violations, the Just In Time strategy inserts them based on the developer-specified storage capacitor voltage threshold. This strategy brings some downsides. The incoming energy can be highly unpredictable [203, Figure 1], which means that the configured voltage level does not directly correlate to the amount of execution time left.[5] In such a system, the configured voltage threshold must be set to the worst case, as even one missed checkpoint can cause a WAR violation, corrupting the system's memory.

5.6. Related Work

The main (and only) system we can compare WARio to was Ratchet [235]. Nonetheless, this is not the only system available. The most concrete comparison is given in Table 5.4.

Loop Transformations. Early works considering the macro-level idea of instruction relocation and loop unrolling, however with specifics different from WARio, include [57] (in the context of an automatic coarsening of the granularity of locks [by making one lock for multiple objects that can be accessed together] for the data manipulated by a program in a parallel computing system) and [134] (in the context of increasing instruction-level parallelism for processor instruction scheduling). Some volatile memory-based systems, e.g., [248], have introduced counters into loops to check when to create a checkpoint. Sadly, this does not work when the main memory is NV. Some form of loop-result buffering for task-based AI systems programmed using a special Domain-Specific Language (DSL) was introduced in [70]. However, this approach does not work for general-purpose C-based applications.

Extensions of WARio. Other works can enhance WARio by tackling other optimizations. For instance, WARio can 'cache' some data in volatile memory if that data is both generated and used in one idempotent section, as in [147].

[5] The time between reaching the configured voltage level of the comparator, and when the system experiences a power failure, can highly fluctuate, even for a predictable energy harvesting source [117, Section 6.4].

5.7. Conclusions

We have presented WARio: a set of compiler transformations that generate a binary that can be safely executed on intermittently-powered platforms with non-volatile main memory. WARio injects checkpoints to resolve WAR violations but does it only after transforming the input code. WARio moves ‘Write‘ operations from individual WAR operations closer together, and for loops, it applies a novel unrolling algorithm to make this rescheduling of ‘Write‘ operations more impactful. Additionally, WARio adds a hitting-set-based checkpoint placement algorithm in the back end and protects stack pointer modifications by temporarily disabling interrupts. Together these transformations significantly reduce the number of required checkpoints, reducing checkpointing overhead. Nonetheless, there is still a considerable overhead, often around double the execution time or more than running plain, uninstrumented C code. However, this is currently the required cost to allow incorruptible intermittent execution of battery-free applications without additional hardware support.

6

Avoiding Checkpoints Using a Data Cache

We introduced two compiler-based methods to support intermittent computing in the previous two chapters. These systems both work with standard microcontroller systems, with the only exception being that they contain non-volatile main memory. Both systems are software-only approaches and, therefore, still have a non-negligible overhead despite our successful efforts to reduce it. Chapter 4 introduced runtime logging that introduces a check for each memory access and stack segment management that increases the size of a checkpoint by including the active stack. Chapter 5 introduced checkpoints for WAR dependencies that are identified during compile time, which is a very conservative approach as many WAR dependencies will depend on the program's execution. This conservativeness is because compiler-based approaches rely on alias analysis and must insert checkpoints to break all potential WAR violations, even if they don't appear during execution.

In this chapter, we take a different approach. Instead of directly monitoring accesses to the non-volatile main memory, we explore the principle of locality as done in Chapter 4. We take advantage of this principle by introducing a volatile data cache that also addresses the higher memory access cost of non-volatile memory in terms of both energy and latency. We introduce a hardware component that can dynamically detect WAR dependencies, significantly reducing the leading cause of overhead in both software-based approaches. Additionally, we present a novel method to closely integrate WAR detection within the data cache, eliminating the need for additional memory tracking and significantly reducing the number of checkpoints. Finally, we tightly couple this new data cache to the program's execution behavior by tracking the program's stack pointer to further reduce the amount of data written to non-volatile memory.

Vito Kortbeek, Sourav Mohapatra, Saad Ahmed, and Przemysław Pawełczak.
A Data Cache for Intermittent Computing Systems with Non-Volatile Main Memory. Under submission.
The accompanying archive containing all the software and data can be found at https://doi.org/10.4121/22259011.

6.1. INTRODUCTION

One way to mitigate the problem caused by re-execution is to create a *checkpoint* between the read and write of all WAR dependencies, such as in Chapter 5. However, using NVM as the main memory of intermittently-powered devices brings several downsides that greatly reduce the system's performance. First, as discussed in Chapter 1 and at the beginning of Part two of this thesis, WAR dependencies are a common memory phenomenon during the execution of a program. Thus many checkpoints are required to protect the program from power failures—*significantly more* than are needed to allow just the forward progress of the program. These extra checkpoints are due to the conservativeness of compiler-based approaches. Compiler-based approaches must insert checkpoints to break all potential WAR violations because missing one WAR could cause incorrect re-execution. However, not all these speculated WAR violations result in actual WAR violations during execution because the compiler uses alias analysis to find all possible WAR violations. When the compiler detects that a pair of memory access *may* cause a WAR, the compiler *must* insert a checkpoint, even if this sequence of memory accesses might not (always) result in a WAR violation during execution. Additionally, a necessary feature for any intermittent computing framework is to be *incorruptible*. That is, even if the power fails at any time during execution—including during the creation of a checkpoint—the system should continue correctly from the most recent successfully completed checkpoint. Looking at the results of Chapter 5, even an optimized solution has double the execution time compared to native unmodified binaries without checkpoints (while still using NVM as the main memory). Using hardware detection of power failure, such as [88], results in less overhead, as the program does not require to be over-instrumented with checkpoints by the compiler. However, NVMs such as FRAM and MRAM are still considerably slower and require more energy to access than their volatile counterpart SRAM [71, Section 2], [96, Section 8.4]. Therefore the execution time and energy consumption of systems using non-volatile main memory will be considerably higher compared to volatile systems. Hence, intermittent systems would *benefit from finding a balance between using volatile and non-volatile memory*.

6

Problem Statement: Previous works attempted to achieve this balance by reducing the size of checkpoints while still using volatile components, i.e., a mixed-memory model [147]. Another direction is using a volatile *data cache* in combination with non-volatile main memory. Adding a data cache will decrease the cost of using non-volatile main memory by allowing for faster access speeds and fewer NVM accesses. However, integrating a data cache with intermittent systems is not straightforward. The system still needs to address WAR hazards, which become even more complicated in the presence of a cache [252], as the cache delays the actual writeback to NVM. Then, the cache eviction policy that determines which cache block must be evicted to make space for new data (i) must be aware of when the checkpoint will happen and (ii) how to proceed whilst maintaining consistency. Furthermore, since the cache is a volatile entity, it must be written to NVM before a checkpoint. The current state of the art cache-based system, PROWL [91], avoids the chances for memory corruption by creating a checkpoint whenever memory must be written from the cache to the NVM but uses a computationally intensive cache architecture (skew-associative cache with cuckoo-hashing-based eviction policy) that is complicated to implement and potentially energy-consuming.

Our Fundamental Insight: We argue that simply applying existing data cache methods [80, Appendix B] to intermittent computing architectures is inefficient. We take a different position and propose modifications to the cache's workings to better align with intermittent computing. We found that by adding just two bits to the data cache lines and using this information to detect whether a writeback to the NVM is *safe*, we can directly **use the cache for WAR detection** and mitigation to break up WAR dependencies, instead of relying on additional WAR detection hardware [88] or software systems, such the ones presented in chapters 4 and 5. Utilizing the cache as WAR detection reduces the total number of checkpoints, NVM memory accesses, and execution time.

Our Contributions: Based on our insight, we present a new data cache architecture for intermittent computing systems, named NACHO, with the following contributions.
① We define the requirements for a safe data cache in an intermittent computing system with non-volatile main memory. These requirements form the basis of NACHO.
② NACHO, by adding only two extra bits per cache entry combined with a novel algorithm to detect if a write back to memory is *safe*, i.e., not a read-dominated WAR dependency, reduces the number of checkpoints and NVM accesses compared to the state of the art systems.
③ NACHO reduces the checkpoint size by tracking the stack of the executing program and avoids writing memory that is no longer valid to the NVM. All of this reduces checkpointing costs—both in terms of size and time.

Through these contributions we **reduce the overhead** introduced by supporting intermittent computing **on average by 77%** compared to Clank [88] and by **67%** compared to PROWL [91], the state of the art cache-based solution. Additionally, the number of **NVM accesses** is, on average, **reduced by 86%** compared to Clank and **55%** compared to PROWL.

Table 6.1: Features of state of the art intermittent computing systems focussing on ones with a data cache.

	Clank [88]	COACH [92]	ReplayCache [252]	NvMR [34]	PROWL [91]	**NACHO (this work)**
supports data cache	✗	✓	✓	✓	✓	✓
low checkpoint count	✗	✗	✓	✓	✓	✓
low NVM reads/writes	✗	✗	✗	✓	✓	✓
incorruptible	✓	*partially*†	✗	✓	✓	✓
no compiler transformations	✓	✓	✗	✓	✓	✓
cache-agnostic	*n/a*	✓	✗	✓	✗	✓
no extra memory tracker required	*n/a*	✗	✓	✗	✓	✓
tight data cache integration	*n/a*	✗	✗	✗	✗	✓
considers program execution flow	*n/a*	✗	✗	✗	✗	✓

Yes: ✓ , No: ✗ † The work relies on existing checkpointing strategies, thus it can be as incorruptible as the choice of strategy.

6.2. Cache and Intermittency

The size of volatile and non-volatile memory components in an embedded battery-free system greatly impacts the total execution time and consumed energy [91, Table 1]. Finding the right trade-off between the number of volatile and non-volatile components *motivates using cache for fast and energy-efficient intermittent system architectures.*

6.2.1. Trade-off in Memory Persistency

In the case of completely non-volatile systems such as a non-volatile processor [138], everything is in persistent storage, and the need to create checkpoints and restore them in case of power failure disappears. However, the energy consumed to perform memory accesses becomes high, diminishing the gains resulting from memory non-volatility. As we move towards the opposite spectrum of fully volatile architectures, the energy consumed per cycle decreases, but the size and number of the checkpoints and the cost of re-execution increase. Everything needs to be saved and restored to/from a fully volatile system, again skewing the associated costs. The desired solution is thus a *balance between the volatile and non-volatile systems.* That is, we should seek a solution where a volatile SRAM-based data cache provides high access speeds—while being small enough to ensure low checkpointing overheads—and where a non-volatile main memory acts as the persistent entity to ensure data retention across power failures.

6.2.2. Challenges of Intermittency and Cache

Integrating a cache into an intermittent system is not straightforward. Let us look at an example in Figure 6.1, where we compare a traditional system supporting intermittent operation with a data cache-based system. In Figure 6.1, case ②, because the checkpoint placement logic depends on when a "write" to NVM is performed, having a cache that buffers the memory accesses delays the write operation to NVM. This is a runtime phenomenon that the compiler cannot predict, thus rendering the checkpoint placement incorrect.

One might ask, why not use a more complex compiler-directed cache-based system, like ReplayCache [252]. With ReplayCache, compiler transformations create idempotent regions in combination with a parallel cache writeback instruction, replaying cache modification after a failure. However, the ReplayCache-based method limits the use of legacy code and adds a significant amount of complexity in addition to a customized cache while adding many instructions to interact with the parallel writeback mechanism. One could propose using a dedicated hardware memory tracker, such as Clank [88], which could be deployed *in addition* to a data cache. This approach, however, increases the overall cost and latency of the system (as one ends up with two extra memory units—cache and memory tracker—instead of one).

We state that integration between intermittent computing systems using non-volatile main memory and a data cache should not merely utilize an existing cache architecture but rather tightly couple it with detecting WAR violations. Additionally, it should actively attempt to minimize the number of NVM accesses by considering the behavior of the execution flow of a program running on battery-free, intermittently powered systems. Looking at Table 6.1, which compares the most relevant systems for intermittent computation, *no existing solution addresses all* system requirements.

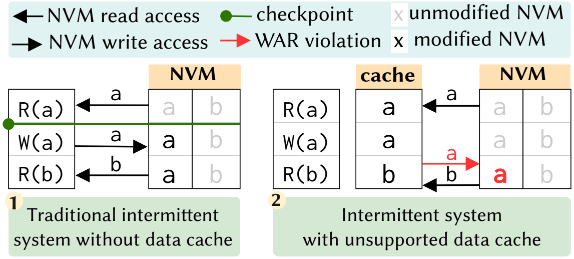


Figure 6.1: An example program performing memory accesses **R**(x) (a read operation at the memory location x) and **W**(x) (corresponding write operation) on two variables, for two systems. System ① is a cache-less (de facto standard, e.g. [235]), where read and write operations interact with NVM—note a compulsory checkpoint at WAR of variable 'a' inserted at compile time. System ② is based on a regular cache which cannot support WAR tracking by design, i.e., a simple direct mapped write-back [80, Appendix B] cache of two lines. In ②, checkpoints cannot be inserted at compile time because the compiler will not know when the eviction of cache-located variable 'a' back to NVM will take place.

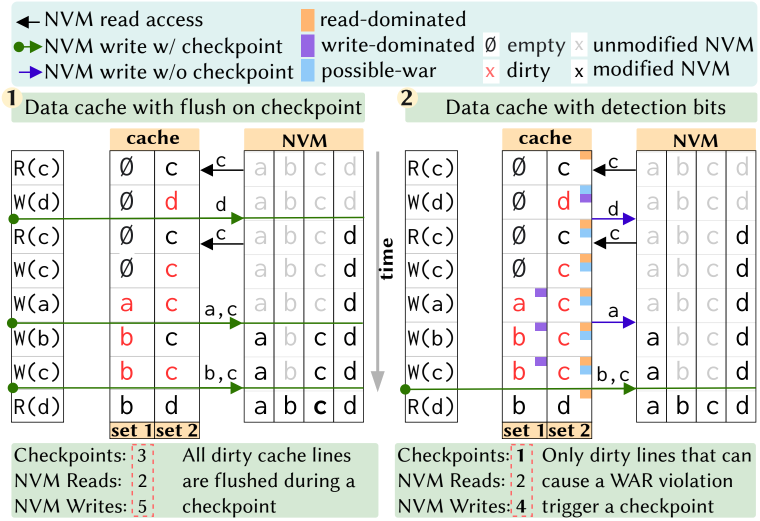


Figure 6.2: *Cache* and *NVM accesses* for an example program. A NVM access is a *read* from the NVM to the cache or a *cache eviction* of a dirty block from the cache to the NVM. The data cache here is *direct associative* with two sets. In some cases, this can cause a checkpoint signal to be raised, which is sent to the processor. In this case, all dirty cache blocks are evicted to the NVM, but kept in the cache. There are four arbitrary memory blocks *a, b, c, d* that are assigned to two cache sets as follows: a, b → set 1 and c, d → set 2.

6.3. Cache for Intermittent Systems

We propose a fundamentally different approach to overcome the challenge mentioned in Section 6.2.2. Our *data cache* will be tasked with (i) avoiding WAR (Section 6.3.1), (ii) optimizing WAR detection (Section 6.3.2), and (iii) reducing the number of NVM writes (Section 6.3.3).

6.3.1. Cache for Avoiding Write After Reads

First, we observe that a data cache *delays* the *write* to NVM, until a cache eviction forces a *write* to the underlying NVM. A WAR violation can only occur when a *write* is performed after a *read* at a NVM location. Therefore, the cache effectively determines when a WAR can occur by tracking the presence of such access patterns in a given cache line. In other words, we take advantage of the fact that *a WAR violation is only possible when a cache block is written back to the NVM*. We term this event, i.e. a cache line being written back to the NVM, as a *Cache Write Back (CWB)*.[1] Upon detecting a CWB, the cache generates a checkpoint signal and instructs the processor to create a checkpoint. During the checkpoint, the processor not only copies the registers to the NVM but also the volatile data cache (otherwise, the volatile data would be lost). To this end, all modified, i.e., dirty, memory blocks are copied (i.e., flushed) to the NVM during a checkpoint. By creating a checkpoint in this way, *we ensure that the system remains consistent*. An added advantage is clearing the cache of *all* dirty lines during the checkpoint, which decreases the possibility of a future WAR and thus reduces the number of created checkpoints and NVM accesses. We illustrate this process in Figure 6.2 ①.

6.3.2. Cache for Optimizing WAR Detection

As explained above, a CWB *can* lead to a WAR violation. However, some of these CWBs may not. This can be understood more formally as the memory being *read-dominated* or *write-dominated* as introduced in [88, Section 3.1.1]. In a sequence of memory instructions, if the first access to memory addresses is a write, then this location is write-dominated. Conversely, if the first access to a memory address (in a given sequence of instructions) is read, then this location is read-dominated. An idempotency violation can then be defined as *a write to a read-dominated memory location*. Any other form of access is safe and will not cause a WAR violation. Since any given memory sequence can be read-dominated or write-dominated, this condition bounds all possible idempotency violations. With this understanding, Clank [88] used dedicated hardware to track whether memory accesses are read- or write-dominated. In contrast, we use *the cache to perform the same tracking*, eliminating the need for an additional hardware component. Henceforth, we will denote write-dominated WARs as *safe* WARs, and read-dominated WARs as WAR violations.

To help understand the above, we redefine read-dominated and write-dominated sequences to track a cache line instead of a memory address. A cache line is read-dominated when the first access to the line is a *read* and write-dominated when the first access is a *write*. A CWB does not result in a WAR violation, and therefore does not require special action, if it comes from a write-dominated cache line. We term this write-back as a *safe write*. A CWB can only result in a violation if the associated cache way is read-dominated, which

[1] Similar to Intel x86 Cache Line Write Back instruction [97, Page 744].

we term as an *unsafe write*. We track these memory sequences to all cache lines during the program execution and create a checkpoint only if an *unsafe write* is encountered. The exact functionality of this tracking is discussed in Section 6.4.

The above process is shown in Figure 6.2 ②. Compared to Figure 6.2 ①, we notice a reduction in the number of checkpoints and NVM writes. An important thing to note is that since the cache stores data based on a hash of the memory address, the *distinction between safe-write and unsafe-write is also based on the hashed address*. This implies that the WAR detection is not exact, and although it can never contain false negatives, it does lead to few false positives. This is a trade-off in using the cache (and not a dedicated hardware module) as a memory tracker. However, we show later (Section 6.6) that this impact is negligible.

6.3.3. Reducing Unnecessary NVM Writes

Not all memory in the system is still in use (*live*) during a checkpoint. This is most notable when considering the stack memory of the program. Stack is allocated/deallocated constantly during execution when entering/leaving functions. However, stack memory that is no longer in use, i.e., has been deallocated, will never be read first during execution but is still marked as *dirty* in the data cache. This insight is based on the fact that deallocated stack is first written to when allocated. Hence the unallocated stack does not need to be written to NVM when creating a checkpoint, potentially reducing the number of WAR violations and reducing the number of writes to the NVM during a checkpoint.

6.4. System Architecture

We present NACHO: an architecture based on the data cache design presented in Section 6.3. NACHO ensures system incorruptibility during intermittent operation.

6.4.1. System Requirements

Along with supporting intermittent computing, NACHO has the following requirements.

Incorruptibility: NACHO ensures that the program's state is correct. As shown in Table 6.1, state of the art systems do not always guarantee incorruptibility. By using the data cache as a WAR detector, we guarantee memory consistency without energy prediction to create checkpoints.[2]

Cache Architecture Agnostic: Even though our system incorporates a custom data cache, NACHO is agnostic to the cache architecture (with any placement/replacement policies). Our additions are two extra bits that can be integrated seamlessly with most cache architectures.

[2] Energy detection consumes energy, estimating the threshold for the system is not accurate [203, Figure 1] and is difficult as system's complexity grows.

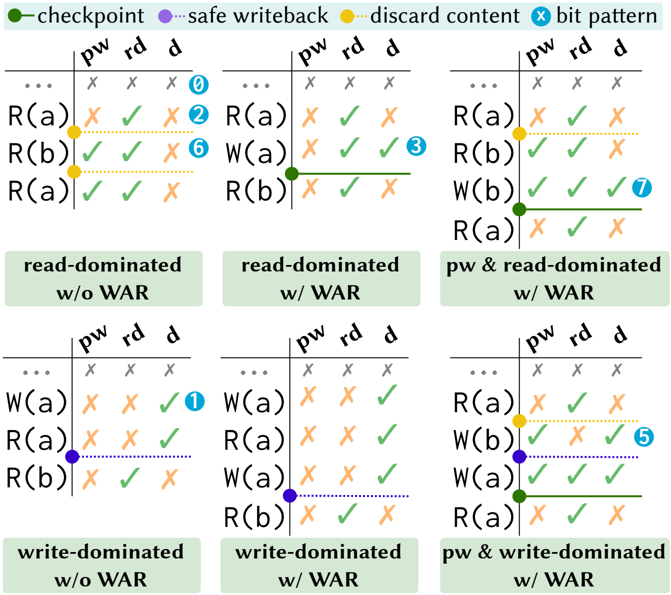


Figure 6.3: Six memory sequences and their corresponding bit patterns, including their decimal representation (blue circles). Every shown memory access maps to the one shown cache line. The data in the cache line is omitted. Only the cache line's three bits of interest are shown: `pw` (possible-war), `rd` (read-dominated), and `d` (dirty). The figure depicts all possible bit patterns. Note that configuration 4 (only the `pw` bit set) is invalid and can never occur.

6.4.2. Data Cache Controller

The WAR violation detection mechanism introduced in Section 6.3.2 glosses over real-world obstacles that prevent it from working in a fully-functioning data cache. In this section, we address the simplifications made and describe the exact workings of the optimized WAR detector.

Cache Line Bits

In Section 6.3 we introduced the concept of *read-dominated* and *write-dominated* bits in the cache line in order to detect WAR violations. We still track both these classifications but reduce the number of bits required to represent them. We express the *write-dominated* bit as a combination of the *dirty* bit (which already exists in a standard data cache) and the *read-dominated* bit because a line can only be *write-dominated* after it is written to, which sets the dirty bit. Additionally, we introduce the *possible-war* bit to reduce the conservativeness when detecting WAR violations by introducing history information into the cache line. In total, this results in just **two additional bits** compared to a standard data cache line, *possible-war* and *read-dominated*. Figure 6.3 shows all the possible bit patterns and the memory sequences required to reach them.

The Possible WAR Bit

The *possible-war* bit is set when the cache line is *read-dominated* and the data in a cache line is replaced. Multiple memory addresses are mapped to the same cache line when using a cache. While the *read-dominated* and *write-dominated* bits are a good start, they fall short when considering a memory location that is read into the cache, then evicted, and later written to (scenario "pw & write dominated w/ WAR" in Figure 6.3). In this scenario, the cache line would not be marked as *read-dominated* if the *possible-war* was not set (scenario "pw & read dominated w/ WAR"). Without the *possible-war* bit, all writes after a read to the data cache must be marked as *read-dominated*, leading to more checkpoints. However, this scenario could never be a WAR violation, as the incoming write **must** be to another memory address to evict the original entry. Thus the *possible-war* bit functions as a one-bit history, recording if there was a *read-dominated* cache entry in the block since the last checkpoint. But since the *possible-war* bit is set last, it will not be taken into consideration during the first transition from a *read-dominated* to a *write-dominated* cache line.

Possible WAR and Cache Associativity

When applying the cache bits to track WAR violations, we must consider the data cache associativity, i.e., the number of "ways" in the cache. Assuming an n-way cache, the cache controller can map a memory location to n different cache lines. At which cache line the memory location is placed depends on the cache replacement policy, e.g., *least recently used*. Now, consider a memory read to location m, marking the cache block as *read-dominated*. Next, the line containing m is evicted, removing it from the cache. Finally, memory location m is written to; however, this time, the data is written to another cache block for the same hash (which is possible when n is greater than one, i.e., the cache is not directly mapped). If this final cache line does not have its *possible-war* bit set, it will be marked *write-dominated*. Because the same memory location m was read before—but it was not detected because it occurred in a different cache line—this can lead to a WAR violation since the cache line is mismarked as *write-dominated*. To avoid marking cache lines incorrectly as *write-domianted*, we must slightly change our approach to set the *possible-war* bit. Instead of considering the *possible-war* bit of only the cache line to which data is moved, we must consider all the n cache lines in the set when deciding if it can be marked as *write-dominated*.

Stack Tracking

To both improve the execution time and lower the energy consumption, writing to the NVM should be avoided as much as possible. One situation where data in the cache is written back to NVM without it ever being read again is when a deallocated stack frame, i.e., a stack frame no longer in use because the function completed, is written back to NVM. To avoid writing this data back to NVM, we need to track the stack movement of the program. This can be straightforward, as the top of the stack is constantly being tracked by the Central Processing Unit (CPU)'s stack pointer sp. By also tracking $\texttt{sp}_{min}$, i.e., the lowest address the stack pointer reaches between two checkpoints (assuming the stack memory grows downwards in memory), we can discard all memory between sp and $\texttt{sp}_{min}$. By applying this technique, we avoid writing a dirty cache line to NVM during a checkpoint or cache eviction.

Data Cache Controller Algorithm

Algorithm 8 shows the manipulation of the two extra bits introduced by this work: *possible-war* (**pw**) and *read-dominated* (**rd**), in addition to the *dirty* bit (**d**), for each memory request. The other cache line-related bits (e.g., valid) and functionality (e.g., details regarding the ReplacementPolicy) are not shown in Algorithm 8 for the sake of brevity. We will now go through this algorithm, discussing each procedure in detail.

MemoryAccess: During a memory request (Line 1), the default data cache behavior is first to check if the request is a miss (Line 3). If the request results in a miss, an existing cache line must be evicted to make room for the new request. If the request is a hit, the memory location requested already resides in the cache. When a hit occurs, we introduce one special case. If this is the first hit for this cache line after a checkpoint was created, the cache line bits must be updated using the UpdateLine procedure (Line 20). We can identify this is the first hit by checking if all the considered flags are cleared (Line 5). If the cache line was already visited before, bit changes are needed, and the cache hit continues as usual.

CacheMiss: When a cache miss occurs (Line 8), we use a standard cache replacement policy, e.g., least recently used, to select the line that must be evicted to make room for the new request (Line 9). If the current line is not dirty, i.e., it was only read and never written to (Line 10), we can safely discard the data in the cache line and finish the request by updating the bits (Line 18). Later we update the cache line with data from the NVM (Line 7). If the cache line to be evicted is dirty, we can not simply write back the current content of the line to NVM, as this could cause a WAR violation. Instead, we first check if we can ignore the data because it is in a region of the stack that is no longer live (Line 11), in which case we can reset the cache line (Line 17). After all, the data does not need to be written back to NVM.

If the memory might still be in use, we check whether the memory accesses *could* have been read-dominated by checking the **rd** flag associated with the cache line (Line 12). If the cache line could be *read-dominated*, writing the memory back to NVM *could* cause a WAR violation, so we must create a checkpoint (Line 13). However, if we know for sure that the cache line is *write-dominated*, which **must** be the case if the cache line is *dirty* and **not** *read-dominated*, we can safely write (evict) the data directly to the NVM without creating a checkpoint (Line 15). Finally, we can continue the cache miss as usual by updating the cache line (Line 18) and returning the updated line to the MemoryAccess procedure.

UpdateLine: If the cache line is currently *read-dominated*, we store this fact, because this means that the *possible-war* bit must be set after updating the other bits to indicate that the *next* access could be *read-dominated* even if the request is a write (Line 33). If the request is a read (Line 22), only the *read-dominated* flag has to be set. However, if the request is instead a write (Line 24), we consider if any of the lines in the set have their *possible-war* bit set (Line 29), or the size if not equal to the size of the cache line (four bytes), in which case we must mark the current line as *read-dominated* (Line 32). Otherwise the *read-dominated* bit is cleared, marking it as *write-dominated*. The size requirement is introduced because a write request smaller than the cache line will first read from the NVM, which could lead to a WAR violation.

Checkpoint: The Checkpoint procedure (Line 35) performs a double-buffered write-back to NVM for all the dirty bits in the cache (abstracted for brevity as SafeEvict, Line 39)

Algorithm 8: Data cache controller

```
 1 Algorithm MemoryAccess(address, type, value, size) :
 2     line, miss ← CacheLine(address)
 3     if miss is true then
 4         line = CacheMiss(address, type, size)
 5     else if (line$_{pw}$ is false) and (line$_{rd}$ is false) and (line$_{dirty}$ is false) then
 6         UpdateLine(line, type, size)                          // 1st hit after checkpt
 7     UpdateData(line, value)                                    // Fill cache line with data

 8 Procedure CacheMiss(address, type, size) :
 9     line ← ReplacementPolicy(address)                          // Evicting line
10     if line$_{dirty}$ is true then
11         if InUnusedStack(address) is false then
12             if line$_{rd}$ is true then
13                 Checkpoint()
14             else
15                 Evict(line)                                    // Writeback without a checkpoint
16         else
17             ResetLine(line)                                    // No need for a writeback
18     UpdateLine(line, type, size)                               // Update new cache line
19     return line

20 Procedure UpdateLine(line, type, size) :
21     was-read-dominated ← line$_{rd}$
22     if type is Read then
23         line$_{rd}$ ← true                                     // Mark line as read-dominated
24     else if type is Write then
25         possible-WAR ← false
26         Set = GetSet(line)                                     // Get set associated with line
27         for line in Set do                                     // For each line in the set
28             possible-WAR ← (possible-WAR or line$_{pw}$)
29         if (possibe-WAR is false) and (size is 4) then
30             line$_{rd}$ ← false                                // Mark line as write-dominated
31         else
32             line$_{rd}$ ← true                                 // Mark line as read-dominated
33     if was-read-dominated then
34         line$_{pw}$ ← true                                     // Mark line as possible WAR

35 Procedure Checkpoint() :
36     for line in Cache do                                       // For each line in the cache
37         if line$_{dirty}$ then
38             if InUnusedStack(address) is false then
39                 SafeEvict(line)                                // Double buffered evict
40         ResetLine(line)                                        // Clear all the bits in the line
41     CheckpointRegisters()                                      // Checkpoint CPU registers
```

that are in use (Line 38). After the checkpoint is completed, the data still resides in the cache, but the WAR detection bits are cleared (Line 40) because the detection must be performed from one checkpoint to the next.

6.5. IMPLEMENTATION

We employ RISC-V [98] as the target Instruction Set Architecture (ISA) of NACHO, due to RISC-V's configurability and open-source nature. The evaluation of NACHO was performed using ICEmu [186], an emulator designed to evaluate intermittent computing systems, built around the QEMU-based [181] Unicorn CPU emulator [183]. For the purpose of NACHO evaluation we extended ICEmu to closely represent the *SiFive E21 standard core* processor [214]—by modeling its pipeline [214, Section 3.3]—as this is a basic 32-bit embedded processor. We note that NACHO code will be open source [174].

6.5.1. PROCESSOR EMULATION

We chose emulation instead of a hardware-based implementation of NACHO. The emulation enables us to evaluate the correctness of NACHO and all other systems used as NACHO's benchmark (introduced in Section 6.6.1), which will be *hard to accomplish with an MCU implementation*. This correctness evaluation is done as follows. As the first safety measure, the emulator duplicates the same access to a shadow memory for every memory access generated by the processor. This way, a correct memory access request handled by NACHO must return the same value as contained in the shadow memory. As the second safety measure, the emulator performs WAR detection to verify the absence of any WAR violation, as done in [148, Section 5.2] and [118, Section 5.1.1] by using read- and write-specific address lists and observing memory access patterns. Additionally, emulation allows us to collect detailed metrics without interfering with the program's execution.

6.5.2. MEMORY ACCESS COST MODEL

For the purpose of this evaluation, we assume a processor speed of 50 MHz. Additionally, we assume an access latency of a common onboard NVM of 125 ns [91, Table 1], [71, Section 2]. Furthermore, we assume that a data cache hit induces a two-cycle latency to the pipeline [214, Section 3.1] and an NVM access induces a six-cycle latency (rounded down). Note that all the above values are chosen conservativly[3], as a higher processor speed results in a larger difference between the data cache latency and the NVM latency, leading to an even better performance of NACHO than shown in this chapter (see Section 6.6).

6.5.3. CACHE CONTROLLER

We extended ICEmu with non-volatile main memory, and implemented NACHO's cache controller as an ICEmu memory subsystem. Within this subsystem, we implemented a data cache with a *least recently used* replacement policy and four bytes of data per cache line. Moreover, we enable the configuration of the data cache size and associativity. The additional bits introduced in Section 6.4 are implemented together with existing data cache bits to emulate a fully functional cache. On every memory access, the algorithm outlined in Algorithm 8 is executed, and the execution pipeline is updated using the cost model given in Section 6.5.2 to maintain an accurate cycle count. To enable the stack tracking (Section 6.4.2) the stack pointer is tracked during execution, storing the minimum address since the last checkpoint.

[3] Current MCUs targetting ultra-low-power applications often operate at speeds over 100 MHz, e.g. [18].

6.6. Evaluation

We compare NACHO against existing prior works and show that NACHO reduces the number of NVM and cache accesses, thus ensuring energy-efficient execution of intermittently powered applications. We further dissect NACHO's performance to show that NACHO's energy-efficient design choices incur very low computational overhead.

6.6.1. Evaluation Setup

We compare NACHO against reference systems using multiple benchmark applications and record various performance metrics to show the benefit of NACHO. We begin with the outline of our setup.

Benchmarks

We use CoreMark [59], an industry-grade benchmark for measuring embedded systems' CPU performance, to evaluate NACHO. Additionally, we use the CRC, SHA, and Dijkstra from the MiBench suite [77] to broaden the set of benchmarks. Finally, we use TinyAES [182] and picojpeg [68] to represent two real-life embedded applications. All benchmarks are compiled using version 9.1 of the `clang` [185] compiler using the `-O1` optimization level.

Systems

We compare NACHO against intermittent computing systems employing NVM as main memory. We ensure that our choice of systems covers both software and hardware support to solve the challenges arising from NVM main memory, as it would help establish the benefit of our system. These systems are as follows.

► **Clank [88]:** A memory-tracking hardware module that detects data inconsistencies during execution time. Our implementation is an *ideal* version of Clank, as it does not utilize any memory buffers that can fill up during the WAR detection [88, Section 3.1], nor does it count any memory access cost to these buffers. Since NACHO also performs WAR detection (but using just the data cache), we include Clank as a baseline to compare the performance metrics.

► **PROWL [91]:** A data cache implementation reducing NVM accesses, which avoids frequent checkpoints due to WARs by employing a custom cache replacement policy that delays the eviction of a dirty cache block. We include PROWL as another reference, in addition to Clank, as it relates most closely to NACHO, as they both introduce data cache modifications for intermittent systems.

► **Naive NACHO:** A basic version of NACHO, as described in Section 6.3.1, that does not have a WAR detector and no stack tracking support. The use of naive NACHO helps us dissect the performance gains achieved by each component of the ultimate NACHO.

► **Oracle NACHO:** An ideal version of NACHO that acts as its theoretical lower bound. The key difference between Oracle NACHO and NACHO lies in the detection of WARs based on the cache line addresses. While NACHO detects WAR using read/write-dominated cache lines, Oracle NACHO makes this detection using exact addresses, thus making it a perfect WAR violation detector. However, implementing such a system increases the hardware cost and complexity, thus making it impractical to implement.

All evaluations (for NACHO and for the above four systems, except for Clank, as this is a cache-less system) are performed for two different cache sizes (256 and 512 bytes [91]) of a 2-way set associative cache, to show the impact of the cache size on performance.

Metrics

We consider *five* evaluation metrics: (1) *Execution time:* the time required to complete a given workload along with performing the checkpoint; (2) *Checkpoints:* Number of times the device had to checkpoint its state due to a WAR violation; (3) *Number of NVM accesses:* the number of times a NVM memory read/write occurs during program execution; (4) *Number of cache accesses:* the number of times a cache read/write occurs during program execution; (5) *Interrmittent execution overhead:* the percentage computational overhead incurred when running on an intermittent energy.

6.6.2. Evaluation Results

We now proceed with the evaluation of NACHO.

Execution Time

Figure 6.4 shows the execution time of all systems for each of the considered benchmarks for two different cache configurations, normalized to a system with *fully volatile memory* of the respective benchmark. Note that the volatile memory system does not support intermittent computing and assumes the same memory technology for the main memory as the one used for the data cache. We can see that, on average, the normalized execution time for NACHO is 78% and 82% lower compared to Clank when using a 256 B and 512 B data cache, respectively. When compared against PROWL, NACHO's execution time is 54% and 43% lower for, respectively, 256 B and 512 B data cache (using the same cache size for both systems). On average, NACHO is within 4% and 1% of Oracle NACHO's execution time when using a 256 B and 512 B cache, respectively.

If we further dissect the numbers by removing the baseline program execution cost from all benchmarks, the overhead of all systems becomes even more apparent. Compared to PROWL, NACHO's overhead is, on average, 67% lower, with a maximum overhead reduction of 95% (CoreMark 512 B). Lower execution times for 512 B cache size are because of the cache's ability to retain more addresses, thus delaying the eviction, as explained in Section 6.6.2.

Number of Checkpoints

Figure 6.5 shows a significant decrease in the number of checkpoints of both PROWL and NACHO compared to Clank. It must be noted that a checkpoint in Clank only consists of the registers, whereas in both PROWL and NACHO, the cache has to be written back to the NVM in a double-buffered manner during a checkpoint, making it significantly more costly. In other words, even though NACHO had a larger update size at the time of checkpoint, NACHO is able to significantly reduce the need for checkpoints due to its efficient detection of idempotence violations. Additionally, we can also see a decrease in the number of checkpoints for a larger cache size for all the systems. This is primarily because of the ability of the cache to retain more data, thus reducing the need for eviction and, in turn, the checkpoint. We evaluate this effect further in Section 6.6.2.

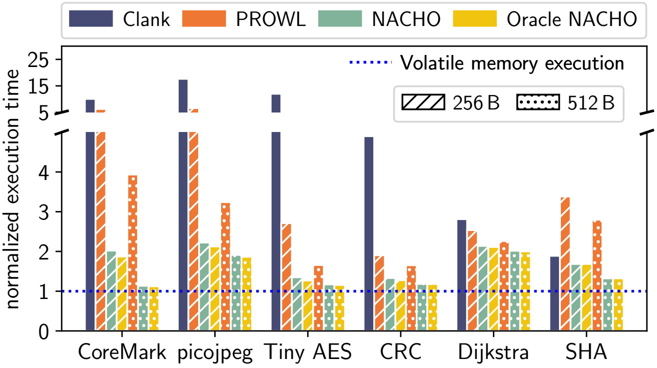


Figure 6.4: Execution time for all benchmarks for Clank [88], PROWL [91], NACHO, and Oracle NACHO. All results are normalized to the execution time of a system containing *only volatile memory*, i.e., a system without non-volatile main memory and intermittent computing support. Oracle NACHO is shown as the hypothetical lower bound that NACHO could reach if NACHO utilized perfect memory tracking. The cache configuration used is a 2-way set-associative for two cache sizes: 256 B and 512 B. Note that Clank is a cacheless system and is thus not affected by cache configuration.

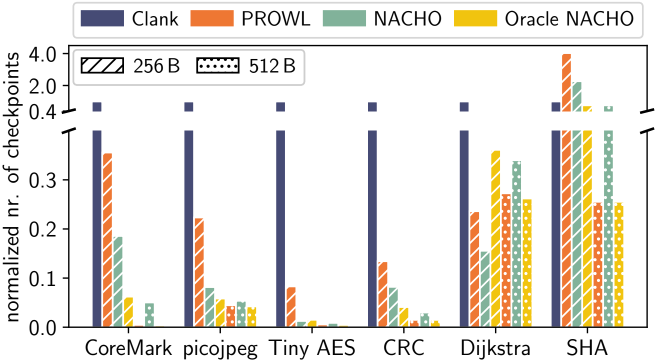


Figure 6.5: Number of checkpoints created during all benchmarks for Clank [88], PROWL [91], NACHO, and Oracle NACHO. All results are normalized to Clank. The system configurations are identical to those in Figure 6.4.

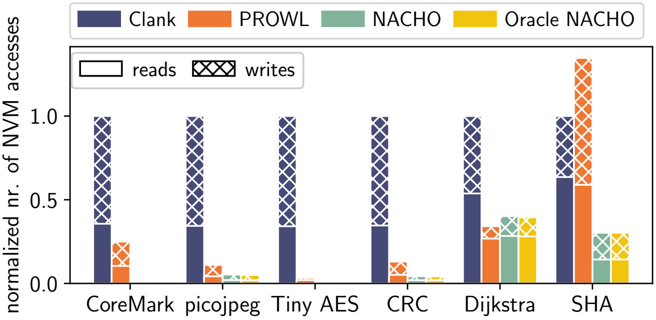


Figure 6.6: Number of non-volatile memory accesses—both *reads* and *writes* using a stacked bar plot—during all the benchmarks for Clank [88], PROWL [91], NACHO, and Oracle NACHO. All results are normalized to Clank. PROWL and NACHO are configured with a 512 B data cache.

Number of NVM Accesses

Figure 6.6 shows both the number of *read* and *write* accesses during the execution of each benchmark normalized to the numbers reported for Clank (which exclusively uses NVM). We can observe that NACHO significantly reduces the number of NVM accesses for almost all benchmarks, with **95% reduction** for CoreMark being the maximum and the trend holds for most benchmarks. On average, NACHO reduces the number of NVM accesses by over 85% and 55% compared to Clank and PROWL, respectively. While the trend holds for most benchmarks, Dijkstra is an extreme outlier as NACHO has 18% more NVM accesses, most likely caused by an unfortunate cache access pattern that causes many checkpoints in NACHO, therefore benefitting from the cache relocation strategy of PROWL. All of these numbers include the double-buffering cost associated with checkpoints for both PROWL and NACHO, demonstrating the effectiveness of adding a data cache.

Number of Cache Accesses

Figure 6.7 shows the number of data cache accesses required by each system. We observe that, on average, NACHO has 58% fewer cache accesses than PROWL. The higher number of cache accesses for PROWL stems from its custom cache line replacement policy. On every cache miss, PROWL performs cuckoo hashing in an attempt to avoid a memory eviction, leading to a much higher number of cache accesses (see the cuckoo bar stack in Figure 6.7) compared to NACHO.

Re-execution Overhead

An important metric to evaluate for any intermittently powered device is the cost of re-execution, i.e., the cost associated with a power failure. On every power failure, checkpointed system state must be restored whenever energy is available to resume application execution. Resuming the execution incurs an additional cost as the data cache loses its content after a power failure, resulting in cache misses. Furthermore, checkpoints are not created precisely before a power failure occurs, so there is a limited amount of *code re-execution* that adds to the computational overhead. Lastly, to guarantee forward progress, the system must introduce periodic checkpoints to guarantee that at least one

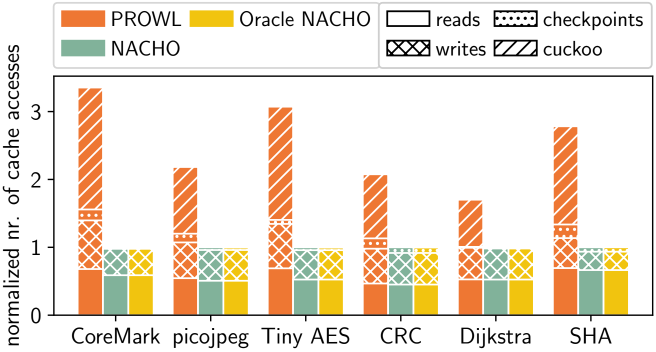


Figure 6.7: Number of data cache accesses—*reads*, *writes*, *checkpoints* (cache accesses during a checkpoint), and *cuckoo* (cache accesses by PROWL to move data within the cache) using a stacked bar plot—during all the benchmarks for PROWL [91], NACHO, and Oracle NACHO. All results are normalized to NACHO. PROWL and NACHO are configured with a 512 B data cache. Clank [88] is excluded from this plot as it does not have a data cache.

checkpoint is created during the on-duration.

Table 6.2 shows the re-execution cost with different power interruption intervals (on-duration), executing at a processor frequency of 50 MHz. The shorter the power interruption interval, the higher the cost of the overall cost of re-execution to complete the workload. For every on-duration n, we configure a periodic checkpoint to occur every $n/2$ ms to guarantee forward progress. We can observe that even with the shortest power interruption interval, the additional cost is less than 2% on average for all benchmarks, with CoreMark being the outlier with nearly 6%—which is still relatively low for such a worst-case operating scenario. With more reasonable interruption intervals, such as a power failure every 50 ms, we can see that the average additional cost is less than 0.2%. The low overhead can be attributed to the fact that five milliseconds is still a considerable amount of clock cycles and memory accesses, masking the cost of some additional periodic checkpoints, refilling the cache, and re-execution.

NACHO's Components Evaluation

Table 6.3 shows the percentage reduction achieved by WAR violation detection (PW) and stack-tracking approaches (ST) individually as well as the NACHO (N) overall. We see that for all benchmarks and considered metrics, the overall improvement of NACHO over Naive NACHO is significant, with an average overhead reduction of nearly 30% and a reduction in the number of NVM writes of almost 35%. It must be noted here that the reduction achieved by each component individually can not be summed to the overall reduction achieved, as both techniques (WAR violation detection and stack-tracking) can target similar memory access patterns.

Table 6.2: Re-execution overhead of NACHO, running at 50 MHz. The overhead consists of periodic checkpoints with half the period of the on-duration (to guarantee forward progress) and the re-execution cost of power failures.

On-duration	CoreMark	picojpeg	Tiny AES	CRC	Dijkstra	SHA
5 ms	5.75%	1.33%	0.69%	0.00%	0.72%	1.99%
10 ms	4.93%	0.67%	0.57%	0.00%	0.38%	1.41%
50 ms	0.20%	0.16%	0.35%	0.00%	0.07%	0.19%
100 ms	0.00%	0.07%	0.03%	0.00%	0.03%	0.19%

Table 6.3: Percental *reduction* of selected metrics compared to *Naive NACHO* for the two individual NACHO components, possible war (**PW**) and stack-tracking (**ST**), and finally the complete system—NACHO (**N**).

	CoreMark			picojpeg			Tiny AES			CRC			Dijkstra			SHA		
Metric	PW	ST	N	PW	ST	N	PW	ST	N	PW	ST	N	PW	ST	N	PW	ST	N
overhead	13%	7%	**21%**	10%	8%	**12%**	39%	40%	**46%**	16%	10%	**33%**	2%	0%	**3%**	38%	30%	**58%**
checkpoints	26%	4%	**31%**	8%	5%	**8%**	59%	48%	**59%**	33%	17%	**33%**	6%	0%	**6%**	40%	20%	**60%**
NVM reads	11%	6%	**18%**	9%	7%	**11%**	32%	32%	**37%**	14%	8%	**27%**	1%	0%	**1%**	33%	25%	**50%**
NVM writes	16%	9%	**25%**	11%	8%	**13%**	47%	47%	**54%**	18%	11%	**37%**	6%	0%	**6%**	41%	33%	**64%**

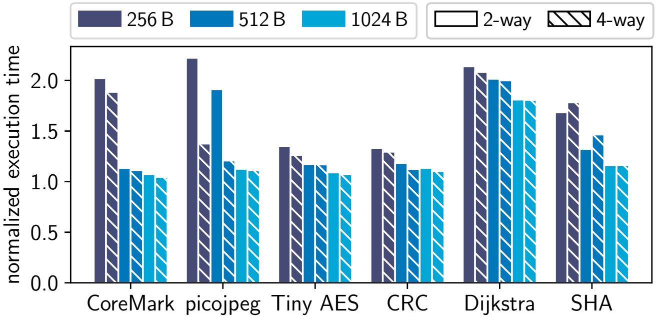


Figure 6.8: Cache configurations design space exploration of NACHO. All results are normalized to a 2-way 256 B cache.

Design Space Exploration

We now evaluate the effect of cache configurations on NACHO. Figure 6.8 shows NACHO's execution time in different cache configurations with varying sizes and associativity.

Cache Size: As we have shown, increasing the data cache size improves NACHO performance as a larger data cache can store more dirty blocks, effectively increasing the time between WARs and, therefore, checkpoints. Also, a larger data cache creates smaller mappings between cache lines and program memory, which gives higher accuracy to per-line WAR detection. However, as can be seen in Figure 6.8, at least for the considered benchmarks, the jump in performance between a data cache size of 512 B and 1024 B is not as significant as the jump from 256 B to 512 B.

Cache Associativity: A higher cache associativity implies that the cache can store more blocks for a given mapping before it evicts to make space. Similar to cache size, cache associativity also improves NACHO's performance as increasing the cache associativity decreases the probability of a cache collision. However, the increase in performance due to the increase in associativity is not as significant as it is with the change in size (it even reduced the performance in the case of SHA). NACHO must consider all cache blocks associated with a hash, reducing the benefits of higher associativity.

We conclude that, with NACHO, a 2-way set associative data cache is preferred over a more complicated 4-way cache implementation. The marginal increase in performance when utilizing a 4-way cache does not outweigh the additional complexity. Hence, during our evaluation, we configured NACHO with a 2-way set associative data cache[4].

6.7. Related Work

We now briefly review related works not listed in Section 6.1 or Chapter 1.

Energy-efficient Program Execution: Other works optimize the energy consumption of intermittently executing programs. Work of [13] proposes a dynamic voltage and frequency scaling approach to reduce the cost of program execution by allowing an intermittent system to dynamically regulate its operating voltage based on the changing

[4] Another reason is to aid comparisons against PROWL, which only provides hashing functions for a 2-way set associative data cache.

energy conditions, thus reducing the cost of program execution.

Static Volatile Memory Mapping: Another approach integrating volatile memory is a virtual memory manager that automatically maps data to either volatile or non-volatile memory during compilation [147]. However, only a limited number of accesses can be made volatile using this approach.

6.8. Discussion and Future Work

Integration with Other Systems: NACHO takes a different approach than PROWL [91], ReplayCache [252], and NvMR [34]. Even though all these systems utilize a data cache, their cache is not tightly integrated with WAR violation detection. PROWL and ReplayCache do not use WAR violation detection, and NvMR focuses on renaming NVM accesses to avoid WAR violations as much as possible and uses a detection mechanism similar to Clank [88]. These systems could **incorporate NACHO**, profiting from the techniques introduced in this chapter. This stems from the observation that NACHO focuses on an efficient way to *detect and avoid* WAR violations *using* the data cache instead of requiring a separate hardware module that introduces complexity and increases latency.

Chip Area and Energy Cost: Because NACHO is implemented as a memory subsystem in an emulator, it is impossible to gather information regarding the additional area or energy cost of the module. However, NACHO introduces just two additional bits per cache line in addition to some novel but small algorithmic changes to update these bits.

Energy Prediction: NACHO is incorruptible through double buffering. However, if the system can guarantee that enough energy is available to complete the cache writeback and register checkpoint, double buffering is not needed, halving the number of NVM writes during a checkpoint.

Peripherals: NACHO does not focus on supporting peripherals or other input/output operations needed to communicate with sensors and actuators. Rather, we consider this a separate topic addressed by other research [144, 250, 117] whose techniques could be integrated into NACHO.

6.9. Conclusions

We presented NACHO, a system where a data cache is coupled with the intermittent computing paradigm. NACHO, with the cache as a WAR violation detection entity, removes the need for additional memory tracking. NACHO accomplishes this by introducing two extra bits per cache line combined with a novel cache controller algorithm. Using these techniques, NACHO achieves significantly better performance than the state of the art solutions by reducing both the number of required checkpoints and accesses to slow, non-volatile memory, while offering support for different data cache architectures.

7
Conclusion

In this thesis, we examined and addressed multiple limitations brought forth by intermittent computing. In particular, we addressed ways to enable and speed up programmer-friendly intermittent computing. We did this both for traditional embedded systems and those with non-volatile main memory. In this chapter, we briefly conclude each of the introduced solutions, reflect on their differences, address the research questions, and finally look at the future of intermittent computing.

7.1. Contributions

The contributions presented in this thesis were split into two distinct parts: intermittently powered systems with *volatile* and ones with *non-volatile* main memory. Although the goal remained the same, improving intermittent computing, both memory architectures have unique challenges that differentiate them, leading to different approaches and solutions.

7.1.1. Contributions to Systems with Volatile Main Memory

In the first part of this thesis, we tackled challenges related to enabling intermittent computing on *conventional* embedded systems.

Battery-Free Prototyping

Before attempting to optimize intermittently powered systems, we first explored the challenges associated with converting a battery-operated embedded system to work intermittently and allow for rapid prototyping. To this end, we introduced BFree in Chapter 2. BFree is a complete intermittent-computing prototyping platform with out-of-the-box support for a plethora of sensors and actuators through existing libraries. This massive set of libraries is thanks to it being based upon CircuitPython [6], a Python interpreter targeting embedded systems. Because the interpreter introduces a layer of abstraction between the executing code and the hardware, it is ideally suited to support intermittent computing by modifying the interpreter.

Two major goals with BFree were supporting existing Python code and requiring as little programmer input as possible. BFree achieves this by starting with a very aggressive periodic checkpointing scheme. If the programmer chooses, they can dynamically tune the frequency and strategy of the checkpoints to their liking with a built-in API without risking corruption. However, the programmer can configure the system in a way that results in a lack of forward progress. To demonstrate the flexibility of BFree, we created two battery-free applications. One application measures the temperature and displays it on an E-Ink display. The other application measures the temperature and humidity, then applies some processing, and periodically sends the result over LoRa to a constantly-powered base station.

With BFree, we have shown that enabling programmer-friendly rapid prototyping for intermittently-powered embedded devices is possible, including the use of peripherals, all without requiring changes to existing embedded Python programs. Additionally, user studies have shown that quick prototyping using Python is preferable over task-based intermittent computing solutions and that many subjects would select BFree to create their intermittently-powered applications.

Differential Checkpoints

Although highly convenient for prototyping, BFree lacks the performance required to be suited for more demanding applications because it saves the complete memory during each checkpoint. In Chapter 3, we no longer targeted Python and focused on a more performance-oriented scenario. We introduced MPatch to address the high checkpointing cost by introducing a differential checkpoint mechanism that only stores the changes made to the memory since the previous checkpoint.

What makes MPatch unique is that it creates differential checkpoints while being incorruptible, because it never overwrites content from the previous checkpoint. MPatch achieves this by creating memory patches that contain the changes in memory when making a checkpoint. The restoration process can successively apply these patches to rebuild the memory state completely. We demonstrated MPatch as part of a battery-free GameBoy emulator named ENGAGE. Emulation is a computationally intensive process. Applying a checkpoint technique similar to the one introduced in BFree would create perceivable delays when playing games which would diminish the gaming experience. MPatch significantly reduces the time it takes to make a checkpoint. Because of this, MPatch also allows for more frequent checkpoints because the time it takes to create a checkpoint is directly related to the amount of memory changed between checkpoints.

With MPatch, we have shown that differential checkpoints are possible while remaining *incorruptible* by utilizing memory patches that do not overwrite the content of the previous checkpoint.

7.1.2. Contributions to Systems with Non-Volatile Main Memory

The second part of the thesis addressed a less common embedded system architecture, namely one with non-volatile main memory. In such architectures, the non-volatile memory can be accessed like traditional SRAM in a microcontroller, allowing it to be used as the system's main memory. Using non-volatile main memory reduces the size of checkpoints by not including the main memory. However, not including the main memory in the checkpoint also introduces a new problem, as introduced in Chapter 1, where registers are restored from a checkpoint, but the memory is not, which can lead to desynchronization of the memory and register state, potentially causing corruption during re-execution.

Reducing Checkpoints Using a Segmented Stack and Undo-Logging

A big problem with existing automated checkpointing solutions was the number of checkpoints that must be created during execution to avoid WAR violations (Section 1.2.2). Each WAR violation must be broken using a checkpoint (i.e., a checkpoint must be placed between the Read and Write of the WAR) to avoid corrupting the memory during re-execution. All these checkpoints introduce significant overhead; therefore, other methods have been introduced that instead rely on logging the changed memory. This way, memory modifications can be undone as part of the restoration process, removing the need to break all WAR violations, reducing the number of required checkpoints. However, this introduces another source of overhead because logging memory access in software is costly. Additionally, placing checkpoints to break the remaining WAR violations introduces additional complexity for some architectures. In register–memory architectures, some instructions can directly read from, modify, and write to memory. These instructions can introduce implicit WAR violations, where the same instruction causes both the read and the write in a WAR. These WAR violations must be artificially broken up to resemble a load-store architecture scheme, adding even more overhead.

Despite its limitation, we chose the less well-suited MSP40FR microcontroller that has instructions that can cause implicit WAR violations [1] because, at the time of publishing, it was the only commercially available microcontroller with byte-addressable memory—in the case of the MSP40FR in the form of FRAM. We introduced TICS to overcome these limitations, targeting the MSP40FR microcontroller series. TICS takes a hybrid approach, where part of the memory accesses are logged and undone during the restoration process, and part of the memory is included in the checkpoint. TICS achieves this by introducing a segmented stack, where the active stack segment is checkpointed together with the registers and can therefore be freely modified without risking memory corruption. Memory accesses outside the active stack are logged and undone during restoration. Where the active stack is in memory depends on the stack pointer and is assumed to be the most frequently accessed region of memory due to the principle of locality. Because the active stack is included in a checkpoint, the time it takes to create a checkpoint increases. In return, TICS avoids many costly log operations and checkpoints to break WAR violations.

With TICS, we have shown that it is possible to enable intermittent computing on register-memory architectures without requiring instructions to be split to avoid implicit WARs. We demonstrated that logging-based intermittent computing utilizing recursion is possible while achieving similar performance compared to the related work. We have shown that introducing a segmented stack can reduce the overhead introduced by undo logging while allowing the execution of unmodified C-based programs.

Reducing Checkpoints using Instruction Rescheduling

Although TICS avoids checkpoints, it does so by introducing a segmented stack, and an undo log, both of which need their size to be configured by the programmer. Additionally, the logging adds uncertainty regarding execution time to the system, which can be problematic for some applications (e.g., if they rely on a region of code executing without a checkpoint or in a predictable amount of time). Finally, because of the runtime aspect, the performance of TICS heavily depends on the memory access characteristics of the application. Suppose we target a load-store architecture such as ARM instead of the MSP40FR. ARM-based processors are a lot more common in the IoT world [27], and ARM processors [17] offer significant benefits in terms of power consumption compared to the relatively outdated MSP430FR [226]. In that case, we can apply the alternative method to reduce the checkpointing overhead—placing a checkpoint to break all WAR violations. However, this approach, as noted, comes with many mandatory checkpoints. We developed WARio to reduce the number of these mandatory checkpoints.

WARio uses compiler analysis and transformations to reduce the number of mandatory checkpoints. WARio reduces checkpoints while preserving predictability and simplicity, as there is no need for a logging runtime. WARio achieves checkpoint reduction by introducing multiple optimizations to reschedule memory operations to break as many WAR violations as possible with a single checkpoint. The most influential optimization is the custom loop unrolling that allows WAR violations from multiple loop iterations to be rescheduled so they can be broken using a single checkpoint. Additionally, WARio employs more sophisticated alias analysis techniques to reduce the number of required checkpoints even further. We have also shown that enabling hitting-set checkpoint placement in the back end, in addition to the front end, can further reduce the number of checkpoints. Finally, we have shown that we can further reduce the number of checkpoints required when performing stack modifications by temporarily disabling interrupts.

Avoiding WAR Violations Using a Data Cache

Until now, all the methods introduced in this thesis support intermittent computing primarily through software. Doing so allows us to target existing microcontroller architectures without needing specialized microcontroller hardware architectures, which could limit the adoption of intermittent computing. However, one may argue that the intermittent computing consistency problem is better suited to be solved in hardware instead.

We introduced NACHO in Chapter 6 to explore hardware-based support for intermittent computing. NACHO combines concepts from chapters 4 and 5 but realizes some of their concepts in hardware while also addressing another downside of intermittent computing, costly non-volatile memory accesses. Even though non-volatile memory architectures like MRAM, FRAM, and ReRAM are functionally very close to SRAM, the energy consumption when accessing memory is currently still higher, and the access speeds are slower. NACHO addresses this problem by introducing a volatile data cache and addresses the consistency problem by making novel modifications to both the cache entries (lines) and the cache controller algorithm to detect WAR violations. When using NACHO, the code can freely modify data in the volatile memory without risking corruption because the volatile cache is part of the checkpoint, similar to the active stack in TICS. When memory needs to be evicted from the cache, NACHO uses custom cache line bits to determine if this could cause a WAR, in which case NACHO creates a checkpoint first, breaking the WAR violations as done in WARio.

With NACHO, we have shown that we can merge the detection and avoidance of WAR violations with the functionality of a data cache. We have demonstrated that avoiding WAR violations using a data cache is possible by adding two additional bits per cache line and a novel cache controller algorithm that initiates checkpoints when a potential WAR violation is detected. By utilizing a volatile data cache in combination with non-volatile memory, we have demonstrated a significant increase in performance and a reduction in the number of checkpoints forced by WARs.

7.2. LOOKING BACK

A significant benefit of using software-based approaches to support intermittent computing is that no dedicated intermittent-computing-specific hardware is required. Even if the method requires non-volatile main memory such as FRAM, MRAM or ReRAM, these memories already replace flash in some microcontrollers due to their lower power consumption. Therefore, these non-volatile memories do not need to be included for the sole purpose of enabling intermittent computing. However, even though our software-based approaches reduce the checkpointing overhead, a hardware-based solution—utilizing an architecture especially designed to support intermittent computing—can reduce it even further.

By introducing a volatile data cache and monitoring memory accesses in hardware, we combine the best of the previous two approaches in Chapter 6. Memory residing in the volatile data cache can be modified freely without requiring checkpoints, just like in the active stack segment in Chapter 4. And just like discussed in Chapter 5, a checkpoint is inserted whenever a modification to the non-volatile main memory can cause a WAR violation, although this time dynamically instead of at compile time. NACHO is the most performant system introduced in this thesis because it uses a modified architecture. However, as mentioned, the need for a dedicated architecture might result in limited adoption. Typical microcontrollers are applicable in many different situations and use cases, which is not true for NACHO or any other intermittent-computing specific processor. For such solutions to take off, the number of intermittently powered applications deployed in practice must increase dramatically. However, this dramatic increase may happen shortly, as intermittent computing is still a relatively new concept and has only recently started to mature. Until the market is ready, software-based approaches like the ones demonstrated in this thesis will show the importance and value of intermittent computing without needing specialized hardware.

7.3. FUTURE WORK AND CHALLENGES

Research into intermittent computing has become more mature over the last few years, but there are still more improvements and innovations to be made in the domain. Much research has been dedicated to enabling intermittent computing with as little effort as possible for application developers to “do their thing.” But even with all that effort, we are not quite there yet.

7.3.1. Interacting With the Outside World

Even though computation on intermittently powered platforms is now possible using checkpoint-based systems like the ones presented in this thesis or task-based methods, communicating with the outside world remains difficult and understudied. Interacting with the outside world is complicated because once an action is performed, it can not be undone. For most existing systems, this is abnormal behavior, so there are often no mechanisms to handle these situations gracefully. Moreover, it is sometimes unclear how to handle such a scenario. In the case of radio transmissions, one might want to retransmit a whole packet or perhaps restart the handshaking procedure altogether. For different actuators or situations, there might be different desired responses. Because of all these different approaches, it is nearly impossible for an intermittently powered system to independently deal with these situations. Therefore, to successfully interact with the outside world, the constantly powered systems must be made aware of the limitations associated with intermittently powered systems. Hence, more research is needed into interaction with constantly-powered systems and the outside world.

7.3.2. Hardware Support

Most of this thesis introduces software-based solutions, but even then, Part Two of this thesis relies on having non-volatile memory. Although byte-addressable non-volatile memory like MRAM and FRAM is already present in some microcontrollers, it is far from the level of adoption needed to allow intermittent computing to catch on as a mainstream concept. Additionally, systems with these non-volatile technologies do not expect them to be utilized as the system's main memory but rather as a location to store the program and perhaps some logging data instead of using flash. Microcontroller designers must be aware of intermittent computing to adopt the software-based approaches introduced in this thesis or any other work based on non-volatile main memory. More importantly, microcontroller designers must adopt intermittent computing as a possible use case for their devices by including byte-addressable non-volatile memory on their chips.

In this thesis, we also explored a hardware-based solution to support intermittent computing in Chapter 6. Hardware support would be ideal, as it removes most of the checkpointing overhead and even reduces the additional cost of using non-volatile memory. However, full hardware support is challenging to realize. Intermittent computing is still a niche, with few existing applications operating in the real world. Therefore, convincing a microcontroller manufacturer to produce chips solely targetting intermittent computing will likely not make financial sense soon and will only make sense when there already is widespread adoption. On the other hand, hardware adoption can kickstart the adoption of intermittent computing if a company is willing to take the risk to produce a dedicated architecture. However, there is still a long way to go to realize such an architecture on actual silicon. To this end, we targeted our hardware-based solution toward the RISC-V architecture, as it is open source and provides the lowest barrier of entry for a company to produce a microcontroller that supports intermittent computing.

7.3.3. Applications and Adoption

As already alluded to, intermittently powered applications are still in their infancy. There have been more and more applications in research, but real-world applications that are truly intermittently operating are still few and far between. Part of the reluctance to adopt intermittent computing comes from the fact that we are used to constantly powered systems that can sense or actuate at any time. For a company to let potential users know that their monitoring system might miss readings in certain scenarios will lead them to stray away from battery-free systems and opt to use traditional battery-based approaches. Even though the scenarios where no energy can be harvested are rare, and missing some sensor reading for a short period often does not affect the overall result, the fact that these (brief) periods of downtime exist introduces uncertainty that many people and businesses are unwilling to accept–even if using intermittently powered systems keep batteries out of landfills and reduces the workload of employees who have to replace batteries. To this end, research must be targeted toward distributed intermittent computing to aid the adoption of intermittently powered applications to reduce the effect of downtime. In addition, more real-world experiments must be performed to demonstrate that this is either a non-issue or novel techniques that mitigate the effect of intermittent operation must be researched and introduced.

Bibliography

[1] Extended MSP430X 16-bit RISC CPU (CPUX) with 1 MB memory access. https://www.ti.com/lit/ug/slau391f/slau391f.pdf, March 2018. Last accessed: Jul. 21, 2021.

[2] Engage open source repository. https://github.com/tudssl/engage, July 2020. Last accessed: Jul. 22, 2020.

[3] Adafruit. Si7021 temperature and humidity sensor breakout board. https://www.adafruit.com/product/3251, September 2016. Last accessed: Oct. 27, 2020.

[4] Adafruit. PCF8523 real time clock assembled breakout board. https://www.adafruit.com/product/3295, August 2017. Last accessed: Oct. 27, 2020.

[5] Adafruit. Adafruit Metro M0 express - designed for circuitpython - atsamd21g18. https://www.adafruit.com/product/3505, April 2018. Last accessed: Sep. 14, 2019.

[6] Adafruit. Welcome to CircuitPython! https://learn.adafruit.com/welcome-to-circuitpython, September 2019. Last accessed: Oct. 27, 2020.

[7] Adafruit. RFM95W LoRa radio transceiver breakout board. https://www.adafruit.com/product/3072, January 2020. Last accessed: Oct. 27, 2020.

[8] Kofi Sarpong Adu-Manu, Nadir Adam, Cristiano Tapparello, Hoda Ayatollahi, and Wendi Heinzelman. Energy-harvesting wireless sensor networks (EH-WSNs): A review. *ACM Transactions on Sensor Networks*, 14(2):10:1–10:50, July 2018.

[9] Mikhail Afanasov, Naveed Anwar Bhatti, Dennis Campagna, Giacomo Caslini, Fabio Massimo Centonze, Koustabh Dolui, Andrea Maioli, Erica Barone, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Battery-less zero-maintenance embedded sensing at the mithræum of circus maximus. In *Proc. SenSys*, pages 368–381, Virtual Event, 2020. ACM. https://doi.org/10.1145/3384419.3430722.

[10] Sayed Saad Afzal, Waleed Akbar, Osvy Rodriguez, Mario Doumet, Unsoo Ha, Reza Ghaffarivardavagh, and Fadel Adib. Battery-free wireless imaging of underwater environments. volume 13, pages 1–9. Nature Publishing Group, 2022.

[11] Saad Ahmed, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, Naveed Anwar Bhatti, and Luca Mottola. Poster abstract: Towards smaller checkpoints for better intermittent computing. In *Proc. IPSN*, pages 132–133, Porto, Portugal, 2018. ACM/IEEE.

[12] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Effient intermittent computing with differential checkpointing. In *Proc. LCTES*, pages 70–81, Phoenix, AZ, USA, June23 2019. ACM.

[13] Saad Ahmed, Qurat ul Ain, Junaid Haroon Siddiqui, Luca Mottola, and Muhammad Hamad Alizai. Intermittent computing with dynamic voltage and frequency scaling. In *Proc. EWSN*, pages 97–107, Lyon, France, 2020. ACM. https://doi.org/10.5555/3400306.3400319.

[14] Ambiq Micro. APOLLO ultra-low power microcontrollers and SoC solutions. https://ambiqmicro.com/mcu, 2018. Last accessed: Oct. 27, 2020.

[15] Ambiq Micro Inc. Cortex-M4 revision r0p0 technical reference manual. https://documentation-service.arm.com/static/5f19da2a20b7cf4bc524d99a, March 2010. Last accessed: Nov. 10, 2021.

[16] Ambiq Micro Inc. Apollo3 Blue ultra-low power microcontroller. https://ambiqmicro.com/static/mcu/files/Apollo3_Blue_MCU_Data_Sheet_v0_11_0.pdf, 2018. Last accessed: Apr. 25, 2020.

[17] Ambiq Micro Inc. Apollo4 Blue ultra-low power microcontroller. https://ambiq.com/apollo4-blue/, November 2021. Last accessed: Nov. 10, 2021.

[18] Ambiq Micro Inc. Apollo4 ultra-low power microcontroller. https://ambiq.com/apollo4/, October 2022. Last accessed: Oct. 17, 2022.

[19] Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Greg Chan, Simone Campanoni, and David I. August. SCAF: A speculation-aware collaborative dependence analysis framework. In *Proc. PLDI*, pages 638–654, London, UK, 2020. ACM. https://doi.org/10.1145/3385412.3386028.

[20] Arduino. Arduino GitHub Repository. https://github.com/arduino/Arduino, September 2019. Last accessed: Oct. 27, 2020.

[21] Arduino. Arduino Uno Rev3. https://store.arduino.cc/arduino-uno-rev3, March 2019. Last accessed: Oct. 27, 2020.

[22] Rauf Arif. With an economic potential of $11 trillion, Internet Of Things is here to revolutionize global economy. Forbes, https://www.forbes.com/sites/raufarif/2021/06/05/with-an-economic-potential-of-11-trillion-internet-of-things-is-here-to-revolutionize-global-economy, June 2021. Last accessed: Oct. 14, 2022.

[23] Arm Limited. ARM1156T2-S revision r0p4 technical reference manual. https://documentation-service.arm.com/static/5e8e116188295d1e18d34a29, July 2007. Last accessed: Nov. 5, 2021.

[24] ARM Limited. Cortex-m0. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0, September 2019. Last accessed: Oct. 27, 2020.

[25] ARM Limited. Mbed os 5 website. https://www.mbed.com/en, 2019. Last accessed: Oct. 27, 2020.

[26] Arm Limited. Arm cortex-M series processors. https://developer.arm.com/ip-products/processors/cortex-m, 2021. Last accessed: Nov. 4, 2021.

[27] Arm Limited. The Arm ecosystem ships a record 6.7 billion Arm-based chips in a single quarter. https://www.arm.com/company/news/2021/02/arm-ecosystem-ships-record-6-billion-arm-based-chips-in-a-single-quarter, February 2021. Last accessed: Nov. 4, 2021.

[28] Nivedita Arora, Steven L. Zhang, Fereshteh Shahmiri, Diego Osorio, Yicheng Wang, Mohit Gupta, Zhengjun Wang, Thad Eugene Starner, Zhonglin Wang, and Gregory D. Abowd. SATURN: A thin and flexible self-powered microphone leveraging triboelectric nanogenerator. *ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2(2):60:1–60:28, June 2018.

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

[30] Domenico Balsamo, Alex S. Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M. Al-Hashimi, Geoff V. Merrett, and Luca Benini. Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 35(12):1968–1980, 2016.

[31] Domenico Balsamo, Alex S. Weddell, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Syst. Lett.*, 7(1):15–18, March 2015.

[32] Chris Baraniuk. Why is there a chip shortage? https://www.bbc.com/news/business-58230388, August 2021. Last accessed: Mar. 18, 2022.

[33] Uwe Becker. Gameboy-emulator per STM32F746 (in german). https://mikrocontroller.bplaced.net/wordpress/?page_id=1290, November 2017. Last accessed: May 5, 2020.

[34] Abhishek Bhattacharyya, Abhijith Somashekhar, and Joshua San Miguel. NvMR: Non-volatile memory renaming for intermittent computing. In *Proc. ISCA*, pages 1–13, New York, NY, USA, 2022. ACM. https://dl.acm.org/doi/10.1145/3470496.3527413.

[35] Naveed Anwar Bhatti and Luca Mottola. HarvOS: Efficient code instrumentation for transiently-powered embedded sensing. In *Proc. IPSN*, pages 209–219, Pittsburgh, PA, USA, April 18–21 2017. ACM/IEEE. https://doi.org/10.1145/3055031.3055082.

[36] Eli Blevis. Sustainable interaction design: Invention & disposal, renewal & reuse. In *Proc. CHI*, pages 503–512, San Jose, CA, USA, 2007. ACM.

[37] Adriano Branco, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Intermittent asynchronous peripheral operations. In *Proc. SenSys*, pages 55–67, New York City, NY, USA, 2019. ACM.

[38] Lars Büthe, Michael Hardegger, Patrick Brulisauer, and Gerhard Tröster. RFID-Die: Battery-free orientation sensing using an array of passive tilt switches. In *Proc. UbiComp Adjunct*, pages 215–218, Seattle, WA, USA, September 13 - 17 2014. ACM.

[39] Pierre Carbonnelle. PYPL: PopularitY of programming language. http://pypl.github.io, August 2020. Last accessed: Oct. 27, 2020.

[40] Stephen Cass. The top programming languages 2020. https://spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020, July 2020. Last accessed: Oct. 27, 2020.

[41] Arunkumar Chandrasekhar, Gaurav Khandelwa, Nagamalleswara Rao Alluri, Venkateswaran Vivekananthan, and Sang-Jae Kim. Battery-free electronic smart toys: A step toward the commercialization of sustainable triboelectric nanogenerators. *Sustainable Chemistry and Engineering*, 6(5):6110–6116, April 2018.

[42] Arunkumar Chandrasekhar, Gaurav Khandelwal, Nagamalleswara Rao Alluri, Venkateswaran Vivekananthan, and Sang-Jae Kim. Sustainable biomechanical energy scavenger toward self-reliant kids' interactive battery-free smart puzzle. *Sustainable Chemistry and Engineering*, 5(8):7310–7316, June 2017.

[43] Tzuwen Chang, Neng-Hao Yu, Sung-Sheng Tsai, Mike Y. Chen, and Yi Ping Hung. Clip-on gadgets: Expandable tactile controls for multi-touch devices. In *Proc. MobileHCI*, pages 163–166, San Francisco, CA, USA, 2012. ACM.

[44] Jongouk Choi, Hyunwoo Joe, Yongjoo Kim, and Changhee Jung. Achieving stagnation-free intermittent computation with boundary-free adaptive execution. In *Proc. RTAS*, pages 331–344, Montréal, QC, Canada, April 16–18 2019. IEEE.

[45] Clare Church and Laurin Wuennenberg. Sustainability and second life: The case for cobalt and lithium recycling, March 2019.

[46] Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson Sample. An energy-interference-free hardware/software debugger for intermittent energy-harvesting systems. In *Proc. ASPLOS*, pages 577–589, Atlanta, GA, USA, 2016. ACM.

[47] Alexei Colin and Brandon Lucia. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proc. OOPSLA*, pages 514–530, Amsterdam, The Netherlands, 2016. ACM.

[48] Alexei Colin and Brandon Lucia. Termination checking and task decomposition for task-based intermittent programs. In *Proc. Conference on Compiler Construction*, pages 116–127, Vienna, Austria, 2018. ACM.

[49] Alexei Colin, Emily Ruppel, and Brandon Lucia. A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices. In *Proc. ASPLOS*, pages 767–781, Williamsburg, VA, USA, 2018. ACM.

[50] Rodrigo Copetti. Architecture of consoles: Gameboy. https://copetti.org/projects/consoles/game-boy, February 2019. Last accessed: May 3, 2020.

[51] Bandai Corporation. LCD Solarpower handheld electronic games series. https://en.wikipedia.org/wiki/Bandai_LCD_Solarpower, 1982. Last accessed: Sep. 22, 2020.

[52] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent code generation: Implementation, analysis, and evaluation. In *Proc. CGO*, Shenzhen, China, February 23–27 2013. ACM/IEEE.

[53] Marc de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static snalysis and compiler design for idempotent processing. In *in Proc. PLDI*, pages 475–486, Beijing, China, 2012. ACM. https://doi.org/10.1145/2254064.2254120.

[54] Jasper de Winkel, Carlo Delle Donne, Kasım Sinan Yıldırım, Przemysław Pawełczak, and Josiah Hester. Reliable timekeeping for intermittent computing. In *Proc. ASPLOS*, pages 53—-67, Lausanne, Switzerland, 2020. ACM.

[55] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. Battery-free game boy. *ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(3):111:1–111:34, September 2020. https://doi.org/10.1145/3411839.

[56] Christine Dierk, Molly Jane Pearce Nicholas, and Eric Paulos. Alterwear: Battery-free wearable displays for opportunistic interactions. In *Proc. CHI*, pages 210:1–210:13, Montréal, QC, Canada, 2018. ACM.

[57] Pedro C. Diniz and Martin C. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *J. Parallel Distrib. Comput.*, 49(2):218–244, March 1998. https://doi.org/10.1006/jpdc.1998.1441.

[58] Conrad Donovan, Alim Dewan, Deukhyoun Heo, and Haluk Beyenal. Batteryless, wireless sensor powered by a sediment microbial fuel cell. *Environmental Science & Technology*, 42(22):8591–8596, October 2008.

[59] Embedded Microprocessor Benchmark Consortium. CoreMark benchmark. https://github.com/eembc/coremark/releases/tag/v1.01, May 2018. Last accessed: Jun. 29, 2021.

[60] EnOcean. Enocean wall mounted occupancy sensor. https://www.enocean.com, April 2018. Last accessed: Oct. 27, 2020.

[61] Xiaoran Fan, Han Ding, Sugang Li, Michael Sanzari, Yanyong Zhang, Wade Trappe, Zhu Han, and Richard E. Howard. Energy-ball: Wireless power transfer for batteryless internet of things through distributed beamforming. *ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2(2):65:1–65:22, June 2018.

[62] Laura Marie Feeney, Christian Rohner, Per Gunningberg, Anders Lindgren, and Lars Andersson. How do the dynamics of battery discharge affect sensor lifetime? In *Proc. Annual Conference on Wireless On-demand Network Systems and Services*, pages 49–56, Obergurgl, Austria, April 2014. IEEE. https://doi.org/10.1109/WONS.2014.6814721.

[63] Micro:bit Educational Foundation. BBC micro:bit. https://www.microbit.org, February 2016. Last accessed: Oct. 27, 2020.

[64] Jon Froehlich, Leah Findlater, Marilyn Ostergren, Solai Ramanathan, Josh Peterson, Inness Wragg, Eric Larson, Fabia Fu, Mazhengmin Bai, Shwetak N. Patel1, and James A. Landay. The design and evaluation of prototype eco-feedback displays for fixture-level water usage data. In *Proc. CHI*, pages 2367–2376, Austin, TX, USA, 2012. ACM.

[65] Fujitsu Semiconductor Ltd. MB85RS4MT 512 KB SPI FRAM. https://www.fujitsu.com/uk/Images/MB85RS4MT.pdf, 2018. Last accessed: Apr. 25, 2020.

[66] Karthik Ganesan, Joshua San Miguel, and Natalie Enright Jerger. The what's next intermittent computing architecture. In *Proc. HPCA*, pages 211–223, Washington, DC, USA, February 16–20 2019. IEEE.

[67] Rich Geldreich. picojpeg: Plain C JPEG decompressor. https://github.com/richgel999/picojpeg, March 2020. Last accessed: Nov. 5, 2021.

[68] Rich Geldreich. Picojpeg. https://github.com/richgel999/picojpeg, July 2022. Last accessed: Oct. 17, 2022.

[69] Damien P. George. Micropython home page. https://micropython.org, September 2019. Last accessed: Oct. 27, 2020.

[70] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proc. ASPLOS*, pages 199–213, Providence, RI, USA, April 13–17 2019. ACM.

[71] William Goh, Andreas Dannenberg, and Johnson He. Texas Instruments MSP430 FRAM technology – how to and best practices. https://www.ti.com/lit/an/slaa628b/slaa628b.pdf, August 2021. Last accessed: Oct. 17, 2022.

[72] Rabeeh Golmohammadzadeh, Fariborz Faraji, Brian Jong, Cristina Pozo-Gonzalo, and Parama Chakraborty Banerjee. Current challenges and future opportunities toward recycling of spent lithium-ion batteries. *Renewable Sustainable Energy Rev.*, 159:112202:1–112202:24, May 2022. https://doi.org/10.1016/j.rser.2022.112202.

[73] Tobias Grosse-Puppendahl, Steve Hodges, Nicholas Chen, John Helmes, Stuart Taylor, James Scott, Josh Fromm, and David Sweeney. Exploring the design space for energy-harvesting situated displays. In *Proc. UIST*, pages 41–48, Tokyo, Japan, October 16–19 2016. ACM.

[74] Manoj Gulati, Farshid Salemi Parizi, Eric Whitmire, Sidhant Gupta, Shobha Sundar Ram, Amarjeet Singh, and Shwetak N. Patel. Capharvester: A stick-on capacitive energy harvester using stray electric field from ac power lines. *ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2(3):110:1–110:20, September 2018.

[75] Philip J. Guo. Online python tutor: Embeddable web-based program visualization for CS education. In *Proc. SIGCSE*, pages 579–584, Denver, CO, USA, 2013. ACM.

[76] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. Workload Characterization Workshop*, pages 3–14, Austin, TX, USA, 2001. IEEE.

[77] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. Workload Characterization Workshop*, pages 3–14, Austin, TX, USA, 2001. IEEE. https://doi.org/10.1109/wwc.2001.990739.

[78] Carl Hartung, Richard Han, Carl Seielstad, and Saxon Holbrook. FireWxNet: A multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. In *Proc. MobiSys*, pages 28–41, Uppsala, Sweden, 2006. ACM.

[79] Sara Heitlinger, Nick Bryan-Kinns, and Rob Comber. The right to the sustainable smart city. In *Proc. CHI*, pages 317:1–317:13, Glasgow, Scotland, UK, 2019. ACM.

[80] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (Fifth Edition)*. Morgan Kaufman, Burlington, MA, USA, 2012.

[81] Mehrdad Hessar, Ali Najafi, and Shyamnath Gollakota. NetScatter: Enabling large-scale backscatter networks. In *Proc. NSDI*, pages 271–283, Boston, MA, USA, February 26–28 2019. USENIX.

[82] Josiah Hester, Timothy Scott, and Jacob Sorber. Ekho: Realistic and Repeatable Experimentation for Tiny Energy-Harvesting Sensors. In *Proc. SenSys*, pages 330–331, Memphis, TN, USA, 2014. ACM.

[83] Josiah Hester, Lanny Sitanayah, and Jacob Sorber. Tragedy of the coulombs: Federating energy storage for tiny, intermittently-powered sensors. In *Proc. SenSys*, pages 5–16, Seoul, South Korea, 2015. ACM.

[84] Josiah Hester and Jacob Sorber. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proc. SenSys*, pages 19:1–19:13, Delft, The Netherlands, 2017. ACM.

[85] Josiah Hester and Jacob Sorber. The Future of Sensing is Batteryless, Intermittent, and Awesome. In *Proc. SenSys*, pages 21:1–21:6, Delft, The Netherlands, 2017. ACM.

[86] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proc. SenSys*, pages 17:1–17:13, Delft, The Netherlands, 2017. ACM.

[87] Josiah Hester, Nicole Tobias, Amir Rahmati, Lanny Sitanayah, Daniel Holcomb, Kevin Fu, Wayne P. Burleson, and Jacob Sorber. Persistent clocks for batteryless sensing devices. *ACM Trans. Embed. Comput. Syst.*, 15(4):77:1–77:28, August 2016.

[88] Matthew Hicks. Clank: Architectural support for intermittent computation. In *Proc. ISCA*, pages 228–240, Toronto, ON, Canada, 2017. ACM.

[89] Matthew Hicks. Ratchet (source code from OSDI 2016). https://github.com/impedimentToProgress/Ratchet, January 2017. Last accessed: Nov. 5, 2021.

[90] Jason Hill, Mike Horton, Ralph Kling, and Lakshman Krishnamurthy. The platforms enabling wireless sensor networks. *Communications of the ACM*, 47(6):41–46, June 2004.

[91] Ali Hoseinghorban, Mohammad Abbasinia, and Alireza Ejlali. PROWL: A cache replacement policy for consistency aware renewable powered devices. *IEEE Trans. Emerging Top. Comput.*, 10(1):476–487, Jan.-March 2020. https://doi.org/10.1109/TETC.2020.3031114.

[92] Ali Hoseinghorban, Amir Mahdi Hosseini Monazzah, Mostafa Bazzaz, Bardia Safaei, and Alireza Ejlali. COACH: Consistency aware check-pointing for nonvolatile processor in energy harvesting systems. *IEEE Trans. Emerging Top. Comput.*, 9(4):2076–2088, October 2021. https://doi.org/10.1109/TETC.2019.2961007.

[93] Meng-Ju Hsieh, Jr-Ling Guo, Chin-Yuan Lu, Han-Wei Hsieh, Rong-Hao Liang, and Bing-Yu Chen. RFTouchPads: Batteryless and wireless modular touch sensor pads based on RFID. In *Proc. UIST*, pages 999–1011, New Orleans, LA, US, October 20–23 2019. ACM.

[94] Meng-Ju Hsieh, Rong-Hao Liang, Da-Yuan Huang, Jheng-You Ke, and Bing-Yu Chen. RFIBricks: Interactive building blocks based on RFID. In *Proc. CHI*, pages 1–10, Montréal, QC, Canada, 2018. ACM.

[95] Kaori Ikematsu, Masaaki Fukumoto, and Itiro Siio. Ohmic-sticker: Force-to-motion type input device for capacitive touch surface. In *Proc. CHI*, pages LBW0223:1–LBW0223:6, Glasgow, Scotland, UK, 2019. ACM.

[96] Texas Instruments. Texas Instruments msp430fr599x, msp430fr596x mixed-signal microcontrollers. https://www.ti.com/lit/ds/symlink/msp430fr5994.pdf, January 2021. Last accessed: Oct. 19, 2022.

[97] Intel Corp. Intel 64 and IA-32 architectures software developer's manual; combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d and 4, April 2022. https://cdrdv2.intel.com/v1/dl/getContent/671200.

[98] RISC-V International. Official RISC-V website. https://riscv.org/about/, October 2022. Last accessed: Oct. 17, 2022.

[99] Vikram Iyer, Elyas Bayati, Rajalakshmi Nandakumar, Arka Majumdar, and Shyam Gollakota. Charging a smartphone across a room using lasers. *ACM Interact. Mob. Wearable Ubiquitous Technol.*, 1(4):143:1–143:21, December 2017.

[100] Jennifer Jacobs and Leah Buechley. Codeable objects: Computational design and digital fabrication for novice programmers. In *Proc. CHI*, pages 1589–1598, Paris, France, 2013. ACM.

[101] Ravi Jain and John Wullert II. Challenges: Environmental design for pervasive computing systems. In *Proc. MobiCom*, pages 263–270, Atlanta, GA, USA, 2002. ACM.

[102] Junsu Jang and Fadel Adib. Underwater backscatter networking. In *Proc. SIGCOMM*, pages 187–199, Beijing, China, 2019. ACM.

[103] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. Quickrecall: A HW/SW approach for computing across power cycles in transiently powered computers. *J. Emerg. Technol. Comput. Syst.*, 12(1):8:1–8:19, July 2015.

[104] Asangi Jayatilaka, Quoc Hung Dang, Shengjian Jammy Chen, Renuka Visvanathan, Christophe Fumeaux, and Damith C. Ranasinghe. Designing batteryless wearables for hospitalized older people. In *Proc. ISWC*, pages 91–95, London, UK, September 9–13 2019. ACM.

[105] Xiaofan Jiang, Joseph Polastre, and David Culler. Perpetual environmentally powered sensor networks. In *Proc. IPSN*, pages 1–12, Los Angeles, CA, USA, 2005. ACM/IEEE.

[106] Haojian Jin, Jingxian Wang, Zhijian Yang, Swarun Kumar, and Jason Hong. WiSh: Towards a wireless shape-aware world using passive RFIDs. In *Proc. MobiSys*, pages 428–441, Munich, Germany, June 10–15 2018. ACM.

[107] Kumara Kahatapitiya, Chamod Weerasinghe, Jinal Jayawardhana, Hiranya Kuruppu, Kanchana Thilakarathna, and Dileeka Dias. Low-power step counting paired with electromagnetic energy harvesting for wearables. In *Proc. ISWC*, pages 218–219, Singapore, 2018. ACM.

[108] Pouya Kamalinejad, Chinmaya Mahapatra, Zhengguo Sheng, Shahriar Mirabbasi, Victor C.M. Leung, and Yong Liang Guan. Wireless energy harvesting for internet of things. *IEEE Commun. Mag.*, 53(6):102–108, June 2015.

[109] Hyeonsu Kang and Philip J. Guo. Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations. In *Proc. USIT*, pages 737–745, Québec City, QC, Canada, 2017. ACM.

[110] Mustafa Emre Karagozler, Ivan Poupyrev, Gary K. Fedder, and Yuri Suzuki. Paper generators: Harvesting energy from touching, rubbing and sliding. In *Proc. UIST*, pages 23–30, St. Andrews, UK, October 8–11 2013. ACM.

[111] Keiko Katsuragawa, Ju Wang, Ziyang Shan, Ningshan Ouyang, Omid Abari, and Daniel Vogel. Tip-Tap: Battery-free discrete 2D fingertip input. In *Proc. UIST*, pages 1045–1057, New Orleans, LA, US, October 20–23 2019. ACM.

[112] Bryce Kellogg, Aaron Parks, Shyamnath Gollakota, Joshua R. Smith, and David Wetherall. Wi-Fi backscatter: Internet connectivity for RF-powered devices. In *Proc. SIGCOMM*, pages 607–618, Chicago, IL, USA, August 17–22, 2014. ACM.

[113] Ben Kenwright. Fast efficient fixed-size memory pool: No loops and no overhead. In *Proc. Computation Tools*, Nice, France, July 22–27 2012. IARIA.

[114] Azam Khan. Swimming upstream in sustainable design. *Interactions*, 18(5):12—-14, September 2011.

[115] Hyung-Sin Kim, Michael P. Andersen, Kaifei Chen, Sam Kumar, William J. Zhao, Kevin Ma, and David E. Culler. System architecture directions for Post-SoC/32-bit networked sensors. In *Proc. SenSys*, pages 264–277, Shenzhen, China, 2018. ACM.

[116] Bran Knowles, Lynne Blair, Mike Hazas, and Stuart Walker. Exploring sustainability research in computing: Where we are and where we go next. In *Proc. UbiComp*, pages 305–314, Zurich, Switzerland, 2013. ACM.

[117] Vito Kortbeek, Abu Bakar, Stefany Cruz Kasım Sinan Yıldırım, Przemysław Pawełczak, and Josiah Hester. BFree: Enabling battery-free sensor prototyping with python. *ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(4):135:1–111:39, December 2020. https://doi.org/10.1145/3432191.

[118] Vito Kortbeek, Souradip Ghosh, Josiah Hester, Simone Campanoni, and Przemysław Pawełczak. WARio: Efficient code generation for intermittent computing. In *Proc. PLDI*, pages 777–791, San Diego, CA, USA, 2022. ACM. https://doi.org/10.1145/3519939.3523454.

[119] Vito Kortbeek, Kasım Sinan Yıldırım, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. Time-sensitive intermittent computing meets legacy software. In *Proc. ASPLOS*, pages 85–99, Lausanne, Switzerland, 2020. ACM.

[120] Stacey Kuznetsov and Eric Paulos. Upstream: Motivating water conservation with low-cost water flow sensing and persuasive displays. In *Proc. CHI*, pages 1851–1860, Atlanta, GA, USA, 2010. ACM.

[121] Koen Langendoen, Aline Baggio, and Otto Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *Proc. International Parallel & Distributed Processing Symposium*, pages 1–12, Rhodes Island, Grece, 2006. IEEE.

[122] Gierad Laput, Yang Zhang, and Chris Harrison. Synthetic sensors: Towards general-purpose sensing. In *Proc. CHI*, pages 3986–3999, Denver, CO, USA, 2017. ACM.

[123] Fredrik Larsson and Bengt-Erik Mellander. Abuse by external heating, overcharge and short circuiting of commercial lithium-ion battery cells. *Journal of The Electrochemical Society*, 161(10):A1611–A1617, 2014.

[124] Cheng-Ting Lee, Yun-Hao Liang, Pai H. Chou, Ali Heydari Gorji, Seyede Mahya Safavi, Wen-Chan Shih, and Wen-Tsuen Chen. Ecomicro: A miniature self-powered inertial sensor node based on bluetooth low energy. In *Proc. ISLPED*, pages 30:1–30:6, Seattle, WA, USA, 2018. ACM.

[125] Seulki Lee, Bashima Islam, Yubo Luo, and Shahriar Nirjon. Intermittent learning: On-device machine learning on intermittently powered system. *ACM Interact. Mob. Wearable Ubiquitous Technol.*, 3(4):141:1–141:30, December 2019.

[126] Philip Levis, Sam Madden, Joseph Polastre, Rober Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. TinyOS: An operating system for sensor networks. In Werner Weber, Jan M. Rabaey, and Emile Aarts, editors, *Ambient intelligence*, pages 115–148. Springer, Berlin, Germany, 2005.

[127] Dong Li, Feng Ding, Qian Zhang, Run Zhao, Jinshi Zhang, and Dong Wang. TagController: A universal wireless and battery-free remote controller using passive RFID tags. In *Proc. MobiQuitous*, pages 166–175, Melbourne, VIC, Australia, November 7–10 2017. ACM.

[128] Hanchuan Li, Eric Brockmeyer, Elizabeth J. Carter, Josh Fromm, Scott E. Hudson, Shwetak N. Patel, and Alanson Sample. PaperID: A technique for drawing functional battery-free wireless interfaces on paper. In *Proc. CHI*, pages 5885–5896, San Jose, CA, USA, 2016. ACM.

[129] Tianxing Li and Xia Zhou. Battery-free eye tracker on glasses. In *Proc. MobiCom*, pages 67–82, New Delhi, India, 2018. ACM.

[130] Yichen Li, Tianxing Li, Xing-Dong Yang Ruchir A. Patel, and Xia Zhou. Self-powered gesture recognition with ambient light. In *Proc. UIST*, pages 595–608, Berlin, Germany, October 14–17 2018. ACM.

[131] Rong-Hao Liang, Meng-Ju Hsieh, Jheng-You Ke, Jr-Ling Guo, and Bing-Yu Chen. RFIMatch: Distributed batteryless near-field identification using RFID-tagged magnet-biased reed switches. In *Proc. UIST*, pages 473–483, Berlin, Germany, October 14–17 2018. ACM.

[132] Silvia Lindtner, Garnet Hertz, and Paul Dourish. Emerging Sites of CHI Innovation: Hackerspaces, Hardware Startups & Incubators. In *Proc. CHI*, pages 439–448, Toronto, ON, Canada, 2014. ACM.

[133] Szu-Yu (Cyn) Liu, Shaowen Bardzell, and Jeffrey Bardzell. Symbiotic encounters: Hci and sustainable agriculture. In *Proc. CHI*, pages 317:1–317:13, Glasgow, Scotland, UK, 2019. ACM.

[134] Jack L. Lo and Susan J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *Proc. PLDI*, pages 151–162, La Jolla, CA, USA, 1995. ACM. https://doi.org/10.1145/207110.207132.

[135] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent Computing: Challenges and Opportunities. In *Proc. SNAPL*, pages 8:1–8:14, Alisomar, CA, USA, 2017.

[136] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proc. PLDI*, pages 575–585, Portland, OR, USA, 2015. ACM.

[137] Dong Ma, Guohao Lan, Mahbub Hassan, Wen Hu, Mushfika Baishakhi Upama, Ashraf Uddin, and Moustafa Youssef. Solargest: Ubiquitous and battery-free gesture recognition using solar cells. In *Proc. MobiCom*, pages 12:1–12:15, Los Cabos, Mexico, 2019. ACM.

[138] Kaisheng Ma, Xueqing Li, Karthik Swaminathan, Yang Zheng, Shuangchen Li, Yongpan Liu, Yuan Xie, John Jack Sampson, and Vijaykrishnan Narayana. Nonvolatile processor architectures: Efficient, reliable progress with unstable power. *Micro*, 36(3):72–83, May–Jun. 2016. https://doi.org/10.1109/MM.2016.35.

[139] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. Architecture exploration for ambient energy harvesting nonvolatile processors. In *Proc. HPCA*, pages 526–537, Burlingame, CA, USA, 2015. IEEE.

[140] Yunfei Ma, Zhihong Luo, Christoph Steiger, Giovanni Traverso, and Fadel Adib. Enabling Deep-Tissue Networking for Miniature Medical Devices. In *Proc. SIGCOMM*, pages 417–431, Budapest, Hungary, 2018. ACM.

[141] Yunfei Ma, Nicholas Selby, and Fadel Adib. Drone relays for battery-free networks. In *Proc. SIGCOMM*, pages 335—-347, Los Angeles, CA, USA, 2017. ACM.

[142] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent Execution without Checkpoints. In *Proc. OOPSLA*, pages 96:1–96:30, Vancouver, BC, Canada, 2017. ACM.

[143] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *Proc. OSDI*, pages 129–144, Carlsbad, CA, USA, 2018. USENIX.

[144] Kiwan Maeng and Brandon Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proc. PLDI*, page 1101–1116, Phoenix, AZ, USA, 2019. ACM. https://doi.org/10.1145/3314221.3314613.

[145] Kiwan Maeng and Brandon Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proc. PLDI*, pages 1101–1116, Phoenix, AZ, USA, June 22–26 2019. ACM.

[146] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proc. International Workshop on Wireless Sensor Networks and Applications*, pages 88–97, Atlanta, GA, USA, 2002. ACM.

[147] Andrea Maioli and Luca Mottola. ALFRED: Virtual memory for intermittent computing. In *Proc. SenSys*, Coimbra, Portugal, 2021. ACM. https://arxiv.org/pdf/2110.07542.pdf.

[148] Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Discovering the hidden anomalies of intermittent computing. In *Proc. EWSN*, Delft, The Netherlands, 2021.

[149] Jennifer C. Mankoff, Eli Blevis, Alan Borning, Batya Friedman, Susan R. Fussell, Jay Hasbrouck, Allison Woodruff, and Phoebe Sengers. Environmental sustainability and interaction. In *Proc. CHI*, pages 2121–2124, San Jose, CA, USA, 2007. ACM.

[150] IHS Markit. The internet of things: a movement, not a market. Technical report, IHS Markit, October 2017. https://cdn.ihs.com/www/pdf/IoT_ebook.pdf.

[151] Gaia Maselli, Mauro Piva, Giorgia Ramponi, and Deepak Ganesan. Demo: JoyTag: a battery-less videogame controller exploiting RFID backscattering. In *Proc. MobiCom*, pages 515–516, New York City, NY, USA, October 3–7 2016. ACM.

[152] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, Brian Homerding, Tommy McMichen, David I. August, and Simone Campanoni. NOELLE Offers Empowering LLvm Extensions. In *Proc. CGO*, Seoul, South Korea, 2022. ACM.

[153] Clive Maxfield. Python is better than c! (or is it the other way round?). https://www.embedded.com/python-is-better-than-c-or-is-it-the-other-way-round, March 2016. Last accessed: Oct. 27, 2020.

[154] Will McGrath, Daniel Drew, Jeremy Warner, Majeed Kazemitabaar, Mitchell Karchemsky, David Mellis, and Björn Hartmann. Bifröst: Visualizing and checking behavior of embedded systems across hardware and software. In *Proc. USIT*, pages 299–310, Québec City, QC, Canada, 2017. ACM.

[155] William McGrath, Jeremy Warner, Mitchell Karchemsky, Andrew Head, Daniel Drew, and Bjoern Hartmann. Wifröst: Bridging the information gap for debugging of networked embedded systems. In *Proc. UIST*, pages 447–455, Berlin, Germany, 2018. ACM.

[156] Yogesh Kumar Meena, Krishna Seunarine, Deepak Ranjan Sahoo, Simon Robinson, Jennifer Pearson, Chi Zhang, Matt Carnie, Adam Pockett, Andrew Prescott, Suzanne K. Thomas, Harrison Ka Hin Lee, and Matt Jones. Pv-tiles: Towards closely-coupled photovoltaic and digital materials for useful, beautiful and sustainable interactive surfaces. In *Proc. CHI*, Honolulu, HI, USA, 2020. ACM.

[157] Hashan Roshantha Mendis and Pi-Cheng Hsiu. Accumulative display updating for intermittent systems. *ACM Transactions on Embedded Computing Systems*, 18(5s):72:1–72:22, October 2019.

[158] Geoff V. Merrett and Bashir M. Al-Hashimi. Energy-driven computing: Rethinking the design of energy harvesting systems. In *Proc. DATE*, pages 960–965, Lausanne, Switzerland, March 27–31 2017. IEEE.

[159] Microchip. MIC841/2 comparator with 1.25% reference and adjustable hysteresis. http://ww1.microchip.com/downloads/en/DeviceDoc/20005758A.pdf, April 2017. Last accessed: Oct. 27, 2020.

[160] Microsoft. Makecode: Hands on computing education. https://www.microsoft.com/en-us/makecode, 2020. Last accessed: Oct. 27, 2020.

[161] MIT Media Lab. Scratch Programming Language Official Website. https://scratch.mit.edu, 2002. Last accessed: Oct. 27, 2020.

[162] Iqbal Mohomed and Prabal Dutta. The age of DIY and dawn of the maker movement. *GetMobile*, 18(4):41—43, October 2014.

[163] Saman Naderiparizi, Mehrdad Hessar, Vamsi Talla, Shyamnath Gollakota, and Joshua R. Smith. Towards battery-free HD video streaming. In *Proc. NSDI*, pages 233–247, Renton, WA, USA, 2018. USENIX.

[164] Saman Naderiparizi, Aaron N. Parks, Zerina Kapetanovic, Benjamin Ransford, and Joshua R. Smith. WISPCam: A battery-free RFID camera. In *Proc. IEEE RFID*, pages 166–173, San Diego, CA, USA, September 15–17, 2015. IEEE.

[165] Saman Naderiparizi, Yi Zhao, James Youngquist, Alanson P. Sample, and Joshua R. Smith. Self-localizing battery-free cameras. In *Proc. UbiComp*, pages 445–449, Osaka, Japan, September 7–11 2015. ACM.

[166] Yuji Nakamura. Peak video game? top analyst sees industry slumping in 2019. https://www.bloomberg.com/news/articles/2019-01-23/peak-video-game-top-analyst-sees-industry-slumping-in-2019, January 2019. Last accessed: Jan. 7, 2020.

[167] Matteo Nardello, Harsh Desai, Davide Brunelli, and Brandon Lucia. Camaroptera: a batteryless long-range remote visual sensing system. In *Proc. ENSsys*, pages 8–14, New York, NY, USA, November 10 2019. ACM.

[168] NeDRo. 6 v 0.6 w 80×55 mm mini solar panel. https://etronixcenter.com/en/solar-panels-and-wind-turbines/8168876-al103-nedro-6v-06w-80x55mm-mini-solar-panel-7110218865414.html, August 2020. Last accessed: Oct. 27, 2020.

[169] Phuc Nguyen, Ufuk Muncuk, Ashwin Ashok, Kaushik Roy Chowdhury, Marco Gruteser, and Tam Vu. Battery-free identification token for touch sensing devices. In *Proc. SenSys*, pages 109–122, Stanford, CA, USA, November 14–16 2016. ACM.

[170] Nintendo Co., Ltd. Dedicated video game sales units. https://www.nintendo.co.jp/ir/en/finance/hard_soft/index.html, September 2019. Last accessed: Apr. 30, 2020.

[171] Nintendo Co., Ltd. Nintendo game boy. https://en.wikipedia.org/wiki/Game_Boy, July 2020. Last accessed: Jul. 23, 2020.

[172] NXP. User manual for NXP real time clocks PCF85x3, PCF85x63, PCA8565, PCF2123, and PCA21125. https://www.nxp.com/docs/en/user-guide/UM10301.pdf, July 2015. Last accessed: Oct. 27, 2020.

[173] Delft University of Technology Sustainable Systems Lab. BFree artifact. https://anonymized.artifact, April 2022. Last accessed: Apr. 07, 2022.

[174] Delft University of Technology Sustainable Systems Lab. BFree source code repository. https://anonymized.repository, March 2022. Last accessed: Mar. 18, 2022.

[175] Stephen O'Grady. The RedMonk Programming Language Rankings: June 2019. https://redmonk.com/sogrady/2019/07/18/language-rankings-6-19, July 2019. Last accessed: Oct. 27, 2020.

[176] Kazuya Oharada, Buntarou Shizuki, and Shin Takahashi. Acceltag: A passive smart ID tag with an acceleration sensor for interactive applications. In *Proc. UIST Adjunct*, pages 63–64, Québec City, Canada, October 22–25 2017. ACM.

[177] Open Source Community Contributors. Clang 7 libtooling. https://github.com/llvm-mirror/clang/blob/master/docs/LibTooling.rst, March 2019. Last accessed: Jan. 20, 2020.

[178] Open Source Community Contributors. C language family front-end. https://github.com/llvm/llvm-project/tree/main/clang, June 2021. Last accessed: Nov. 10, 2021.

[179] Open Source Community Contributors. GNU compiler collection. https://gcc.gnu.org/git.html, July 2021. Last accessed: Jul. 21, 2021.

[180] Open Source Community Contributors. The LLVM compiler infrastructure. https://github.com/llvm/llvm-project, June 2021. Last accessed: Jun. 28, 2021.

[181] Open Source Community Contributors. Qemu: a generic and open source machine and userspace emulator and virtualizer. https://gitlab.com/qemu-project/qemu, November 2021. Last accessed: Nov. 5, 2021.

[182] Open Source Community Contributors. Tiny AES in C. https://github.com/kokke/tiny-aes-c, January 2021. Last accessed: Nov. 5, 2021.

[183] Open Source Community Contributors. Unicorn: a lightweight, multi-platform, multi-architecture CPU emulator framework based on qemu. https://github.com/unicorn-engine/unicorn, October 2021. Last accessed: Nov. 5, 2021.

[184] Open Source Community Contributors. Whole program LLVM in Go. https://github.com/SRI-CSL/gllvm, February 2021. Last accessed: Nov. 10, 2021.

[185] Open Source Community Contributors. Clang: a c language family frontend for llvm. https://clang.llvm.org/, October 2022. Last accessed: Oct. 17, 2022.

[186] Open Source Community Contributors. ICEmu: Intermittent computing emulator. https://github.com/tudssl/ICEmu/, January 2022. Last accessed: Oct. 17, 2022.

[187] Panic Inc. Playdate console home page. https://play.date, March 2020. Last accessed: May 2, 2020.

[188] Joseph A. Paradiso and Thad Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Comput.*, 4(1):18–27, Jan.–Mar. 2005.

[189] Aaron N. Parks, Angli Liu, Shyamnath Gollakota, and Joshua R. Smith. Turbocharging ambient backscatter communication. In *Proc. SIGCOMM*, pages 619–630, Chicago, IL, USA, August 17–22, 2014. ACM.

[190] Matthai Philipose, Joshua R. Smith, Bing Jiang, Alexander Mamishev, Sumit Roy, and Kishor Sundara-Rajan. Battery-free wireless identification and sensing. *IEEE Pervasive Comput.*, 4(1):37–45, Jan.–Mar. 2005.

[191] James Pierce and Eric Paulos. Materializing energy. In *Proc. DIS*, pages 113–122, Aarhus, Denmark, 2010. ACM.

[192] James Pierce and Eric Paulos. Beyond energy monitors: Interaction, energy, and emerging energy systems. In *Proc. CHI*, pages 2367–2376, Austin, TX, USA, 2012. ACM.

[193] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: Enabling ultra-low power wireless research. In *Proc. IPSN*, pages 1–12, Los Angeles, CA, USA, 2005. ACM/IEEE.

[194] Powercast Corp. Powercast hardware development kits website. https://www.powercastco.com/products/development-kits/, 2014. Last accessed: Jan. 20, 2020.

[195] Jothi Prasanna Shanmuga Sundaram, Wan Du, and Zhiwei Zhao. A survey on LoRa networking: Research problems, current solutions, and open issues. *IEEE Commun. Surveys Tuts.*, 22(1):371–388, First Quarter 2020.

[196] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, October 2000.

[197] United Nations Environment Programme. Playing for the planet consortium. https://playing4theplanet.org, April 2020. Last accessed: Apr. 28, 2020.

[198] Vijay Raghunathan, Aman Kansal, Jason Hsu, Jonathan Friedman, and Mani Srivastava. Design considerations for solar energy harvesting wireless embedded systems. In *Proc. IPSN*, pages 457–462, Boise, ID, USA, April 15 2005. ACM/IEEE. https://doi.org/10.1109/IPSN.2005.1440973.

[199] Amir Rahmati, Mastooreh Salajegheh, Dan Holcomb, Jacob Sorber, Wayne P. Burleson, and Kevin Fu. TARDIS: time and remanence decay in SRAM to implement secure protocols on embedded devices without clocks. In *Proc. Security Symposium*, pages 36–36, Bellevue, WA, USA, 2012. USENIX.

[200] Damith C. Ranasinghe, Roberto L. Shinmoto Torres, Alanson P. Sample, Joshua R. Smith, Keith Hill, and Renuka Visvanathan. Towards falls prevention: a wearable wireless and battery-less sensing and automatic identification-tag for real time monitoring of human movements. In *Proc. EMBC*, pages 6402–6405, San Diego, CA, USA, 2012. IEEE.

[201] Benjamin Ransford. Traces repository used in ‘mementos: System support for long-running computation on RFID-scale devices‘ paper. https://github.com/ransford/mspsim/tree/mementos/traces, October 2011. Last accessed: Nov. 1, 2021.

[202] Benjamin Ransford and Brandon Lucia. Nonvolatile memory is a broken time machine. In *Proc. Memory Systems Performance and Correctness Workshop*, pages 1–3, Edinburgh, United Kingdom, 2014. ACM. https://doi.org/10.1145/2618128.2618136.

[203] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System Support for Long-running Computation on RFID-scale Devices. In *Proc. ASPLOS*, pages 159–170, Newport Beach, CA, USA, 2011. ACM.

[204] Saul Rodriguez, Stig Ollmar, Muhammad Waqar, and Ana Rusu. A batteryless sensor ASIC for implantable bio-impedance applications. *IEEE Trans. Biomed. Circuits Syst.*, 10(3):533–544, June 2016.

[205] Paulo Rosa, Federico Ferretti, Ângela Guimarães Pereira, Francesco Panella, and Maximilian Wanner. Overview of the maker movement in the european union. Technical report, European Union, 2017. https://core.ac.uk/download/pdf/132627140.pdf.

[206] Kimiko Ryokai, Peiqi Su, Eungchan Kim, and Bob Rollins. Energybugs: Energy harvesting wearables for children. In *Proc. CHI*, pages 1039–1048, Toronto, ON, Canada, 2014. ACM.

[207] Ali Saffari, Mehrdad Hessar, Saman Naderiparizi, and Joshua R. Smith. Battery-free wireless video streaming camera system. In *Proc. RFID*, Phoenix, AZ, USA, April 2–4 2019. IEEE.

[208] Saleae. Saleae logic pro 8 analyzer. http://downloads.saleae.com/specs/Logic+Pro+8+Data+Sheet.pdf, 2020. Last accessed: Jun. 22, 2020.

[209] Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. Design of an RFID-based battery-free programmable sensing platform. *IEEE Trans. Instrum. Meas.*, 57(11):2608–2615, November 2008.

[210] Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. Design of an RFID-based battery-free programmable sensing platform. *IEEE Trans. Instrum. Meas.*, 57(11):2608–2615, November 2008.

[211] Takuya Sasatani, Chouchang Jack Yang, Matthew J. Chabalko, Yoshihiro Kawahara, and Alanson P. Sample. Room-wide wireless charging and load-modulation communication via quasistatic cavity resonance. *ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2(4):188:1–188:23, December 2018.

[212] Mahadev Satyanarayanan, Wei Gao, and Brandon Lucia. The computing landscape of the 21st century. In *Proc. HotMobile*, pages 45–50, Santa Cruz, CA, USA, 2019. ACM.

[213] Rishi Shukla, Neev Kiran, Rui Wang, Jeremy Gummeson, and Sunghoon Ivan Lee. SkinnyPower: Enabling batteryless wearable sensors via intra-body power transfer. In *Proc. SenSys*, pages 55–67, New York City, NY, USA, 2019. ACM.

[214] SiFive, Inc. SiFive E21 core complex manual 21g3.02.00. https://sifive.cdn.prismic.io/sifive/6f0f1515-1249-4ea0-a5b2-79d4aaf920ae_e21_core_complex_manual_21G3.pdf, December 2021. Last accessed: Oct. 16, 2022.

[215] Philip Sparks. The route to a trillion devices: The outlook for IoT investment to 2035. Technical report, ARM Limited, June 2017. https://pages.arm.com/rs/312-SAX-488/images/Arm-The-route-to-trillion-devices_2018.pdf.

[216] Andrew Spielberg, Alanson Sample, Scott E. Hudson, Jennifer Mankoff, and James McCann. RapID: A framework for fabricating low-latency interactive objects with RFID tags. In *Proc. CHI*, pages 5897–5908, San Jose, CA, USA, 2016. ACM.

[217] Evan Strasnick, Maneesh Agrawala, and Sean Follmer. Scanalog: Interactive design and debugging of analog circuits with programmable hardware. In *Proc. USIT*, pages 321–330, Québec City, QC, Canada, 2017. ACM.

[218] Valerie Sugarman and Edward Lank. Designing persuasive technology to manage peak electricity demand in ontario homes. In *Proc. CHI*, pages 1975–1984, Seoul, Republic of Korea, 2015. ACM.

[219] Yulei Sui and Jingling Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *Proc. CC*, pages 265–266, Barcelona, Spain, 2016. ACM. https://doi.org/10.1145/2892208.2892235.

[220] Kazunobu Sumiya, Takuya Sasatani, Yuki Nishizawa, Kenji Tsushio, Yoshiaki Narusue, and Yoshihiro Kawahara. Alvus: A reconfigurable 2-D wireless charging system. *ACM Interact. Mob. Wearable Ubiquitous Technol.*, 3(2):68:1–68:29, June 2019.

[221] Vamsi Talla, Mehrdad Hassar, Bryce Kellogg, Ali Najafi, Joshua R. Smith, and Shyam Gollakota. LoRa backscatter: Enabling the vision of ubiquitous connectivity. *ACM Interact. Mob. Wearable Ubiquitous Technol.*, 1(3):105:1–105:24, September 2017.

[222] Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua R. Smith. Battery-free cellphone. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 1(2):25:1–25:20, June 2017.

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

[224] Texas Instruments. MSP430FR5994 launchpad development kit. http://www.ti.com/tool/MSP-EXP430FR5994. Last accessed: Jan. 20, 2020.

[225] Texas Instruments Inc. TLV36910.9-V to 6.5-V, nanopower comparator. https://www.ti.com/lit/ds/symlink/tlv3691.pdf, November 2015. Last accessed: Oct. 27, 2020.

[226] Texas Instruments Inc. MSP430FR59xx mixed-signal microcontrollers (Rev. F). http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf, March 2017. Last accessed: May 1, 2020.

[227] The Economist. And Now for Something Completely Different. https://www.economist.com/science-and-technology/2018/07/19/python-has-brought-computer-programming-to-a-vast-new-audience, July 2018. Last accessed: Oct. 27, 2020.

[228] Bill Tomlinson, M. Six Silberman, Don Patterson, Yue Pan, and Eli Blevis. Collapse informatics: Augmenting the sustainability & ICT4D discourse in HCI. In *Proc. CHI*, pages 655–664, Austin, TX, USA, May 5–10 2012. ACM.

[229] Hoang Truong, Shuo Zhang, Ufuk Muncuk, Phuc Nguyen, Nam Bui, Anh Nguyen, Qin Lv, Kaushik Chowdhury, Thang Dinh, and Tam Vu. CapBand: Battery-free successive capacitance sensing wristband for hand gesture recognition. In *Proc. SenSys*, pages 54–67, Shenzhen, China, 2018. ACM.

[230] TU Delft Sustainable Systems Lab. BFree GitHub Repository. https://github.com/tudssl/bfree, October 2020. Last accessed: Oct. 27, 2020.

[231] UNI-T. UT383 mini light meter. https://www.uni-trend.com/html/product/Environmental/Environmental_Tester/Mini/UT383.html, 2020. Last accessed: Jun. 22, 2020.

[232] European Union. Renewable energy statistics. Eurostat, https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Renewable_energy_statistics, January 2022. Last accessed: Dec. 20, 2022.

[233] University of Michigan, MI, USA. Umich moo github page. https://github.com/spqr/umichmoo, February 2011. Last accessed: Apr. 19, 2020.

[234] University of Washington, Seattle, WA, USA. Wireless identification and sensing platform github page. https://github.com/wisp, November 2010. Last accessed: Apr. 19, 2020.

[235] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *Proc. OSDI*, pages 17–32, Savannah, GA, USA, 2016. ACM.

[236] Anandghan Waghmare, Qiuyue Xue, Dingtian Zhang, Yuhui Zhao, Shivan Mittal, Nivedita Arora, Ceara Byrne, Thad Starner, and Gregory D. Abowd. Ubiquitouch: Self sustaining ubiquitous touch interfaces. *ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(1):27:1–27:22, March 2020.

[237] Jingxian Wang, Chengfeng Pan, Haojian Jin, Vaibhav Singh, Yash Jain, Jason Hong, Carmel Majidi, and Swarun Kumar. RFID tattoo: A wireless platform for speech recognition. *ACM Interact. Mob. Wearable Ubiquitous Technol.*, 3(4):155:1–155:24, December 2019.

[238] Ju Wang, Liqiong Chang, Omid Abari, and Srinivasan Keshav. Are RFID sensing systems ready for the real world? In *Proc. MobiSys*, pages 366–377, Seoul, Korea, June 17—21 2019. ACM.

[239] Matt Weinberger. How one woman turned her passion for tinkering into a $33 million business—without a dime of funding. https://www.businessinsider.com/adafruit-industries-limor-fried-on-bootstrapping-a-startup-2015-8, August 2018. Last accessed: Oct. 27, 2020.

[240] WEMOS Electronics. ePaper 2.13 Shield. https://www.wemos.cc/en/latest/d1_mini_shiled/epd_2_13.html, 2019. Last accessed: Oct. 27, 2020.

[241] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proc. OSDI*, pages 381—-396, Seattle, WA, USA, 2006. USENIX.

[242] Harrison Williams, Xun Jian, and Matthew Hicks. Forget failure: Exploiting SRAM data remanence for low-overhead intermittent computation. In *Proc. ASPLOS*, pages 53—-67, Lausanne, Switzerland, 2020. ACM.

[243] Allison Woodruff, Jay Hasbrouck, and Sally Augustin. A bright green perspective on sustainable choices. In *Proc. CHI*, pages 313–322, Florence, Italy, April 5–10 2008. ACM.

[244] Chenren Xu, Lei Yang, and Pengyu Zhang. Practical backscatter communication systems for battery-free internet of things. *IEEE Signal Process. Mag.*, 35(5):16–27, September 2018.

[245] Xieyang Xu, Yang Shen, Junrui Yang, Chenren Xu, Guobin Shen, Guojun Chen, and Yunzhe Ni. PassiveVLC: Enabling practical visible light backscatter communication for battery-free IoT applications. In *Proc. MobiCom*, pages 180–192, Snowbird, UT, USA, 2017. ACM.

[246] Wataru Yamada, Hiroyuki Manabe, and Daizo Ikeda. CamTrackPoint: Camera-based pointing stick using transmitted light through finger. In *Proc. UIST*, pages 313–320, Berlin, Germany, October 14–17 2018. ACM.

[247] Yue Yang, Emenike G. Okonkwo, Guoyong Huang, Shengming Xu, Wei Sun, and Yinghe He. On the sustainability of lithium ion battery industry – a review and perspective. *Energy Storage Mater.*, 36:186–212, April 2021. https://doi.org/10.1016/j.ensm.2020.12.019.

[248] Bahram Yarahmadi and Erven Rohou. So far so good: Self-adaptive dynamic checkpointing for intermittent computation based on self-modifying code. In *Proc. International Workshop on Software and Compilers for Embedded Systems*, pages 1–7, Eindhoven, The Netherlands, 2021. https://hal.inria.fr/hal-03410647/document.

[249] Jeongjin Yeo, Mun ho Ryu, and Yoonseok Yang. Energy harvesting from upper-limb pulling motions for miniaturized human-powered generators. *Sensors*, 15(7):15853–15867, 2015.

[250] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemysław Pawełczak, and Josiah Hester. InK: Reactive kernel for tiny batteryless sensors. In *Proc. SenSys*, pages 41–53, Shenzhen, China, 2018. ACM.

[251] Neng-Hao Yu, Sung-Sheng Tsai, I-Chun Hsiao, Dian-Je Tsai, Meng-Han Lee, Mike Y. Chen, and Yi-Ping Hung. Clip-on gadgets: Expanding multi-touch interaction area with unpowered tactile controls. In *Proc. UIST*, pages 367–371, Santa Barbara, CA, USA, 2011. ACM.

[252] Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay Paddayuru Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung. ReplayCache: Enabling volatile caches for energy harvesting systems. In *Proc. MICRO*, pages 170–182, Virtual Event, 2021. ACM. https://doi.org/10.1145/3466752.3480102.

[253] Chi Zhang, Sidharth Kumar, and Dinesh Bharadia. Capterry: Scalable battery-like room-level wireless power. In *Proc. MobiSys*, pages 1–13, Seoul, Korea, June 17—21 2019. ACM.

[254] Yang Zhang, Yasha Iravantchi, Haojian Jin, Swarun Kumar, and Chris Harrison. Sozu: Self-powered radio tags for building-scale activity sensing. In *Proc. UIST*, pages 973–985, New Orleans, LA, USA, October 20–23 2019. ACM.

[255] Chen Zhao, Sam Yisrael, Joshua R. Smith, and Shwetak N. Patel. Powering wireless sensor nodes with ambient temperature changes. In *Proc. UbiComp*, pages 383––387, Seattle, WA, USA, 2014. ACM.

Acknowledgements

One could argue my journey as a PhD student started when I decided to approach Przemysław (Przemek) Pawełczak in search of a final master's thesis project. I approached him because some years earlier, when I first started my bridging year at the TU Delft, I spotted a QR code that led me to a very specific microcontroller question (about the execution-time of a certain CRC with some input). I have always liked more low-level microcontroller work, and based on this first encounter and his recent publications, I felt Przemek's projects would be a great match. This feeling turned out not to be one-sided, as partway through my master's project, Przemek approached me asking if I would like to pursue a PhD after completing my master's degree, and after some consideration (i.e., could I afford to rent an apartment in Delft with a PhD salary), I agreed.

Looking back, over four years later (or over seven years after that initial email on December 2nd, 2015), I have greatly improved as a researcher, programmer, and in the overarching quality that enables both: problem-solving. All this would not have been possible without my promotors Przemek and Koen. Przemek, you were a great supervisor. You always supported the direction I wanted to go in and helped me in any way you could. I remember the late nights in the office when we were submitting my first paper fondly, partially because you were right there with me. You were always incredibly supportive, and even though you think my emotional range is somewhat limited, do know that I appreciate all the help and mentorship you provided me over the years. We are very different people, me wanting a garden, and you making fun of me enjoying a garden. Nonetheless, we made a great team over the past years. I am sad it is over, but I am also very proud of our accomplishments.

Koen, I have never met someone as good as you at finding "flaws" in ideas you just heard about seconds ago. Somehow you always ask precisely the question that the person pitching the idea did not want you to ask or did not think about. It does not matter what field or topic. However, this is an excellent resource for a PhD student because receiving this feedback early on is infinitely better than during the reviews after a submission. I would have liked to collaborate on a publication together, but it was also great to discuss my ideas with you on a bi-monthly basis.

In addition to my promotors, this journey was only possible through the support of numerous individuals. I want to thank everyone in my research group, "Embedded and Networked Systems" (now sadly separated into the "Embedded Systems" and "Networked Systems" groups), who were great coworkers throughout the years, both academically and socially. Jasper, we were the two primary intermittent computing people within the research group. Additionally, we were office mates and collaborators. We even managed to get a photo of us into the Wall Street Journal. Our different interests and goals within the intermittent computing domain made it work, resulting in some great in-depth discussion. My other two office mates over the years were Nikos and James. Nikos, our work styles match perfectly. It would be dead silent whenever we were alone in the office, with both of us focussing entirely on our work (however, with ample coffee breaks)—the complete opposite of when we were on coffee breaks or having a beer together after bouldering. James, our time as office mates was somewhat short. However, I admired your positive attitude and perseverance even when a mouse temporarily terrorized your house. Your more theoretical approach is an excellent addition to the intermittent computing domain.

I would also like to give a special thanks to my pre-corona bouldering (and afterward drinking) group: Kees (without whom we would have probably not been able to climb any of the walls), Belma, Nikos, and occasionally Eric and Jorik (mainly for the drinking part). I have great memories from all our talks over some beers. Sadly corona swooped in and ruined our flourishing tradition. Additionally, I would like to thank Eric (Weizheng) for always helping to lighten the mood. I always enjoyed our conversations, even the strange ones, and our exciting journey of buying you a bike from Marktplaats that was definitely not stolen. During corona, everyone transitioned to an online form of entertainment. Thanks, Jasper, Jorik, Kees, Talia and Belma for playing some online (board) games while being locked up inside our homes with nowhere to go. It was great fun. I would also like to thank the lunch group during the post-corona days. Eric, Gabe, Miguel, Adrian, Talia, Naram (who is also a great mover), and sometimes even my promotor Koen. All those authentic Chinese meals and burritos were not good for my caloric intake, but they sure were nice, and the conversations always took unexpected twists and turns.

During the four years of my PhD, I also had the pleasure of supervising several master students: Hiram, Felix, and Sourav. With Sourav, I even collaborated on Chapter 6 of this dissertation. He even taught me to make butter chicken after he successfully defended his master's thesis. After all this, I found out that I had been saying his name wrong all along. Thanks, Sourav.

On a technical level, this work was only possible with the help of some excellent collaborators. Przemek, since you are my daily supervisor, I had the pleasure of collaborating with you on all my publications. We might have different interests regarding research topics, but you have always supported me in pursuing what I wanted. This willingness to jump head-first into another research domain has led you to have a rather unique (in my opinion, interesting) publishing record. However, this ability to adapt and still push for top-tier publications, even in a domain we had no experience in, is one of your most vital qualities. Sinan, you were my first real collaborator (excluding Przemek) when we started sketching the idea that resulted in TICS during my master's thesis project. I really appreciated your input and enjoyed our technical discussions. Josiah, we have collaborated on all my published papers, yet we have actually never met in real life (I blame corona). I appreciated your ability to make a good story out of anything and your excellent high-level view of what is important and exciting, something I often overlook while focussing on technical details. Simone, we only collaborated on a single publication together, but it was the one most influential for my future career. Our weekly compiler talks, where I could pitch my ideas or issues, were incredibly beneficial. You could always explain the topic at hand or point me to literature. Jasper, it was a fascinating process to see the Battery-Free Game Boy reach international attention the way it did, and the process of creating it was equally fascinating. Your ability to quickly design and develop hardware designs is incredibly valuable, leading to a very efficient collaboration process. I would also like to thank the rest of my collaborators, Abu Bakar, Jacob, Stefany, Souradip, and Saad, for all their valuable input and hard work. Again, without you, this dissertation would not have been possible. I would have liked to thank you all in more detail, but the size of this acknowledgment chapter is getting out of hand.

I would also like to thank my friends. To the members of Pulletje, Aran, Quinten, Elmar, Kjeld, Roos, Stella and Darrin. We all met during our Electrical Engineering bachelor at the HvA, which (most of us) started all the way back in 2011. Recently, our meetups have become much less frequent, having spread out a bit more, but when we meet, it instantly feels like old times. I would also like to thank the friends I made while pursuing my master's, Jeroen, Thijmen, Lourens, and Bram. Working together to tackle the project-based courses was a lot of fun. I have great memories of tracking red cars, going to Twente (maybe it was due to the fantastic montage), creating Doritos-based Java games, and applying LDPC codes to a picture of a certain former US president. The technical insight gained by working on these projects was paramount for me to write this thesis (at least, that is what I tell myself). Finally, I would like to thank my CSGO friends. In particular, I would like to thank Loek. After I started with the PhD, my time playing CSGO, and my skill, reduced dramatically. However, I still immensely enjoyed occasionally booting up the game and (still) destroying people in Wingman before getting over-confident, queuing for a Faceit game, and getting destroyed equally hard.

Mijn grootste dankbaarheid gaat uit naar mijn familie. Jullie hebben me altijd ondersteund en in me geloofd. Romello, bedankt voor het maken van de mooie boekomslag voor dit proefschrift. Ik weet niet of ik zo technisch geworden zou zijn als jij vroeger niet zo vaak je computer sloopte. Ciro en Massimo, bedankt voor jullie steun en gezelligheid. Mama, ik zou hier niet zijn zonder jou. Jij hebt altijd in mij geloofd, van kleins af aan. Zelfs toen ik van vwo afgeschopt werd omdat mijn Duits en Frans te slecht was, en als ik me erg dom voelde door mijn Dyslexie. Jij was er altijd om te zeggen dat het goed zou komen. Opa Joep en oma Sofie, jullie hebben ook een grote rol gespeeld in de ontwikkeling van mijn liefde voor techniek. Ik heb veel super fijne herinneringen aan het sleutelen aan van alles met opa. Van een eenwieler (met zijwieltjes), een skelter gemaakt met volgens mij vier verschillende wielen, tot een fiets die we met magneetvissen uit de sloot gehaald hebben nadat jij me vertelde dat er magneten in speakers zitten (die zomaar langs de weg stonden). En oma natuurlijk ook bedankt voor het zorgen dat het tijdens al deze bezigheden niet te gevaarlijk werd, en het altijd klaar staan met een biscuitje of een dropje. Verder ben ik ook dankbaar voor de steun van de rest van mijn familie, het zijn te veel namen om hier te noemen, maar jullie weten wie jullie zijn.

Također se želim zahvaliti Belminoj porodici. Hvala vam što ste uvijek sve učinili da se osjećam dobrodošao (čak i do te mjere da sam dobio biftek za doručak i janje na ražnju).

Finally, I would like to thank Belma for all her love and support over the years. We met during the PhD, and you are by far the best thing that came out of this journey. Without hesitation, you are always there for me when I need you. You accept me with all my quirks and never look down on me or judge me for my many, sometimes strange interests, and often join them, which is how you ended up with a custom split keyboard that you had to solder yourself, and attended a CSGO Major in Antwerp. I love you.

Curriculum Vitæ

Vito Kortbeek

Vito Kortbeek was born in Heemskerk, the Netherlands, on 17th May 1994. From a young age, he was interested in figuring out "how stuff works." His first venture into electrical engineering and embedded systems was in high school, where he designed an RC car controlled using hand movements[1]. To pursue this interest further, he decided to study electrical engineering at the Amsterdam University of Applied Sciences, where he honed his applied skillset working on various projects, including a hydrogen prototype car and a solar boat. Instead of joining the workforce, Vito decided he wanted to reach a deeper understanding of all things embedded, leading him to pursue a master's in embedded systems at the Delft University of Technology. Here he learned more theoretical aspects, which perfectly complemented his earlier acquired more applied skillset. During his master's thesis project, he started his journey into intermittent computing, which resulted in an offer to remain at the university after graduating to pursue a PhD.

Throughout the PhD he was a member of the Sustainable Systems Lab (SSL) within the Embedded and Networked Systems group. His research focused mainly on developing and improving methods for supporting incorruptible intermittent computing without programmer intervention and resulted in several publications in a diverse set of research domains. His research was covered internationally by media such as Hackaday, The Wall Street Journal, CNET, Make Magazine, and others.

Afterward, he joined the ASIP (Application-specific instruction-set processors) group within Synopsys, where he is further exploring compilers targeting embedded systems.

[1] https://www.youtube.com/watch?v=qo7odE4eWWQ

List of Publications

4. **Vito Kortbeek**, Souradip Ghosh, Josiah Hester, Simone Campanoni, and Przemysław Pawełczak. *WARio: Efficient Code Generation for Intermittent Computing*. In Proceedings of the International Conference on Programming Language Design and Implementation (PLDI), pages 777–991, June 2022. https://doi.org/10.1145/3519939.3523454.

3. **Vito Kortbeek**, Abu Bakar, Stefany Cruz, Kasım Sinan Yıldırım, Przemysław Pawełczak, and Josiah Hester. *BFree: Enabling Battery-free Sensor Prototyping with Python*. Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT), volume 4, issue 4, pages 1–39, December 2020. https://doi.org/10.1145/3432191.

2. Jasper de Winkel, **Vito Kortbeek**, Josiah Hester, and Przemysław Pawełczak. *Battery-Free Game Boy*. Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT), volume 4, issue 3, pages 1–34, September 2020. https://doi.org/10.1145/3411839. *Distinguished paper award*.

1. **Vito Kortbeek**, Kasım Sinan Yıldırım, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. *Time-sensitive Intermittent Computing Meets Legacy Software*. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 85–99, March 2020. https://doi.org/10.1145/3373376.3378476.