Delft University of Technology
Master of Science Thesis in Embedded Systems

Testing Intermittent Battery-free Systems

Mark Fijneman

TU Delft Delft University of Technology

Testing Intermittent Battery-free Systems

Master of Science Thesis in Embedded Systems

Embedded Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

Mark Fijneman

31st August 2023

Author
Mark Fijneman
mark@markfijneman.com

Title
Testing Intermittent Battery-free Systems

MSc Presentation Date
30 Augustus 2023

Graduation Committee

dr. Przemysław Pawełczak (chairman)	Delft University of Technology
dr. Jasper de Winkel	Delft University of Technology
dr. Soham Chakraborty	Delft University of Technology

Abstract

Existing tools for debugging battery-free applications are limited to specific architectures or require code changes of the *Device Under Test* (DUT) to function. These tools also cannot measure the efficiency of the application designed for battery-free systems. Currently, there is a lack of independent broad comparisons of intermittent systems. Our work, therefore, evaluates state-of-the-art frameworks and their artifacts and finds shortcomings in reproducibility and their performance. To overcome these shortcomings, we introduce DIPS+, a multi-platform debugger and measurement platform for intermittent systems with ARM and MSP430 support. DIPS+ introduces new methods to analyse applications for battery-free systems. One method finds the minimum energy budget required for forward progress, crucial for determining the minimal capacitor size for intermittent systems. Furthermore, DIPS+ offers functions to perform automatic profiling tests, like code start-up time, which gives valuable insights into the system's efficiency. DIPS+ achieves significant improvements in debugging performance, with 11 times faster connection time and reduced code execution by 157 times on the DUT before full reconnecting after intermittency occurs. The evaluation of selected frameworks reveals substantial overheads caused by the additional overhead of saving and restoring of system's state. This causes certain benchmarks to take up to 110 times longer to complete than their uninstrumented counterparts. These findings raise concerns about the viability of task-based approaches as an effective solution for managing intermittency in battery-free IoT devices.

Preface

This thesis, "Testing Intermittent Battery-free Systems," was written to complete my MSc in Embedded Systems at the Delft University of Technology, Faculty of Electrical Engineering, Mathematics, and Computer Science. My research journey began in September 2022 and concluded in August 2023.

A course taught by Przemysław Pawełczak, introduced me to the concept of battery-free systems and, in particular, the fascinating world of communication between systems without a reliable connection. This interest was further deepened during my internship where I developed communication between a mesh of embedded devices and smartphones. There, I experienced firsthand the challenges of debugging embedded devices in real-life applications. Jasper de Winkel introduced me to an essential problem in this domain: the lack of a debugger for MSP430 and automatic tests. Jasper de Winkel, Tom Hoefnagel, Boris Blokland, and Przemysław Pawełczak previously published a paper on DIPS, a debugger for intermittent ARM devices, which acted as a foundation for my own research.

I wish to express my respect and gratitude to my daily supervisor, Jasper de Winkel. The weekly meetings provided valuable insights and helped shape a well-defined and concrete end to my thesis. Secondly, I am thankful to Przemysław Pawełczak for offering me the opportunity and valuable guidance throughout this project. Lastly, I would like to thank Soham Chakraborty for reviewing my thesis and being part of the graduation committee.

Mark Fijneman

Delft, The Netherlands
31st August 2023

Contents

Abbreviations

AR Activity Recognition

BC Bitcount

BF Blowfish

BMP Black Magic Probe

CCS Code Composer Studio

CEM Cold-chain Equipment Monitoring

CF Cuckoo filtering

CRC Cyclic Redundancy Check

DAL Device Abstraction Layer

DUT Device Under Test

EEM Embedded Emulation Module

FRAM Ferroelectric Random Access Memory

GDB GNU Debugger

HAL Hardware Abstraction Layer

IoT Internet of Things

ISA Instruction Set Architecture

JIT Just-In-Time

LPM Low Power Mode

MCU Micro Controller Unit

MPU Memory Protection Unit

NVM Non volatile memory

PTL Protocol Translation Layer

RSA Rivest-Shamir-Adleman

RTO Runtime Overhead

SBW Spy-By-Ware

SDO Shutdown Overhead

SUO Startup Overhead

SWD Serial Wire Debug

VM Virtual Machine

Chapter 1

Introduction

There are almost 16 billion *Internet of Things* (IoT) devices today, with projections for 34 billion devices in 2030 [24]. IoT devices are computationally and energy-restrained devices that communicate over the internet or other networks. Rather than using the main power grid, a large number of these devices use small batteries. However, as the number of devices increases, the number of discarded devices also increases. Projections suggest that by 2025, we will discard around 78 million devices daily [14]. The burden to society from these devices is high due to the environmental impact of making new batteries or the cost of recycling.

Removing the battery prolongs the device's lifespan while having a smaller environmental footprint. An alternative to a battery is using an energy harvester and capacitor. The energy harvester converts ambient energy (e.g. radiowaves [55], solar [22], wind [16] or interactions [37]) into usable power for IoT devices. However, the use of an energy harvester and capacitor also introduces some complexity. Namely that there is not always enough energy to always be powered on. Hence, the devices can only work intermittently. Examples of such intermittent devices are a battery-free gaming console [20] or, more commonly, wireless sensors (e.g. [2]).

In order for an intermittent device to achieve forward progress, the work is divided into multiple parts with ad-hoc or periodic saving of the device's state. When the device loses power and intermittency occurs, the previous state is restored (as illustrated in Fig. 1.1). This approach ensures forward progress instead of attempting to execute the entire program in one uninterrupted session.

Transforming the work into multiple parts is not trivial. Various bugs and errors can occur as the device can experience a power failure at any moment in time. Different solutions are developed that try to solve the problem of maintaining a state while experiencing power failures by offering a framework of methods that act as a foundation for an intermittent resistant application. Unfortunately, only a handful of tools [11, 19] can measure the efficiency and correctness of restoring and saving the device's state. Moreover, there is a lack of broad evaluation in the efficiency of those frameworks and what overhead they cause.

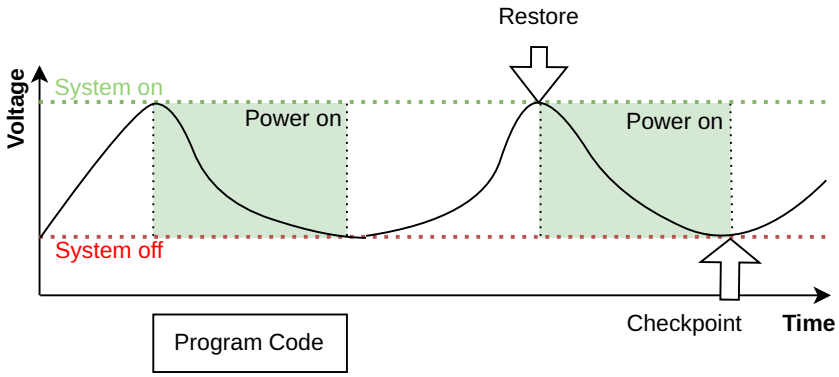


Figure 1.1: **Example of a system saving and restoring state with a checkpointing framework, saves and stores its current state based on the capacitor voltage.**

1.1 Problem Statement

In the field of intermittent battery-free devices, there are several challenges. Firstly, there is a lack of a testing tool that can effectively debug any task-based and checkpointing framework. Existing debuggers [11, 19] are often limited to specific architectures and do not support all major platforms used by intermittent frameworks, such as the MSP430 and Cortex-M0. Secondly, there is a notable absence of comprehensive comparisons among intermittent frameworks, even when they run on the same platform. In addition, the intermittent frameworks are often only evaluated primarily to their intermittent counterparts (E.g. [5, 41, 49, 52]). It does not give a comparison with regard to the most optimal and non-intermittent resistant code, called uninstrumented code. Such comparison would provide a more honest indication of overall performance. Lastly, there is a shortage of tests for larger programs to assess the appropriate placement of checkpoints within the code. In this thesis, we aim to address these problems for intermittent-powered embedded systems by posing the following research question:

> *How can comprehensive runtime analysis for battery-free devices evaluate the reliability and performance of existing frameworks?*

1.2 Contributions

In this thesis, we present four main contributions:

- **Minimum Energy Budget Finder**: We develop an automated solution for determining the minimum energy required to achieve forward progress. The minimum required energy is a crucial metric in determining/optimising capacitor size in intermittent systems.
- **Evaluation of Selected Frameworks**: We analyse a wide range of frameworks for intermittent systems using a wide range of benchmarks.

State-of-the-art papers often only compare their framework with a limited number of other frameworks. Moreover, the efficiency is often only compared with regard to intermittent counterparts and not with uninstrumented code. The evaluation in this thesis provides insights into their efficiency compared to the uninstrumented code. Additionally, during the evaluation we discovered one bug in a framework.

- **Extended Platform Support**: The DIPS debugger [19] has been expanded to include support for MSP430 microcontrollers. This platform is used by 93% of all the published papers about intermittent systems. Additionally, intermittent debugging support has also been added to the debugger for this platform

- **Optimized Debugger Performance**: We optimise the debugging performance and reduce the overhead of debugging. Notably, the connection speed of DIPS+ has been increased by a factor of 11 while reducing the code execution by 157 times before reconnecting after an intermittent event.

Chapter 2

Related Work

2.1 Intermittently-Powered Debugging and Energy Emulation

Software for battery-free devices can be difficult to program and test. For example, we want to make a temperature sensor that works with an intermittent power source. Sometimes this hypothetical sensor gives a value of 70 degrees Celsius while we expect a comfortable 21 degrees Celsius. In such a case, using a debugger to see what is happening inside the sensor can be beneficial. By connecting a hardware debugger to the debug ports of the *Micro Controller Unit* (MCU) of the temperature sensor, we can establish a debug connection. Depending on the features of the hardware debugger, we can use the debug client on the PC to set breakpoints in the code. The program will stop at breakpoints and allow us to check memory and registers or to continue until the next breakpoint. These functionalities can help to find a specific software bug. In the case of the hypothetical temperature sensor, it could be that sometimes the temperature is wrongfully converted to Fahrenheit.

To debug a program on an intermittent device, the debugger needs to cope with the power failures associated with an intermittent device. To prevent the *Device Under Test* (DUT) from switching off, control over the power to the DUT is needed. The DUT cannot be debugged when it is switched off. Table. 2.1 provides an overview of all debuggers.

2.1.1 Debugger

Hardware debuggers, or debug probes, are hardware components that interface with the microcontroller of the DUT. Unfortunately, no single hardware debugger supports all types of microcontrollers. This is because not all microcontrollers share the same *Instruction Set Architecture* (ISA), and certain microcontrollers require different debugging methods.

In the field of intermittently powered systems, two architectures are predominant, with each their own ISA: MSP430FRxx and ARM. Both have low-power modes, which is why they are a popular choice. Additionally, MSP430FRxx

Table 2.1: **Feature comparison of debuggers.**

Framework	J-Link [51]	MSPFET [60]	EDB [11]	DIPS [19]	DIPS+
Intermittent Support	No	No	Yes	Yes	Yes
MSP430 Support	No	Yes	Yes	No	Yes
ARM Support	Yes	No	No	Yes	Yes
IDE Support	Yes	Yes	No	Yes	Yes
GDB Support	Yes	No	No	Yes	Yes
Hardware Breakpoints	Yes	Yes	No	Yes	Yes
Energy Breakpoints	No	No	Yes	Yes	Yes
Minimum Energy Budget finder	No	No	No	No	Yes
Automatic profiling support	No	No	No	No	Yes

offers *Ferroelectric Random Access Memory* (FRAM), which is ideal for storing the state of the devices as it is fast and non-volatile. However, no debugger works out of the box with both architectures.

The commonly used debuggers for each architecture are MSPFET [60] for MSP430 and J-Link [51] for ARM-based boards. Both debuggers can interact with the DUT through a JTAG interface. Unlike MSPFET, J-LINK can directly interact with the popular and feature-rich *GNU Debugger* (GDB). Both debuggers, however, do not have any support for intermittent behaviour of the DUT. Whenever the power fails, the connection is completely lost to the DUT. This means that connecting to the DUT must be manually reestablished.

Fortunately, two debuggers offer intermittent support. EDB [11] only supports MSP430 and is incompatible with GDB. A significant downside of EDB is the need for code changes. All calls to the debugger need to be defined in the code of the DUT. This includes reading memory and setting breakpoints. Consequently, EDB does not allow for a great user experience [19]. The other debugger, DIPS [19], is compatible with GDB and does not require any code changes of the DUT to be used. However, it does support only ARM. As the authors of DIPS noted, it is technically feasible to develop support for MSP430 and no problematic limitations that prevent MSP430 support. Note that in this thesis, the work of DIPS is expended to support MSP430. To make the difference clear, DIPS with MSP430 support is called DIPS+ in this thesis.

2.1.2 Energy Emulation

During a breakpoint in the DUT, it is important to maintain the voltage at the same level to accurately read the device's state and seamlessly resume execution from where the debugger halted (Fig. 2.1). This feature is called Debugger take-over. If Debugger take-over is not present, the DUT will turn off during the debugging, making it no longer possible to debug. Currently, only DIPS and EDB support the Debugger take-over

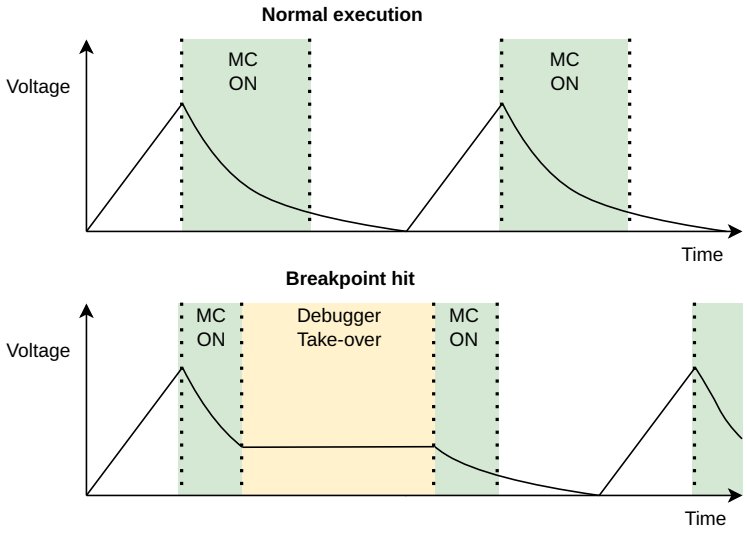


Figure 2.1: **Display of voltage over time for an intermittent device. The top figure shows the normal execution. The bottom figure shows how the voltage is kept at the same level while the DUT is inside a breakpoint. In the yellow zone, the emulator maintains the same voltage as at the start of the breakpoint.**

To further debug particular behaviour, it might be beneficial to simulate specific voltage patterns to invoke certain behaviour from the DUT. Ehko is a recorder of energy harvesting conditions and can also recreate those conditions [31]. Shepherd is an improvement of Ehko as it allows the recording and replaying of voltage patterns for multiple devices and offers more accuracy in recording and emulation [28]. Unlike DIPS and EDB, they do not provide any possibility to detect breakpoints. This prevents Ekho and Shepherd from preventing the shutdown of the DUT. Both EDB and DIPS can use the traces of Ehko to replay energy patterns. DIPS can also simulate certain standard voltage patterns, such as a square wave or a sawtooth.

2.2 Testing Frameworks for Battery-free Devices

Some testing tools are designed explicitly for battery-free devices. They can be divided into static code analysis, simulation-based and hardware-based testing.

Static code analysis only examines the source code without actually executing the program. An example of a static code analysis tool is CleanCut. The tool automatically optimises the placing of task boundaries to prevent some task-specific bugs, such as using too few task boundaries [13]. However, CleanCut does not provide automatic tests for other bugs.

Simulation-based testing does not execute the code on the target microcontroller architecture. Siren [26] introduces the concept of non-volatile memory and energy simulation into the MSPSim emulator [25]. It also allows for energy-neutral *printf*'s, which enables primitive debugging. However, breakpoints must be placed at the machine instruction level and it does not support automatic tests. A similar tool ScEpTIC [45], uses the LLVM compiler [40] and emulator to search for memory inconsistencies caused by the intermittency. This tool can automatically search and find certain kinds of bugs. Nevertheless, since it is only a simulation, inspecting the memory of the DUT or any peripheral is impossible.

An example of hardware-based testing is DIPS [19]. DIPS automated hardware-based testing can be done to find memory inconsistencies. This works by automatically setting the breakpoints on the checkpoint and restoring the function. When the device is halted, it compares a pre-specified block of non-volatile memory from the checkpoint and from after a restore function.

There is a benefit of using hardware-based testing compared to simulators. In simulations, capturing and simulating the variation a real DUT experiences is difficult. An example of such variation is the boot time.

Chapter 3

Software Frameworks for Intermittently-powered Devices

Intermittent devices store their current state in *Non volatile memory* (NVM) to cope with power failures. Otherwise, their current state is lost after each power interruption. The data is often stored inside the fast NVM, as it is quicker than e.g. flash. Storing the current or restoring the previous state introduces overhead. How much overhead there is, is determined by various factors such as how often the device stores the state, what is exactly stored and how it is restored.

Before looking at the overhead, it is important to investigate which methods are there to save and restore the device's state while experiencing intermittency. The methods affect which kinds of overheads are present and when they occur. Hence in Section 3.1, we describe the state-of-the-art frameworks of intermittent devices. As we eventually want to measure the overhead of those frameworks, Section 3.2 analyses whether the state-of-the-art frameworks can actually be replicated and used to build applications. If intermittent frameworks cannot be replicated, they can also not be used in the evaluation of frameworks in this thesis. Both sections are based on 38 recent papers on intermittent battery-free frameworks.

3.1 Checkpoint or Task-based

While constantly experiencing intermittency, the devices must store and restore their state before and after each reboot to make forward progress. The storing and restoring of the state can be achieved through two methods: checkpointing and task-based frameworks.

Around 50% of the found frameworks make use of a checkpointing-based framework [3, 4, 6–8, 21, 36, 38, 43, 47, 48, 52, 63, 65]. In this approach, programmers, or even the compiler, insert checkpoints into the code at strategic points. During

a checkpoint, the device's state is saved. After a reboot, the device can restore its state to the last successful checkpoint. This approach simplifies adaptation as there is no need to rewrite the program in a specific manner.

However, one must be cautious of potential downsides. If manual placement of checkpoints is needed, it could be easy to introduce a *write-after-read* error. Such an error is caused by rebooting after writing a variable that a following instruction depends on and can cause non-deterministic behaviour in the system. Furthermore, the checkpoint approach also introduces overhead each time the program checkpoints. To minimize the overhead, some frameworks choose a *Just-In-Time* (JIT) checkpoint approach [36, 52]. Instead of using periodic checkpoints, these frameworks use a voltage-based interrupt. It will start the checkpoint procedure only when the voltage falls below a certain level.

The other method of storing and restoring the state, the task-based frameworks, make up the remaining 50% [2, 5, 10, 12, 15, 23, 32, 34, 35, 41, 42, 44, 46, 64]. The main idea behind this approach is that all code is transformed into idempotent tasks. The downside of this approach is that every code needs to be transformed. Also, there is a significant overhead each time tasks are switched.

3.2 Replicability of Intermittent Frameworks

Replicability is an essential value of science, allowing others to validate work and build on it. Replicability is also an important underlying assumption to answer the research question because we must replicate frameworks to evaluate them. In the field of intermittent systems, research artifacts can be included with papers submission. Research artifacts are digital objects that are either used in the study or generated as a result of the study. They add significant value to the paper by allowing external parties to validate the experiments and conclusions presented.

However, the submission of artifacts varies for papers with intermittent frameworks, with only 12% of them providing artifacts [18,23,38,39,65]. Nonetheless, even among papers without artifacts, 62% still made the source code publicly available, e.g. through GitHub. Although there are not strictly speaking artifacts, this thesis considers them unreviewed non-paper artifacts. For the remaining 26% of papers, neither artifacts nor source code was made accessible (e.g. [8, 10, 35]). An overview is provided in Table 3.1. The lack of availability of source code makes it challenging to verify results or build upon the work of others.

Moreover, the quality of the artifacts varies widely. In order to evaluate the actual quality of artifacts, guidelines of ACM and Usenix can be used as starting points. ACM specifically checks five characteristics: documentation, consistency, completeness, exercisability, and inclusion of appropriate evidence of verification and validation [1].

Table 3.1: **The evaluation of the presence of research and non-paper artifacts and the reproducibility without any extra help from the relevant authors of those artifacts. The analysis is based around 38 recent pappers.**

None	Source-code only	Provided	Not able to compile	Compiled
26%	62%	12%	52%	22%

Documentation plays a key role in enabling others to use the software of the authors. One specific form of documentation is usage documentation, which explains how people can use the framework to build their own applications. Only 59% of all artifacts provide explanations on how to compile the given examples. Moreover, 26% of all artifacts actually describe some part of how one can use the framework for their own application.

Before frameworks can be used, they need to be installed or built. This is where dependency documentation becomes essential, as it describes the required software for framework installation and usage. Surprisingly, 44% of the frameworks fail to mention the required software at all. Only 29% mentions all required software with version numbers. Additionally, 7% offers a *Virtual Machine* (VM) or docker image with everything already set up [38, 65].

The absence of a VM or software version numbers presents significant challenges, mainly because many frameworks depend on a specific compiler, such as LLVM. There are many breaking changes between LLVM majors and minors, which increases the necessity for detailed dependency information. Furthermore, certain frameworks (e.g. [42]) also require patches to a specific version of LLVM, meaning it needs to be compiled from the source. This in turn, requires more information about dependencies needed.

Another evaluation criterion is exercisability, which for source code means that the included programs can be run. Unfortunately, several frameworks could not be compiled due to ambiguity and errors originating from the LLVM compiler [12, 39, 41–43, 48, 63]. The completeness criteria, as evaluated by ACM, checks whether all relevant components are available. Some had missing submodules in their repository [46, 49]. These difficulties led to only 30% of all frameworks with an artifact offering enough instructions to compile the provided examples without the help of the relevant authors. Ultimately, only the frameworks of 22% of papers could be recreated (See Table 3.1). This highlights the poor quality of software and science in the field of intermittent battery-free frameworks.

Chapter 4

Design Methodology

For us to address the research question, we first need to define further performance in the context of intermittent systems. One interpretation is the ability to accomplish the maximum amount of work or code in the shortest possible time. Ideally, an intermittent program should complete its execution as efficiently as an program designed for a non-intermittent system. However, this is impossible due to the time required for saving and restoring the state. Thus, performance can be evaluated by assessing the overhead caused by the framework.

Another interpretation of performance in intermittent systems is the ability to continue execution with the smallest capacitor or smallest maximum available energy. While related and correlated to the overhead interpretation, this perspective also considers energy requirements from peripherals, placement of checkpoints, and overhead of starting up the device. The smaller the energy needed to progress forward, the higher the performance.

Crucially, both interpretations of performance require that the saving and storing of the state are executed correctly, which can be assessed through an automatic memory restoration verification. To measure the overhead performance, profiling functionality is introduced to the hardware debugger and implemented as a software test, as described in Section 4.1. Determining the minimum energy performance is achieved through the minimum energy budget finder, as described in Section 4.2.

As 74.4% of the frameworks are based on MSP430, support for the chipset is crucial to debug and compare most frameworks, based on the papers found in Chapter 3. Unfortunately, the existing debugger DIPS lacks support for this chipset. Therefore, MSP430 support must be added to support 65% of all intermittent frameworks. MSP430 support will be discussed in Section 4.3.

We provide a complete overview of DIPS+ in Fig. 4.1. DIPS+ exists out of three parts. The first part is the DIPS+ console which provides software tests. It communicates through GDB protocol with the hardware debugger based on a *Black Magic Probe* (BMP) [9]. The debugger also communicates with the Energy Emulator and can debug the DUT. The Energy Emulator can emulate the power for the DUT and can communicate with the DIPS+ console.

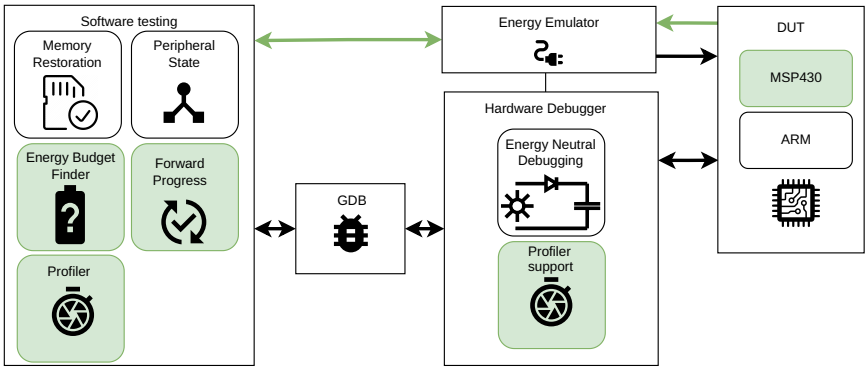


Figure 4.1: **Overview of DIPS+ platform features developed as part of this thesis, in relation to existing debugger DIPS [19]; Light green color denotes specific contributions on op of DIPS.**

4.1 Profiler

Sometimes precise timing is required to measure the characteristics of a program. This is especially the case when measuring the overheads of frameworks. Using breakpoints and measuring time works well until a resolution is needed which is higher than the duration of the breakpoint polling. For more accurate timing, we introduce profile mode. A high-level overview of how profile mode operates within the components of DIPS+ is given in Fig. 4.2. We use the profiler mode to measure overhead as defined in Section 4.1.1.

The accuracy required for the profiler, of course, depends on the characteristics to be measured. However, there is an upper limit to the useful accuracy. It is determined by the clock speed of the DUT, as we are only interested in the cycle count. As the MSP430's CPU runs on only a maximum of 8 MHz and Cortex-M0 below 120 MHz, it is not necessary to have an accuracy greater than $t_{\text{clock cycle}} = 1/f_{\text{cpu}} = 1/120\text{MHz} = 8.3\text{ns}$. For maximum versatility, the profiler should also be able to measure the complete runtime of benchmarks, which can take multiple seconds.

The profiler requires direct control over the emulator. This is necessary as it can automatise specific behaviour, such as measuring startup time after a power outage. The DUT directly controls the timer through an IO pin to indicate when the timer has to start and when to stop. Keeping track of time is done inside the debugger rather than in the emulator or computer to minimise measurement latency. Also, basic statistics like the average of each timer duration are recorded. The profiler can request the timer information from the debugger for further analysis.

The DUT can execute some parts of its program while establishing the debug connection after intermittency; see Section 5.3.4 for more details. If ignored, this phenomenon could wrongly affect timing, as starting and stopping the timer is unjustly called. To ensure that the debugger does not execute code before an active debug connection, an IO pin lets the DUT know it can execute code.

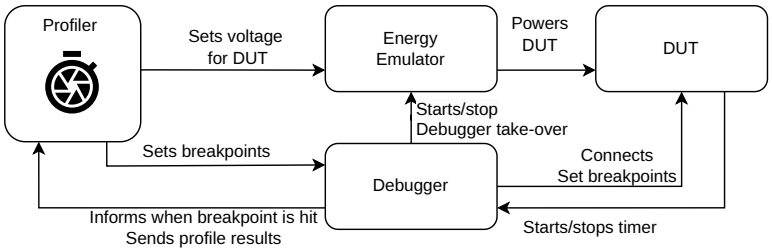


Figure 4.2: **Overview of the interaction between the components used by the automatic profiler of DIPS+.**

4.1.1 Defining Overhead

Frameworks for intermittent devices inevitably cause overhead as the state of the device needs to be stored to the NVM. The overhead is unevenly spread throughout the time the DUT is powered. While turning on, computing, and shutting off, the device encounters several phases with its own overhead, listed below. However, not all frameworks encounter all types of overhead.

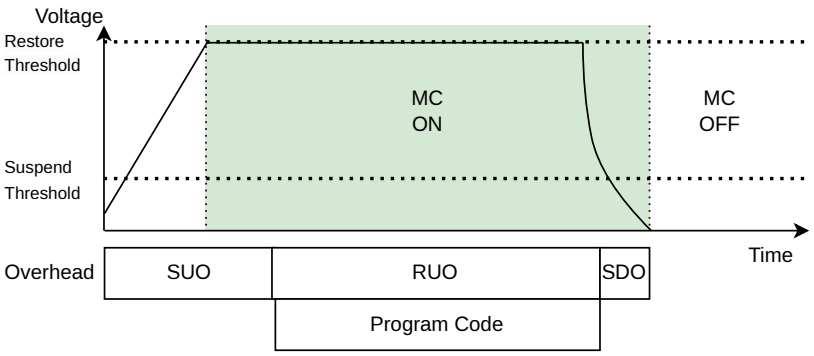
Boot time: The DUT boots by powering on the CPU and initialising registers. The precise time is random and varies depending on factors like device temperature. The randomness is caused by the boot ROM in the MCU. It is, however, independent of the framework used.

***Startup Overhead* (SUO):** This is all the extra code that is executed setting up and loading the framework. It starts from the first line of code until the first line of code of the benchmark itself. This overhead can also vary depending on the state of the device itself, i.e. a fresh boot might be faster as the DUT does not need to restore memory. Moreover, this also includes the time waiting to reach a restore voltage threshold.

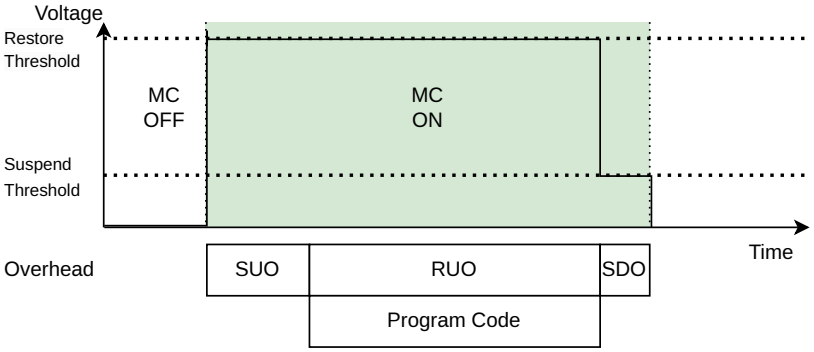
***Runtime Overhead* (RTO):** This is all the overhead that is present while executing the benchmark. It includes the usage of FRAM instead of volatile memory or hidden extra operations caused by the compiler. Additionally, all the checkpoint or task switching done while executing the benchmark is counted towards the runtime overhead.

***Shutdown Overhead* (SDO):** For JIT frameworks, there is also a shutting down overhead. This is the time the DUT uses to save their state.

Accurately knowing when SUO starts and SDO stops requires knowing at which voltage the CPU turns off and on. This voltage differs per MCU and introduces uncertainty. Furthermore, a part of SUO is waiting on a voltage threshold of the power supply being exceeded to restore the state of the DUT. The different voltage thresholds also can cause unfair comparisons of intermittent frameworks, as the SUO depends on how quickly the voltage threshold is exceeded. To avoid these uncertainties and unfair comparison associated with the transient voltages of the power supply of the DUT, the energy emulator does not emulate a voltage pattern usually experienced by battery-free systems. Instead of ramping and slowly falling the power supply of the DUT, the power supply voltage pattern is idealised with constant voltages. An example of this is given in Fig. 4.3b.



(a) **The overhead of a JIT intermittent framework and a regular voltage pattern.**



(b) **The overhead of a JIT intermittent framework and an idealised voltage pattern for profiling purposes.**

Figure 4.3: **Overhead with voltage pattern, below idealised voltage pattern for profiling purposes. The idealised voltage pattern limits the uncertainty associated with the precise start of *Startup Overhead* (SUO) and end of *Shutdown Overhead* (SDO).**

4.2 Minimum Energy Budget Finder

The minimum energy budget finder is a method to find the minimum energy required for forward progress of the DUT. This test needs to control the level of energy sent to the device through the emulator. An approximation is used because the emulator cannot precisely measure the energy sent to the device. The test itself needs to determine the amount of energy to achieve forward progress. To find the amount of energy, the minimum energy budget finder uses a search algorithm, which will be explained in Section 4.2, to try different energy budgets and search for the minimum energy budget to achieve forward progress. As forward progress needs to be measured, breakpoints are set on key locations at which the DUT's saves and restores its state.

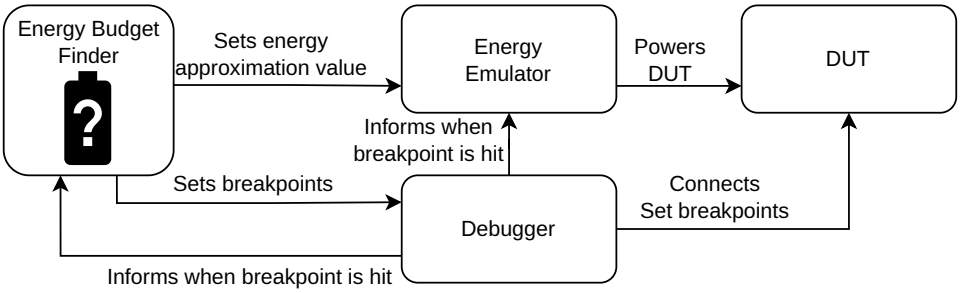


Figure 4.4: **Overview of the interaction between the components used by the energy budget finder test.**

The energy budget finder is a test inside the DIPS+ console on the computer, which hosts multiple automated tests such as the profiler or memory restoration test. The energy emulator can emulate approximately the amount of energy for the DUT specified by the test inside DIPS+ console. How the energy is emulated is discussed in Section 4.2.1. The debugger sets the appropriate breakpoints and informs both the DIPS+ console and the emulator when a breakpoint is hit. An overview of the interaction is given in Fig. 4.4.

4.2.1 Energy Approximation

To simulate an intermittent energy source, the simplest method is to turn the power supply to the DUT on and off according to a predefined pattern. However, this approach only provides a rough approximation of the energy specified by the automated test as it does not consider the current consumption of the MCU. Consequently, devices with different power requirements, such as those with multiple LEDs or none at all, would have the same on/off time. We can use a sampled current to account for the different power usages and obtain an approximated energy for the DUT to achieve forward progress. The relationship between the energy and the voltage and sampled current is defined as $E = u \int i(t)dt \approx u \sum^{t} i_t$, where E is the energy, u is the voltage, $i(t)$ is the continuous current and i_t is the sampled current. The energy approximation calculation is performed for each period that the power supply to the DUT is on. The highest energy budget needed by the DUT of all periods is chosen as the final result, which is the minimum energy budget. It is important to note that this approach assumes a constant voltage throughout the process.

The virtual capacitor, which is already present in DIPS [19], is a more advanced energy approximation tool that runs inside the Energy Emulator. The virtual capacitor simulates a physical capacitor by measuring the current and voltage output and adjusting the output voltage to match the behaviour of a physical capacitor. The energy approximation of the virtual capacitor is defined as $E = \frac{1}{2}C(V^2_{\text{thresH}} - V^2_{\text{threshL}})$, where E is the energy, C is the capacitor of the DUT, V_{thresL} is the voltage threshold that is used to save the state of the DUT and V_{thresH} is the voltage threshold that is used to restore the state of the DUT. By calculating the energy in a capacitor, it is clear that the capacitance correlates directly with the energy. The relationship between capacitance

and energy makes it possible to directly control the approximate energy sent to DUT. By setting the input current parameter to zero of the Energy Emulator's virtual capacitor, we can instantly "charge" the virtual capacitor by updating the voltage to V_{thresH}. By varying the capacitance, we can precisely control the amount of energy being delivered to the DUT.

4.2.2 Search Algorithm

The search algorithm plays a crucial role in controlling the energy released by the emulator and determining the minimum energy required for forward progress. When exceeding the value, it will always have forward progress as long as it does not encounter any bugs. The search algorithm runs on the DIPS+ console, which sets the energy budget on the Energy Emulator of DIPS+. When the energy budget runs out, the Energy Emulator resets automatically after a short delay.

There are several challenges associated with selecting the right algorithm. One significant challenge is the uneven distribution of checkpoints or varying task durations, which may vary in energy requirements. Another limitation is that DUTs cannot be reset easily to their default state due to the persistent nature of the device. Moreover, a program does not always have a defined start and end. Hence, the algorithm used inside the minimum energy budget finder must always consider individual tasks instead of the whole program. This approach is also much quicker as the whole program can have thousands of tasks that must be executed.

Multiple algorithms are available to find the actual minimum energy budget: sequential search, binary search, bisection method, or random search. To illustrate the limitations of these algorithms, all algorithms are discussed through examples. In the examples, we have a task set with five tasks, t_1 to t_5, that will be executed consecutively. All tasks except t_4 take 10 J to complete, while t_4 requires 20 J. The energy budget is determined again after each attempt to execute a task.

A sequential search starts with an energy budget of zero for the DUT and increases linearly until a task of the DUT executes entirely. In the example, t_1 will fail until the Energy Emulator's energy budget set for the DUT is larger than 10 J. As the energy is completely depleted, the Energy Emulator will reset the energy budget to 10 J. After t_1 is passed, t_2 and t_3 will also pass with the energy budget being reset by the Energy Emulator. t_4 will fail until the energy budget is increased to 20 J. t_5 will also pass as it only requires 10 J, which is larger than the set Energy Budget. So the sequential search deals well with varying task sizes, but it takes many attempts to figure out the minimum energy budget. The reason for the many attempts is that the sequential search algorithm only increments the energy budget with the resolution of the energy budget finder after each failed attempt to execute the task. If the resolution of the energy budget finder is very high, e.g. 0.01 J, it can take quite a while to forward progress.

On the other hand, binary search takes a more efficient approach by dividing the set of possible energy budgets that achieve forward progress in half at each step. With an initial low and high energy budget, it calculates the midpoint. If the task succeeds, the high energy budget is set to the current midpoint. If the task fails, the low energy budget is updated. In our example, t_1 might succeed on the first try with 20 J. t_2 will pass with 15 J. t_3 will fail with 7.5 J and then actually succeed with 11.25 J. t_4 will never succeed because the solution is removed from the search space. Hence, this highlights the need for an algorithm that never excludes higher energy budgets. Another algorithm to find minimum energy for the minimum energy budget is the bisection search. Similarly to the binary search, bisection search also tries to reduce the solution space for higher energy budgets. This makes it unsuitable as an algorithm to find the minimum energy budget.

The last possible algorithm to find the minimum energy budget is the random search, which tries random solutions. The problem is that it can mask certain energy requirements. For example, both t_1, t_2, and t_3 will pass the random values 14 J, 15 J and 14 J. t_4 will pass with the random value 22 J. Hence, the algorithm will finish with the wrong conclusion that 14 J is the minimum amount of energy.

Hence, a sequential search is used as it is the only suitable algorithm. This algorithm needs some adjustments, as it is not feasible to determine the energy budget after each executed task. The reason is that there might be some randomness to the boot time. Hence, the DIPS+ Console checks if the time elapsed exceeds the maximum task duration. If this is the case and no forward progress has been made, the energy budget increases. When there is forward progress for at least the minimum test duration, the algorithm finishes. An overview of the algorithm is given in Algorithm 1.

4.3 Debugger

After establishing methods to measure both overhead and energy efficiency, the next challenge was to ensure these methodologies could be applied to the popular MSP430 MCU [58]. MSP430 support ranges from simply reading out registers to debugging while in ultra-low power mode. To do this, we will try to determine a debugger's minimum viable requirements. Table 4.1 gives an overview of the necessary functionalities with the status of their implementation for DIPS+. The architecture used to implement these features will be discussed in Section 4.3.1

One key aspect of a debugger is to read out the current state of the DUT. This requires the DUT to read registers and memory. As it could be beneficial also to alter the state of the device, writing the registers and memory is necessary. The CPU must be controlled to read the state of the DUT. Otherwise, the device keeps changing its memory and registers constantly. Hence, halting and resuming the CPU is a requirement. To halt and resume the CPU more precisely, functions such as breakpoints and step functionality are helpful. These

Algorithm 1: Sequential minimum energy budget search algorithm.

Input:

E_{min}: Initial energy budget
t_{now}: Current time
E: Allowed remaining energy that the DUT can deplete
ΔE: Energy depleted since the last time the variable is used
$t_{\text{task}_{\text{max}}}$: Defined maximum duration of a task
$t_{\text{test}_{\text{min}}}$: Minimum test duration

Output:

E_{budget}: Energy budget

```
▷ All code is executed inside the DIPS+ debugger with two exception
E_budget ← E_min
t_checkpoint = t_now
while true do
    while E > 0 do
        ▷ The next line is executed inside the DIPS+ energy emulator
        E = E − ΔE
        if checkpoint reached then
            t_checkpoint = t_now
        end
    end
    if E == 0 then
        ▷ The next line is executed inside the DIPS+ energy emulator
        Delay
        E ← E_budget
    end
    if t_checkpoint > t_task_max then
        E_budget + +
        t_timeout ← t_now
    end
    if t_timeout > t_test_min then
        exit
    end
end
```

Table 4.1: **Necessary functionalities of MSP430 MCU [58] to be implemented for a fully-operational hardware debugger.**

Command	Attach/ detatch	Register reads/writes	Memory reads/writes	Halt/ resume	Break points
Implemented	Yes	Yes	Yes	Yes	Yes

Command	Stepping	LPM1-LPM4	Watchpoints	LPMX.5
Implemented	Yes	Yes	No	No

functions map the instructions to the source code that the program is written in. Functionalities like watchpoints are also helpful but can easily be replaced by a breakpoint on multiple places. Hence, watchpoints are not implemented here. One prerequisite of a debugger is that it can get the CPU in a state where it can be debugged. This is done through attaching.

Multiple frameworks use the *Low Power Mode* (LPM) of MSP430, for example, waiting for the power supply voltage of the DUT to exceed a certain threshold. There are multiple low-power modes of MSP430 which turn off various components on the MCU. It starts from LPM0, and with LPM1 to LPM5, the LPMs gets progressively more energy efficient. LPM4 turns off the CPU and all clocks, which means only external interrupts can wake the device up. Intermittent frameworks often make use of LPM4 for waiting on a voltage threshold being exceeded (e.g. [36, 52]); hence, it needs to be supported by DIPS+. LPMX.5 turns off the CPU, RAM, JTAG interface, and the *Embedded Emulation Module* (EEM), which contains all the debug logic [58]. On wake-up, the CPU core is reset completely, meaning the debug connection must be reestablished manually. Hence, LPMX.5 is not supported, as the DUT will require a debug reconnection to the DIPS+ hardware debugger.

4.3.1 Architecture of the Debugger of DIPS+

The traditional way of debugging a MSP430 is to use *Code Composer Studio* (CCS), which makes use of a fork of GCC called MSP430-GCC [61]. CCS directly interfaces with the *Texas Instruments* (TI) debug engine, which connects to the MSPFET tool. The MSPFET tool, in turn, communicates with the DUT through JTAG of *Spy-By-Ware* (SBW). However, a significant drawback of this setup is the lack of compatibility with the feature-rich GDB client [27]. There is an alternative method, luckily. When using open-source tools like mspdebug [17] or MSP430GDBProxy [54], these tools act as intermediaries, converting GDB commands from the client into inputs for the proprietary TI Debug Library. This library, in turn, communicates with the MSPFET tool. However, the issue with this approach is that the main debug logic is located inside the PC rather than in the probe. This causes low-level and fine-grained operations, such as reading a specific register, to be executed at the high-level debug tools on the PC. Since the communications speed between the PC and probe is limited, it hinders performance.

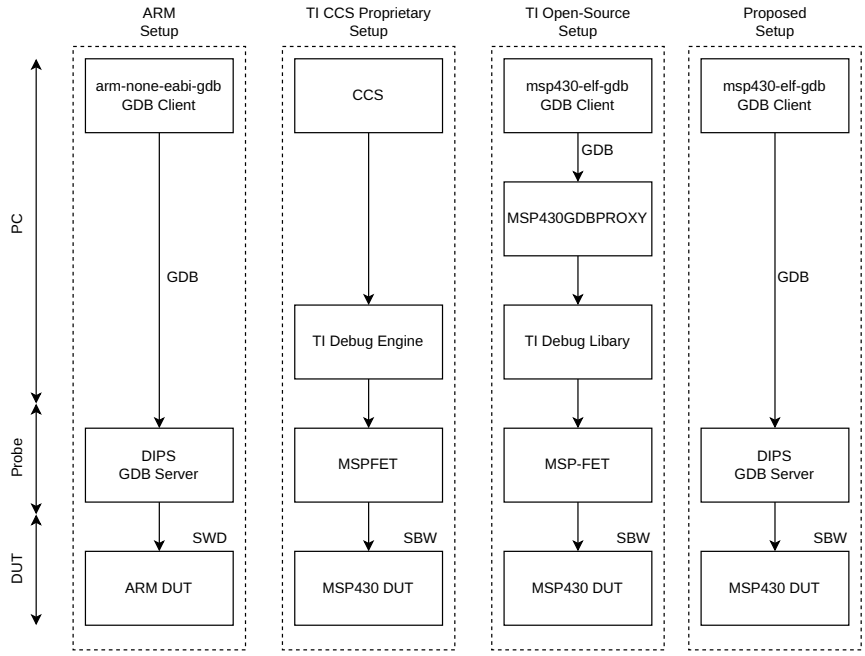


Figure 4.5: **Various approaches on how to implement debugging support for the MSP430. The right-most solution describes the implementation of DIPS+.**

The implemented solution in this thesis is to have DIPS+ receive GDB commands and talk directly to the DUT through SBW. The main benefit of this method is that it is fast and unlocks the already-developed tests. An overview of this approach and the approaches of the alternative solutions are given in Fig. 4.5.

4.4 Testing Existing Frameworks

A framework cannot be directly tested using a debugger, as a framework is only a foundational structure or template to build applications. It cannot be directly compiled into an actual program that can run on a MCU. Rather, only applications can be tested with a debugger at run-time. An application can apply the framework to deal with the intermittent nature. As the usage of applications varies widely in the field of intermittent systems, it is beneficial to use generic applications that represent a similar workload. Hence, benchmarks are used as representative workloads to evaluate and describe the performance of a framework. In order to compare results, it is important to normalise the results to their non-instrumented counterparts.

A selection of benchmarks is required to compare existing frameworks designed for intermittent systems. Among all found frameworks, 44 unique benchmarks were found inside their corresponding paper. On average, each framework used

three benchmarks to assess its capabilities. Important to note is that real-life applications are excluded from this benchmark set. The reason is that they often require specific peripherals, such as an e-ink screen, or specific capabilities from the intermittent system, such as event/interrupt support.

Among the identified benchmarks, the top seven most popular ones, ranked in descending order, are *Rivest-Shamir-Adleman* (RSA) (with a prevalence of 53%), *Activity Recognition* (AR) (50%), *Cuckoo filtering* (CF) (38%), *Bitcount* (BC) (38%), *Cold-chain Equipment Monitoring* (CEM) (26%), *Cyclic Redundancy Check* (CRC) (23%), and *Blowfish* (BF) (19%).

RSA, an encryption algorithm, uses a fixed key and fixed plain text to encrypt data. It primarily uses modular exponential operations and modular multiplications. The algorithm is an older encryption algorithm that can be used by battery-free devices to encrypt data. **CF** stores pseudo-random numbers and then performs filtering the recover the sequence. This benchmark would be representative of some signal-processing applications. **BC** benchmark focuses on counting the number of set bits inside a random variable with seven different algorithms One of the algorithms also covers recursion, which is not always supported for intermittent frameworks (e.g. [32]). The benchmark does not make many memory calls (such as load and store operations) [30]. This makes it a good benchmark to capture overhead from tasks/checkpoints, rather than variables that need to be stored or loaded to NVM. **CEM** stores and logs data from a sensor or pseudo-random number generator. These values are also losslessly compressed. **CRC** is an error-detection algorithm commonly used in data communication **BF** is an encryption algorithm where a string and key are defined. It is a secure encryption algorithm, which makes it possible that batter-free devices will use it to encrypt data.

The combination of benchmarks is necessary as it is not known beforehand what kind of problem the eventual application will solve. These benchmarks cover 4 out of 6 application types [30]: automotive, industrial, network, security, and telecommunication. Consumer and office applications, which focus on text multimedia or multimedia, are not included. Hence, for the evaluation in this thesis, the combination of RSA, CF, BC, CEM, CRC, andBF is used.

Chapter 5

Implementation

In this chapter, we focus on the technical implementations of the ideas discussed in Chapter 4. We first describe the profiler, followed by the minimum energy budget finder, the challenges of porting and details about the testing procedure, and finally the implementation of MSP430 for the support is discussed. We also highlight the changes done to the recommended/reference code to increase the performance of the debugger.

5.1 Profile Mode

The profile mode introduced in Section 4.1 needs an accurate timer in order to measure overhead of frameworks. The DIPS+ debugger is based around the STM32F103RET6 chip, which has multiple 16-bit timers and a maximum clock speed of 72 MHz [53]. A 16-bit timer could have a maximum resolution of 13.8 ns, with the longest interval being 910 µs. If the longest interval is 1 s, the resolution would be 15.2 µs. Both options do not meet the design criteria specified in Section 4.1. Hence, two 16-bit timers are used to obtain a resolution of 910 µs and the longest interval of 59.6 s. The timers are coupled through a gated mode, which means that one timer acts as the input of the next timer (Fig. 5.1). Other approaches would introduce significant overhead, such as using an interrupt to handle the overflow of the fastest timer and writing to a variable. For approaches that use an interrupt to handle the overflow, the context needs to be switched and interrupt handled every 59.2 µs.

Starting and stopping the timer is controlled by the DUT. The timer can be directly controlled by the timer through three simple macros: *TIMER_START*, *TIMER_STOP*, and *PROFILE_SET_PIN_DIR*. An overview of all available macros is given in Table 5.1. The *TIMER_START* and *TIMER_STOP* macros work by either setting or clearing an IO pin connected to the debugger. The debugger registers these changes through an interrupt and starts or stops timer.

The achievable resolution in practice would be less than the theoretical maximum due to the latency in starting and stopping the timer, which is caused by two factors. Firstly, the debugger must register and process the interrupt caused by the IO pin. In an ideal case, this takes at least 12 cycles [66] and with

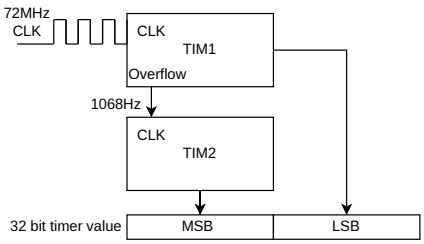


Figure 5.1: **Two 16-bit timers inside the hardware debugger that are linked together to achieve one 32-bit timer. This combined timer is used to profile the DUT. Inside the figure, *CLK* stands for Clock and *TIM* for the different timers, *MSB* as Most Significant Bytes and *LSB* as Least Significant Bytes.**

Table 5.1: **The overview of the C Macros to interact with DIPS+.**

C API	Description
PROFILE_SET_PIN_DIR	Configures the pins of the DUT
PROFILE_WAIT_UNTIL_READY	Wait until the debugger is connected
TIMER_START	Starts the profiler timer of the debugger
TIMER_STOP	Stops the profiler timer of the debugger

a clock speed of 72 MHz for the debugger, this takes at least 166 ns. However, since the debugger's code uses a *Hardware Abstraction Layer* (HAL) library and does not do bare register operations, this is increased even further. The second reason for the error in the profiler is the need to read out the current timer register and write it to a variable. These extra instructions also take at least 13 ns each. As the timer needs to start and stop to register the time, one would expect that the delays mentioned above are cancelled out.

To prevent the debugger from executing code before establishing a debugger connection after intermittency, the DUT waits with *PROFILE_WAIT_UNTIL-_READY* macro. The macro continuously polls the IO pin that indicates that the DUT is fully connected. Before the MSP430 debug connection is fully established, the MCU reboots thrice. By default, the ready signal is given by the debugger after the last reboot. The macro itself is placed after disabling the watchdog and configuring the pins of the profiler.

5.2 Minimum Energy Budget Finder

The minimum energy budget finder directly controls the output voltage of the emulator. Control of the output voltage of the emulator requires that the DIPS+ console can receive and interact with the emulator through the emulator's protobuf [29] interface. Protobuf is a programming language-agnostic data serialisation format to transfer data between software systems. Using the protobuf interface of the emulator, we can enable modes such as the virtual capacitor or a square wave. The interface also allows the DIPS+ console to gather information about the measured voltage and current outputted by the emulator.

The debugger is not able to instantly connect to the device and set the relevant breakpoints. The time spent at connecting and setting breakpoints is the overhead of debugging. The emulator can ignore the overhead's energy usage as long as it knows exactly when the DUT can execute its program code. In order to enable this functionality, we use the macro *PROFILE_WAIT_UNTIL_READY* of the profiler to prevent executing the code before actually being able to check if breakpoints are reached. Otherwise, forward progress might be unjustly ignored. The second, and optional, feature is the ability to let the DUT communicate to the emulator through an IO pin. With this extra information, energy spent while the DUT cannot execute code is ignored. This increases precision at the cost of requiring code changes to the DUT.

5.3 Debugger

In DIPS+ MSP430 support is added to the open-source debugger BMP [9]. Since MSP430 uses an entirely different architecture, it is also essential to describe the differences and limitations of debugging.

To describe the changes needed for MSP430 support in DIPS+, it is useful to shortly discuss the current architecture of BMP. The components of the debugger can be split into four distinct components: main debug logic, device scanner, *Device Abstraction Layer* (DAL), and *Protocol Translation Layer* (PTL) (Fig. 5.2). The main debug logic parses GDB commands and manages all high-level functionalities. This component can also search for DUTs through the device scanner. It can also interact with the DAL through basic operations such as writing registers or setting breakpoints. The scanning and device layer sends device-specific instructions and data to the PTL. This layer transforms the instructions into binary data and sends it at an appropriate speed to the DUT. This layer also transforms the response of the DUT back and sends it to the device scanner and DAL. To fully offer support of MSP430, major changes are needed to the device scanner, DAL, and the PTL, which we describe below.

5.3.1 PTL: Protocol Translation Layer

The PTL converts device-specific instructions into a binary stream according to a certain protocol. ARM uses either JTAG directly or *Serial Wire Debug* (SWD) through a proprietary 10-pin connector. DIPS+ has a 10-pin connector to connect to ARM devices easily. However, to add MSP430 support, the protocol SBW needs to be supported. As SBW only uses four pins and an adapter is required (See Fig. 5.3).

Not only is an adapter needed, but actual support for the SBW is needed. The implementation of this protocol is based on the specifications of TI [59] and a reference application [56]. The specifications, however, mention a clockspeed frequency of 18 MHz. Sending instructions at that clockspeed does not work reliably as instructions are sporadically ignored or return inaccurate information. When using a clock frequency of 500 kHz, as obtained from the reference application, SBW works reliably.

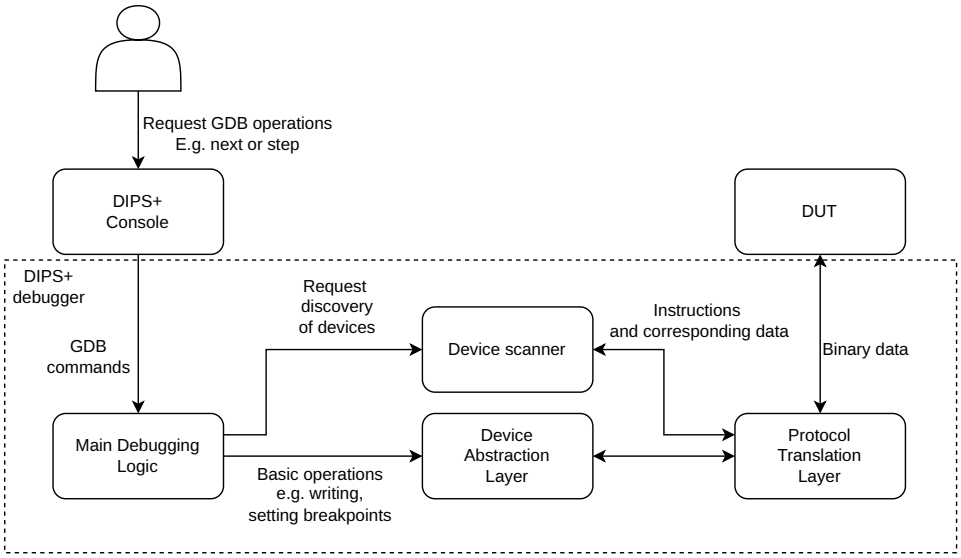


Figure 5.2: **High level-overview of the components of the DIPS+ debugger and the typical interactions that the components have with each other. The components of the DIPS+ debugger are already present in BMP [9], however, changes were applied to all components to support MSP430.**

5.3.2 DAL: Device Abstraction Layer

The DAL transforms requests from the main debug logic to device-specific instructions. Luckily, some reference code for simple operations exists, provided by SBW specification [59] or the MSPDebugStack [62]. For more complex operations, such as settings breakpoints or halting the DUT, MSPDebugStack offers less guidance as the codebase is very complex and poorly documented. It is unclear when and how the TI Debug Engine calls which function precisely. By reverse engineering the MSPFET, we were able to add DAL support for the MSP430. The functionalities of the debugger can be split up into four parts: basic connectivity, reading and writing memory and registers, continuing and halting, and breakpoints. Each part is discussed separately.

Using the reference code [56], we were able to put the MCU in debug mode. After successfully establishing the basic connectivity, the next step involved rewriting functions for memory and register read/write operations. To ensure accuracy and validate the functionality, traces of communication between the probe and DUT were recorded and compared to a MSPFET with CCS. This comparison enabled us to verify and adjust timing and delays within the functions.

The resuming and halting capabilities were implemented once the essential read and write functionalities functioned correctly. At this stage, the MSPDebugStack becomes significantly less helpful. The MSPDebugStack streams specific values and addresses from the TI Debug Engine during runtime. Such beha-

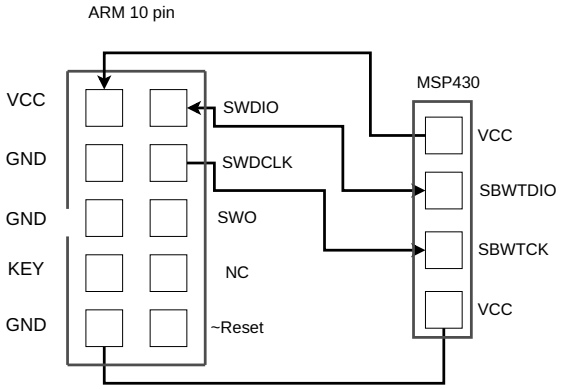


Figure 5.3: **Interface how to convert a 10-pin SWDP connector to a 4-pin SBW connector.**

viour is not visible at all using static code analysis alone. As a result, a lot more needed to be reverse-engineered. To verify the functionality, the debug connections underwent hot-swapping. Hot-swapping was done by starting the debug process in MSPFET with CCS and moving the DUT in a specific state, e.g. stopped in a breakpoint. Afterwards, the data lines of the MSPFET were swapped to those of the DIPS+. This allows for easy verification of individual functions.

The final and most challenging phase is implementing setting and clearing breakpoints. The features are barely mentioned at all in the documentation. For instance, the user manual spends only six pages describing the features that complex EEM has [57]. The EEM controls all crucial topics related to the debugger, such as breakpoints and clock control. One would expect much more documentation on how to use it. Moreover, there is no information on how to set the breakpoints. This required significant reverse-engineering by analysing prerecorded traces of setting and removing breakpoints through CCS using MSPFET.

During the validation of the breakpoints, an interesting quirk was found regarding breakpoints and intermittency. When the GDB function *break* is called, a breakpoint is not immediately set. Instead, the breakpoint is stored within the debugger and the GDB client. It is only actually set after a *continue* command. On a breakpoint or interrupt, all breakpoints are cleared from the memory of the DUT. Additionally, setting a breakpoint on the exact address where the program is currently halted is impossible. Otherwise, it might lead to race conditions where DUT continuously halts on the same breakpoint without executing other code. Instead, GDB sets all other breakpoints, performs a step, and then sets the remaining breakpoint (see Fig. 5.4 for a flowchart). However, this process can fail if there is a power interrupt during the step, preventing the remaining breakpoint from being set. On boot, DIPS+ only restores the remaining checkpoints. Hence, the device must be halted to ensure that GDB sets the breakpoints as expected.

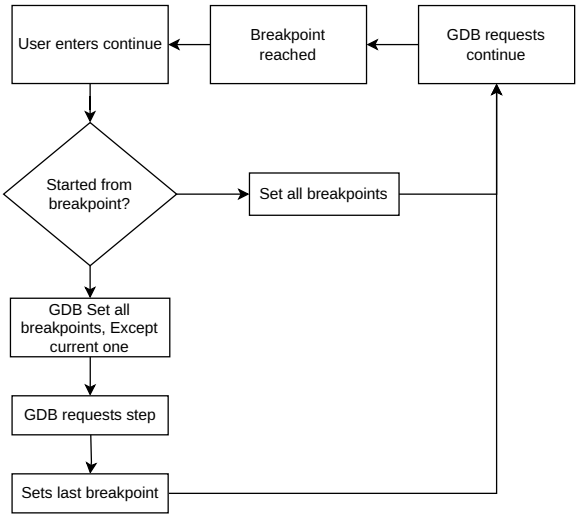


Figure 5.4: **An overview of the interaction and the procedure of the GDB client used in combination with an MSP430, called msp430-elf-gdb, to set a breakpoint on a DUT.**

5.3.3 Intermittency Support

As DIPS+ is a debugger for intermittent systems, it also needs specific support for the intermittent behaviour of the MSP430 DUT. Luckily, the code for intermittency support is mainly inside the main debugging logic, which is mostly platform-agnostic. Hence, adding intermittency support is trivial for MSP430, besides a few bug fixes.

A significant difference between MSP430 and ARM is that ARM is always entirely in control. That is, after intermittency occurs, the CPU of the ARM chip will not execute any code of the application. This is unlike the MSP430, which requires the connection to be reestablished manually. It sounds minor, but it has a significant impact when used for profiling code.

5.3.4 DIPS+ Optimizations

The basic variant of the debugger of DIPS+ supports all features specified in 4.3. However, there is room for improvement as the (re)connecting the DUT to DIPS+ takes around 1.2 s. Another bottleneck is the amount of code executed while reconnecting the debugger to the DUT. Four changes, described below, were performed to the debugger to increase performance.

Architecture awareness: The first change is to make the main debug logic of DIPS+'s debugger architecture-aware. Doing a SWD scan for MSP430 and vice versa makes no sense. This sped up the reconnecting process significantly. The initial scan for DUTs still needs to do all kinds of scans, as the debugger is unaware of which architecture is being attached.

Remove unnecessary functionality and support: Certain features of the debugger are unnecessary and outside the debugger's scope of normal operation,

as defined in Section 4.3. One of those features is support for attaching to a MSP430 in LPM5 mode. Another example is a backup method for connecting the debugger to the DUT, where you can connect by sending a specific code sequence called a magic pattern. Both these features are in the critical path for reconnecting to the DUT and are time-consuming. By removing these features, the reconnect time can be decreased significantly.

Adjust and shorten delays: Delays used in the reference code or specifications are sometimes quite long. The exact delays are now experimentally determined to reduce connection overhead.

More precise control of emulator supply: Although not directly related to the performance, errors were caused by letting the emulator know too late that the device was inside a breakpoint. The previous approach waited before detecting and fetching the state was finished, after which the debugger let the emulator know it was inside a breakpoint. We improved this by letting the emulator know as soon as possible.

5.4 Testing Existing Frameworks

Certain benchmarks we selected for the evaluation, such as CEM or BF, pose specific challenges due to their requirement for large arrays or variables. On embedded systems, like the MSP430, the RAM is limited and these benchmarks exceed the available memory. Luckily, all frameworks extensively use the NVM. Hence, it can be argued that the non-instrumented code also uses the NVM for certain benchmarks. Important to note that the default approach is that variables are placed inside the RAM until it does not fit anymore.

The experiments will be conducted at a clock frequency of 1 MHz, which allows for a precise measurement of code duration compared to the default 8 MHz. Furthermore, operating at this frequency eliminates the need for wait states in the FRAM.

To test the frameworks that are compile-able, the frameworks need to be applied to each benchmark selected in Section 4.4. Luckily, some frameworks have already applied their methods to a benchmark. For example, bitcount did not need to be ported to any framework. However, BF must be ported to InK, QuickRecall, AllocatedState, and ManagedState. Similarly, both RSA and CEM needed to be ported to InK, QuickRecall, AllocatedState, and ManagedState. The CF only needed to be ported to InK. Additionally, the benchmarks must also need to have an uninstrumented version. As QuickRecall, AllocatedState, and ManagedState are all JIT checkpointing frameworks, it is as easy as copying uninstrumented code.

Chapter 6

DIPS+ Evaluation

In this chapter, we evaluate the performance of DIPS+ and its associated automated tests. Instead of directly using a benchmark, our evaluation is centred around a simple program. The pseudocode of this program is shown in Code 6.1. The program boots inside *main()* with initialising the ports, enabling pins necessary for DIPS+. The program continues inside a while loop. Inside this loop, the program will reach two dummy functions: restore and checkpoint. The checkpoint and restore function toggle dedicated IO pins in order to be able to analyse the behaviour with a logic analyzer. It also has a wait loop where an IO pin called work is toggled continuously. This pin indicates that the program is executing code. The logic analyzer is a Saleae Logic 8 [50].

6.1 Debugger

The basic functionalities are implemented and tested using the dummy program (Code 6.1). By setting breaking points, stepping, reading registers etc., we verified that all the functionalities, defined in Section 4.3, are working correctly. The performance and effect of the changes defined in Section 5.3.4 are discussed in section 6.1.1 and 6.1.2.

6.1.1 Connection Speed

As discussed in Section 5.3.3, the debugger does not always control the MSP430 MCU. The debugger must reestablish the connection each time the DUT experiences intermittency. This connection process involves three main phases: enabling JTAG access, getting the CPU under JTAG control, and disabling the *Memory Protection Unit* (MPU). A reset is executed after each phase before achieving full connection. During reestablishing the debug connection, it is impossible to halt the execution for breakpoints as they have not been set. This means that user code or work is executed before being fully connected.

The amount of user code executed before achieving full control can be measured using a logic analyzer to analyze the work IO pin. The IO pin work is described in Code 6.1. The actual connection time is measured from when the power crosses the CPU-on threshold until full control is established, including

Listing 6.1: **Dummy program for DIPS+ evaluation**

```
main(){
    initPorts();      // Initalises the ports and pins used by DIPS+
    enableAllPins(); // Sets all initalised pins to high
    toggleBootPin(); // Toggle the IO pin assoicated with boot

    while(){
        // Dummy function that restores the state
        restore();

        for(int i = 0; i< 1000; i++){
            delay();

            // Toggle the IO pin assoicated with work
            toggleWorkPin();
        }

        // Dummy function that saves the state
        checkpoint();
    }
}
```

the final toggle of the IO boot pin. Before implementing the proposed changes suggested in Section 5.3.4, reconnecting took nearly 1200 ms with DIPS before being halted and achieving complete control. Additionally, almost over 1000 ms of code execution occurred before the debugger was connected fully. This defeats the purpose of the intermittent debugger, as most code can be executed before being connected.

Our changes reduced the connection time to under 175 ms, which is **11 times faster** than the DIPS+ without the improvements to the debugger. However, this is around a factor of 2 slower than the reconnection time that DIPS achieved for ARM chips [33]. Notably, there is no difference in connection time between initial connections and reconnections for the MSP430, making the process quicker for initial connections. Using the logic analyzer on the work IO pin of the dummy program, it is determined that 7.6 ms of user code is performed before being fully connected. Even though **157 times less** user code is executed, it cannot be reduced any further as it is a limitation of the MSP framework.

6.1.2 Control of the Emulator

In our design, the debugger informs the emulator when the code of the DUT is not executed. There are three types of non-executing code: initial connection, reestablishing the connection, and during a breakpoint. To quantify the effectiveness of the changes proposed in Section 5.3.4, we compare DIPS+ with and without the changes.

During the connection and reestablishing the connection phase, the debugger connects to the DUT and by the DUT actually executes user code until the connection process is fully finished. For initial connects, no optimisation is done and it does not relay to the emulator that code is not executed. The reason is

that it simply is irrelevant, as connecting to a DUT is a prerequisite to start an automated test in the console. The reconnection time is, however, necessary as ignoring it would give an inaccuracy of 157 µs in the duration that the DUT is performing program code after each power outage. For example, this inaccuracy causes the minimum energy budget finder to always overestimate the required minimum energy to maintain forward progress. With adjustments in the DAL, the debugger can now detect and relay in 8.0 ms that the debugger is reestablishing the connection, which is **18 times faster** compared to DIPS+ without the improvements to the debugger.

Accurately determining when the DUT is inside a breakpoint requires: detecting when the DUT leaves the breakpoint and when it encounters a breakpoint. As the debugger instructs to DUT to leave the breakpoint and the device will continue instantly after the command, the breakpoint exit can be detected and relayed to the emulator within 4.2 µs. Detecting and relaying the encounter of a breakpoint is slower as the debugger has to poll the DUT whether there is a breakpoint. With the default debugger implementation, it takes around on average 10.7 ms to relay that the DUT is halted. By increasing the interval between polls of the debugger to the DUT, it takes on average 2.1 ms to relay a breakpoint to the emulator. This is **4 times faster** compared to DIPS+ without the improvements to the debugger. This reduction becomes even more significant when considering that the step function, called after each GDB continue command when there are multiple breakpoints, is also just a breakpoint with the breakpoint address being on the next line. Increasing the breakpoint polling interval is unfortunately impossible with the current clock frequency used by SBW, as the interval cannot be shorter than the polling instruction. A higher breakpoint could be possible by using a different SBW clock speed for that specific operation, reducing the duration of the polling instructions.

6.2 Profiling

Profiling can measure the time of the DUT's program fragments. This should happen as accurately as possible. Rather than measuring the overhead of the dummy program, the delay time is measured inside the program. This is done by adding the relevant macros inside the dummy program around the delay function. By measuring the communication pin between the debugger and DUT, the logic analyser can measure with an accuracy of 4 ns [50].

The mean average error between the logic analyser and DIPS+ is around 113 ns. Hence, the theoretical accuracy calculated in Section 5.1 is far from achieved. When 95% accuracy is required, the accuracy is around 2.26 µs. This is still **928 times more accurate** than without the profile mode. This accuracy represents around 18 instructions on the MSP430 and 162 instructions on the ARM cortex M0, assuming one cycle per instruction.

The profile mode also enables faster communication of the device being inside a breakpoint to the emulator. The time that the debugger, with implemented changes, takes to relay a breakpoint hit takes on average 2.1 ms (See Section 6.1.2). With the profile mode, the relay delay is only 1.6 µs as it is not

polling-based anymore. This is **1311 times faster** than using the regular mode on DIPS+. While using the profile mode does not affect the connection time, no new code is executed until the connection is fully established, as expected.

It is important to consider the impact of the profiling macros on the performance of the DUT. To assess the impact of these macros, IO pins are toggled before and after each individual macro, and the time between these toggles is measured using a logic analyzer. The results are both 3321 ns for *TIMER_START* as well as for *TIMER_STOP*. In reality, the overhead will be even lower because IO pins used for measuring are not set atomically. Nevertheless, this concludes that the profile macros' overhead is insignificant in a realistic application/program.

6.3 Minimum Energy Budget Finder

The minimum energy budget finder is a tool to find the minimum energy budget, regardless of how the tasks are ordered or how much energy they require. It can operate in regular mode or, with code changes of the DUT, in profile mode.

The accuracy of the minimum energy budget finder with the sampled current energy approximation, defined in Section 4.2.1, can be determined by letting the tool find a known duration task. This known duration task can be specified in the dummy Code 6.1. This is done for small tasks of 15 ms to larger task to 1000 ms. As the impact of an error increases as the task duration becomes smaller, it makes sense to talk about the relative error. This normalises the error according to the true pre-programmed task size.

The energy budget finder in regular mode shows a relative error below 5% for task durations of 250 ms to 1000 ms (Fig. 6.1). The relative error rises rapidly when the task duration is smaller than 60 ms. For 10 ms, the relative error even approaches 50%. A task of 64 µs seems to be more accurate than a task of 124 µs. The difference in relative error could be attributed to a small outlier measurement. The effect of the outlier measurement is enlarged as the task duration is smaller; hence, the relative error increases.

The profile mode of the energy budget finder reduces the relative error significantly (Fig. 6.1). For the same range of task durations, the relative error is halved. The sub-5% relative error can be achieved for five times smaller tasks than the regular mode. However, the same trend occurs where the relative error increases as the task duration becomes shorter. For a task of 10 ms, the relative error is 24%.

Diving deeper into the sources of these errors, the DUT and measuring circuit acts like a capacitor. Additionally, the voltage signals of the debugger signals charge the device. These phenomenons require that the emulator fully charges and discharges the DUT before the CPU turns on or off. This affects the timing of the emulator and minimum energy budget finder. The source of this error can be primarily explained by the lack of IO pin's driving power.

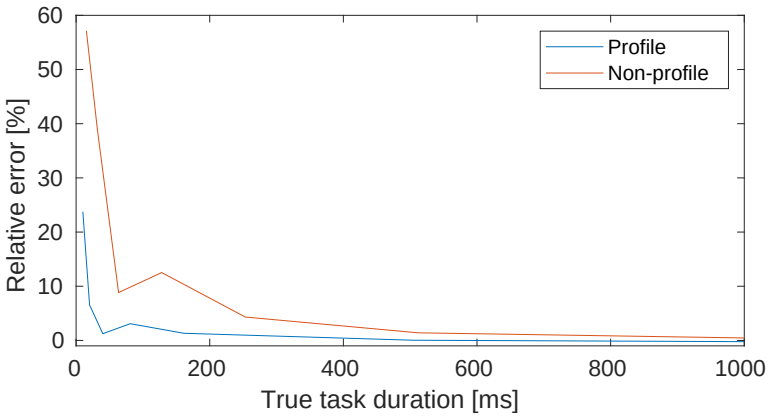


Figure 6.1: **Relative error in measuring the minimum energy budget for a range of task durations. The energy approximation used by the minimum energy budget finder is the sampled current approximation as described in Section 4.2.1. Both the regular and profile mode are used.**

Chapter 7

Evaluation of Frameworks for Intermittently-Powered Systems Using DIPS+

In Section 3.2, we state that most frameworks developed in literature are not replicable. From those that were re-plicable, we selected the following frameworks for further evaluation: Alpaca [42], Chain [12], Dino [41], Ink [64], QuickRecall [36], AllocatedState [52], ManagedState [52]. Using the benchmarks chosen in Section 4.4, the three categories of overhead, as defined in Section 4.1, are measured. These measurements give an indication of the efficiency of the framework.

7.1 Performance Measurement Using DIPS+

7.1.1 Runtime Overhead

Below are the RTOs of each benchmark discussed and analysed. Suppose a paper of a framework mentioned a framework with either a normalised RTO or the execution time of the framework combined with an absolute time of the uninstrumented code. In that case, it is included in the analysis. Unfortunately, the relative comparisons of frameworks to only the performance of other frameworks, often mentioned in intermittent papers as stated in Section 1.1, cannot be used in the analysis.

Bitcount: The RTO causes that BC takes around 1.5 to 11 times longer than the instrumented code (Fig. 7.1a). It is clear that JIT frameworks perform significantly better as the overhead is limited to 1.05 and 2 times longer compared to the uninstrumented code. The reason there is any overhead, even though the frameworks only execute code, is explained by the fact that every variable is stored in FRAM. This is fast but not as fast as the volatile memory inside the CPU. The difference between QuickRecall and ManagedState or AllocatedState cannot be explained. For task-based systems, InK performs the best but is closely followed by Dino and Alpaca. Chain takes almost ten times as much as the uninstrumented code.

The findings of Alpaca, Chain and Dino line up with the RTO mentioned in the paper of Alpaca [42]. However, the measured overhead is consistently slightly larger for all benchmarks than the mentioned overhead. This could be explained by the fact that the paper of Alpaca uses an internal timer of the MSP430 DUT, which is less accurate than the timer used by the profiler of DIPS+. Moreover, when the internal timer of DUT overflows, an interrupt routine resets the timer and increments an overflow variable. Handling this interrupt and writing the overflow variable does not happen instantly and can cause inaccuracies in measuring the RTO.

Blowfish: The RTO of BF benchmark causes the runtime takes around 30% to 11052% times longer (Fig. 7.1b). Surprisingly, Alpaca is the most efficient algorithm, even though it is a task-based system. One reason is that alpaca might very efficiently select which variables are placed inside the FRAM and which ones are not. This is more efficiently done than the uninstrumented code. The JIT frameworks share second place at a 50% increase in runtime. Chain takes 110 times (!) longer to complete than the uninstrumented code. That Chains' overhead is larger than Dino or Alpaca is also mentioned in the paper of Alpaca [42]; however, the measured overhead is ten times larger than the mentioned overhead. This discrepancy between the measured result and the result of Alpacas' paper [42] for Chain cannot be explained and requires further research.

CEM: Frameworks take between 3% and 8000% longer for the CEM benchmark. JIT systems have a very little RTO. AllocatedState and ManagedState are the most efficient as they only take .8% longer, as opposed to the 1.6% for QuickRecall. The high efficiency can be explained by the fact that the dictionary variable, which is often called, is also stored in FRAM for the uninstrumented version. Again, Alpaca scores the best of the task-based systems with an increase of 600%. Other frameworks take around 20-87 times longer than the instrumented code. Dino has a measured overhead roughly twice the size of the mentioned overhead of Dino in the Alpaca paper [42].

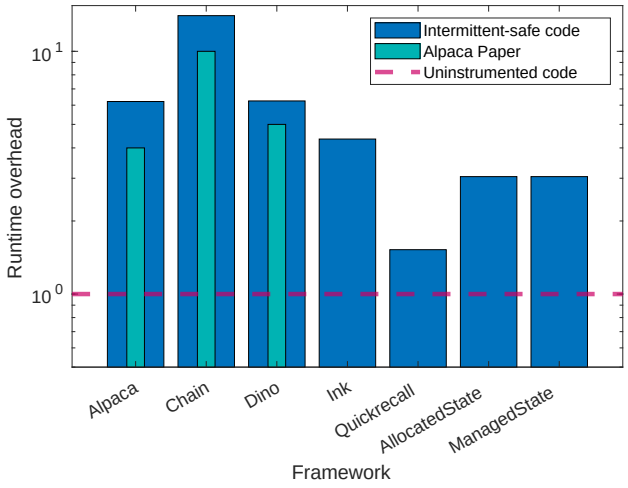
Cuckoo: The impact of RTO is that it takes between 3% and 1264% when testing on the CF benchmark. Even though the uninstrumented code does not store any variables in the FRAM, it got similar performance for the JIT frameworks. Hence, FRAM usages cannot explain this difference. The task-based frameworks show a similar trend to those of CEM: Alpaca is the most efficient, and other implementations take around 23 to 87 times longer than the uninstrumented version.

RSA: The effect of RTO causes the RSA benchmark to be around 8% to 2323% slower executed for frameworks compared to the uninstrumented counterpart. JIT based systems take around 8% longer. This is in line with the other benchmarks where the uninstrumented code uses FRAM. Alpaca and Ink are quite efficient for a task-based system with just 60%-100% overhead. Other systems take around 7 to 20 times longer. The measured overhead of QuickRecall is roughly the same as the overhead mentioned in the paper of QuickRecall [36]. Furthermore, the findings of Alpaca, Chain and Dino line up with the RTO mentioned in the paper of Alpaca [42].

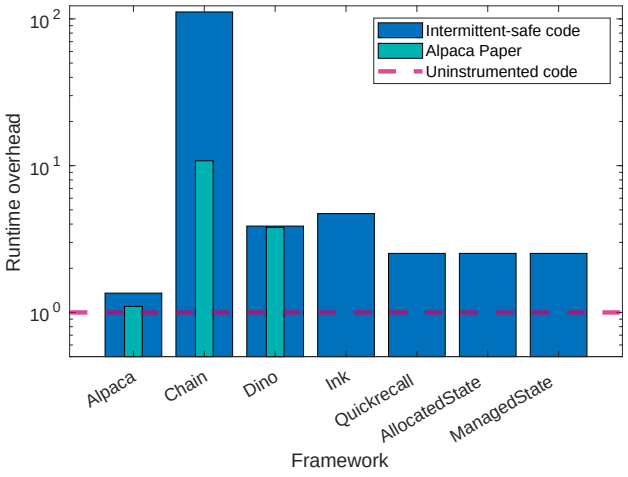
7.1.2 Startup Overhead and Shutdown Overhead

The startup varies widely, from an average from 102 µs to an average from 500 µs (Fig. 7.2a). The trend is that task-based systems are more efficient than JIT frameworks. A notable exception is InK, which takes the longest startup time. This can be explained by the fact that InK has a complex scheduler that even supports threads. Of the JIT frameworks, the order of shortest SUO is QuickRecall, AllocatedState, and ManagedState.

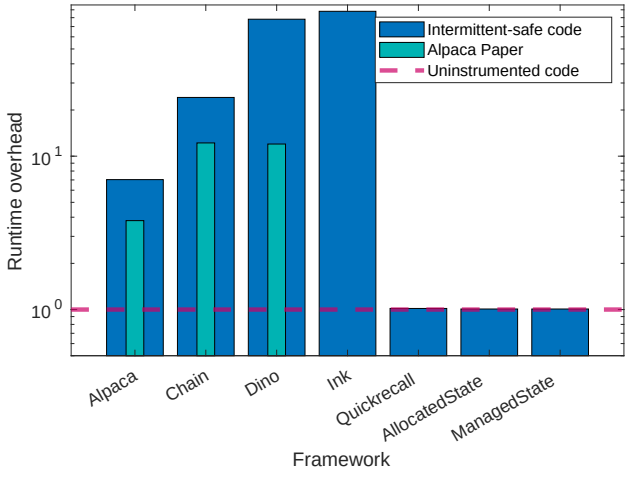
SDO is not applicable for non-JIT systems as the state saving is periodic and not coupled to a voltage interrupt. When it is not zero, the SDO varies between 120 and 1900 µs (Fig. 7.2b). QuickRecall is the most efficient as it just stores the register values of the CPU. AllocatedState and ManagedState take longer as they not only store the register values but rather also save various snapshots of the stack and metadata. Notably, the SDO is consistent for the selected benchmarks.



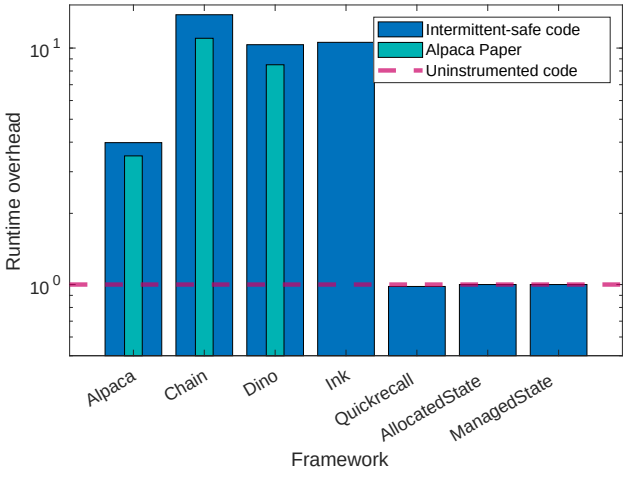
(a) **Effiency runtime for Bitcount.**



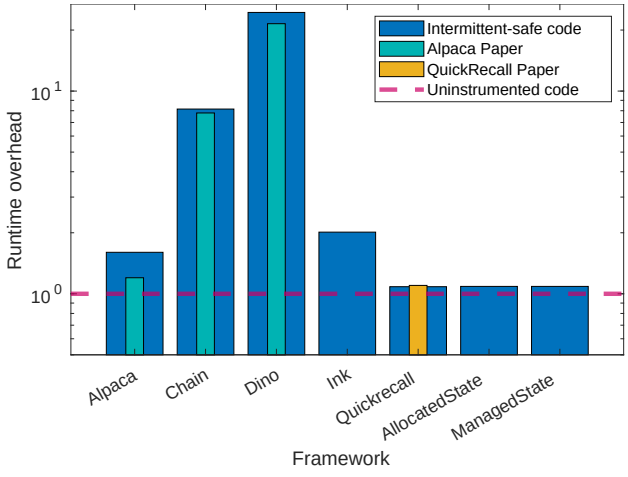
(b) **Effiency runtime for Blowfish.**



(c) **Effiency runtime for CEM.**

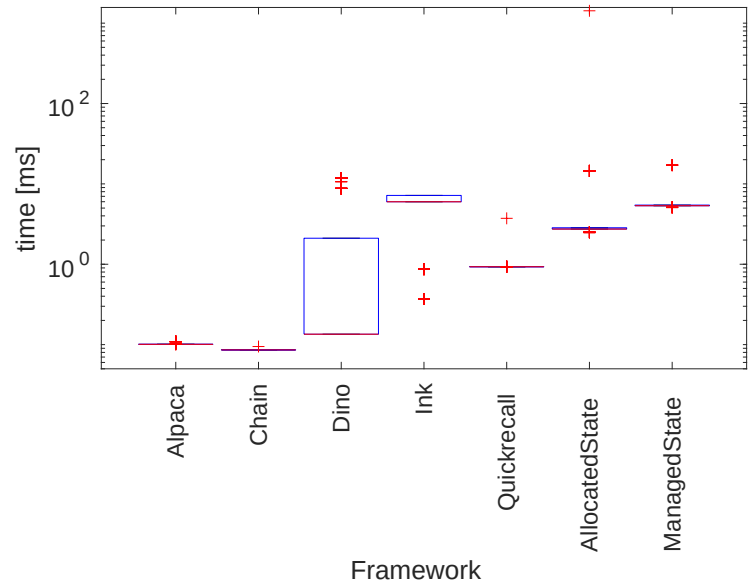


(d) **Effiency runtime for Cuckoo.**

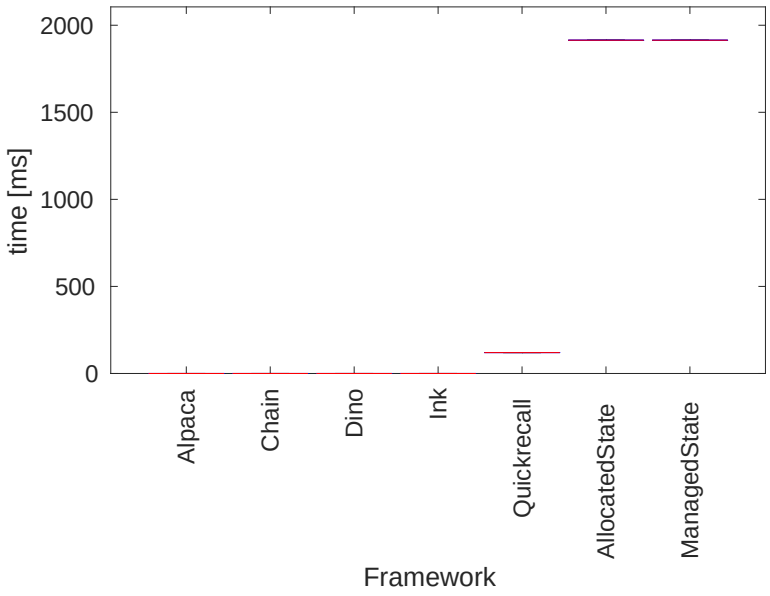


(e) **Effiency runtime for RSA.**

Figure 7.1: **Normalised run-time overhead measured with DIPS+ for the chosen frameworks for the selected benchmarks. The results are compared with the claims of the Alpaca paper [42] and QuickRecall paper [36].**



(a) **Startup Overhead for each selected intermittent framework.**



(b) **Shutdown Overhead for each selected intermittent framework.**

Figure 7.2: **Startup Overhead and Shutdown Overhead for the chosen frameworks aggregated over all selected benchmarks.**

7.2 Stability of Frameworks

One bug that was found during the evaluation is highlighted in this section. In other frameworks, no bugs were found with the selected benchmarks defined in Section 4.4. The found bug can only be replicated automatically using DIPS+ as the DUT needs to experience a power outage multiple times at the right time, as explained below. Finding the bug cannot be done with DIPS [19] as it does not support MSP430. It cannot be replicated with only EDB [11] as that debugger cannot trigger a power outage.

While profiling the SUO of InK during CF, the bug was found. The benchmark did not finish multiple times and required a reflash of the firmware before the benchmark could work again as expected. To understand the bug, first, the program needs to be described. The CF has various tasks and steps. Often, a hardcoded task is returned to specify the next tasks. Two tasks have an exception to this: *task_generate_key* and *task_calc_indexes*. The next task is determined through the variable *_v_next_task*, which is set in a previous task. We can see which functions call those shared tasks and what they set their next_task variable to. However, if the device reboots every second time a task has been executed but not committed yet, the next_task variable is not updated even though the task continues. This invalidates the result of the Cuckoo filtering and moreover points to a bug in the kernel of InK with regards to storing the state of the DUT.

A possible kernel bug of InK would mean that other applications can be affected. However, it is only apparent for CF that the *_v_next_task* variable is used to select the next task. As this variable is corrupted or stored incorrectly, the CF program will be in an endless loop. The other selected frameworks do not have such a variable that causes the DUT to end up in an infinite loop.

Chapter 8

Discussion and Future Work

One of the significant findings of this research is the substantial overhead encountered in *Runtime Overhead* (RTO) and task-based systems. With the large RTO of task-based systems, it seems unlikely they will become ever more efficient than *Just-In-Time* (JIT) systems. The large *Shutdown Overhead* (SDO) and *Startup Overhead* (SUO) of the JIT systems show that they are not useful for short bursts of energy. These findings have significant implications for the development and usage of these battery-free systems.

As we consider the future trajectory of this research, four main areas of focus emerge:

Quality of Software: A significant limitation encountered is the poor state of software for battery-free systems, as illustrated by the artifact survey. This limitation, which caused substantial difficulties in compiling and using the frameworks, suggests a pressing need for better maintenance and user support in software repositories within this field. It is beneficial if publishers, such as ACM, require software artifacts for each paper. For these artifacts, the guidelines for documentation should be more stringent. A specific check for dependency documentation would ensure that the result is reproducible. At the same time, a review of usage documentation would ensure that the work can be used for further research and real-life applications.

Debugger Improvements: Although the MSP430 support of DIPS+ is improved, it can be further optimised. One main bottleneck is the relatively low clock speed that drives *Spy-By-Ware* (SBW). With more research, it could be determined if instructions can be safely performed with a higher frequency.

Improve Emulator Performance: Currently, the minimum energy budget detection is only reliable from 10 us. The main reason for this limitation is the capacitance in the measuring setup. The accuracy can be significantly improved with a higher IO pin driving strength. This will allow more applications to be tested and even benchmarks directly.

Testing Real-world Applications: Benchmarks are only a representative workload and hence can only be used as an indication. Evaluating actual intermittent applications would give more valuable insights.

Chapter 9

Conclusions

This research addresses the absence of uniform testing tools and the lack of a broad and honest evaluation of the performance of battery-free frameworks. Existing debuggers are also limited by their design to specific architectures and do not support the most commonly used platforms. To achieve this, we optimised and extended the platform support of the existing debugger DIPS [19]. Moreover, an automated hardware test was developed to find the minimum energy budget to have forward progress. Finally, the performance of selected frameworks is analysed and compared.

The changes to the debugger allow for a significantly better performance by connecting **11 times faster** while executing **157 times less code** on the *Device Under Test* (DUT) before being fully connected. Using profile mode, we can detect a breakpoint 928 times faster.

The evaluation of the frameworks showed, in addition to one bug in a framework, an enormous overhead. The RTO causes task-based frameworks for certain frameworks and benchmarks to take up to **110 times longer** to execute compared to their uninstrumented counterpart. For JIT-based systems, have the SDO and SUO more impact than RTO.

Overall, this research has shed new light on the performance of intermittent, battery-free frameworks, while also suggesting that comprehensive run-time analysis can evaluate the performance and reliability.

Bibliography

[1] ACM. `https://www-acm-org.tudelft.idm.oclc.org/publications/policies/artifact-review-and-badging-current`, 2020. Last accessed: June 29, 2023.

[2] 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 *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*. ACM, November 2020.

[3] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Efficient intermittent computing with differential checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, June 2019.

[4] Khakim Akhunov and Kasim Sinan Yildirim. AdaMICA. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 6(3):1–30, September 2022.

[5] Abu Bakar, Alexander G. Ross, Kasim Sinan Yildirim, and Josiah Hester. REHASH. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 5(3):1–42, September 2021.

[6] 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 Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12):1968–1980, 2016.

[7] Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. Peripheral state persistence for transiently-powered systems. In *2017 Global Internet of Things Summit (GIoTS)*. IEEE, June 2017.

[8] Naveed Anwar Bhatti and Luca Mottola. HarvOS. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks*. ACM, April 2017.

[9] Black Magic. Black magic debug: The plugplay mcu debugger. `https://black-magic.org/`, 2023. Last accessed: Aug. 21, 2023.

[10] Michael Buettner, Benjamin Greenstein, and David Wetherall. Dewdrop: An Energy-Aware runtime for computational RFID. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[11] Alexei Colin, Graham Harvey, Alanson P. Sample, and Brandon Lucia. An energy-aware debugger for intermittently powered systems. *IEEE Micro*, 37(3):116–125, 2017.

[12] Alexei Colin and Brandon Lucia. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, October 2016.

[13] Alexei Colin and Brandon Lucia. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction*. ACM, February 2018.

[14] European Comission Cordis. `https://cordis.europa.eu/article/id/430457-up-to-78-million-batteries-will-be-discarded-daily-by-2025-researchers-warn`, 2021. Last accessed: June 29, 2023.

[15] Tuan Dang, Trung Tran, Khang Nguyen, Tien Pham, Nhat Pham, Tam Vu, and Phuc Nguyen. ioTree. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*. ACM, October 2022.

[16] Tuan Dang, Trung Tran, Khang Nguyen, Tien Pham, Nhat Pham, Tam Vu, and Phuc Nguyen. Iotree: A battery-free wearable system with biocompatible sensors for continuous tree health monitoring. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, MobiCom '22, page 769–771, New York, NY, USA, 2022. ACM.

[17] Daniel Beer. Mspdebug. `https://github.com/dlbeer/mspdebug`, 2022. Last accessed: Jul. 8, 2023.

[18] Jasper de Winkel, Carlo Delle Donne, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. Reliable timekeeping for intermittent computing. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, March 2020.

[19] Jasper de Winkel, Tom Hoefnagel, Boris Blokland, and Przemysław Pawełczak. Dips: Debug intermittently-powered systems like any embedded system. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, SenSys '22, page 222–235, New York, NY, USA, 2023. ACM.

[20] 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*, 4(3), sep 2020.

[21] 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*, 4(3):1–34, September 2020.

[22] Jasper de Winkel, Haozhe Tang, and Przemysław Pawełczak. Intermittently-powered bluetooth that works. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, page 287–301, New York, NY, USA, 2022. ACM.

[23] Jasper de Winkel, Haozhe Tang, and Przemysław Pawełczak. Intermittently-powered bluetooth that works. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. ACM, June 2022.

[24] Ericsson. Ericsson Mobility Report. Technical report, Ericsson, November 2022.

[25] Joakim Eriksson, Fredrik Østerlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro Josê Marrôn. Cooja/mspsim: interoperability testing for wireless sensor networks. ICST, 5 2010.

[26] Matthew Furlong, Josiah Hester, Kevin Storer, and Jacob Sorber. Realistic simulation for tiny batteryless sensors. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ENSsys'16, page 23–26, New York, NY, USA, 2016. ACM.

[27] GDB Developers. Gdb: The gnu project debugger. `https://www.sourceware.org/gdb/`, 2023. Last accessed: Aug. 20, 2023.

[28] Kai Geissdoerfer, Mikołaj Chwalisz, and Marco Zimmerling. Shepherd. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*. ACM, November 2019.

[29] Google. Protocol buffers documentation. `https://protobuf.dev/`, 2023. Last accessed: Aug. 20, 2023.

[30] Matthew R. Guthaus, Jeff S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization*, pages 3–14, 2001.

[31] Josiah Hester, Timothy Scott, and Jacob Sorber. Ekho. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*. ACM, November 2014.

[32] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. ACM, November 2017.

[33] Tom Hoefnagel. Debugging intermittently-powered embedded systems like any other embedded system. Master thesis, Delft University of Technology, Delft, The Netherlands, 2022.

[34] Bashima Islam and Shahriar Nirjon. Scheduling computational and energy harvesting tasks in deadline-aware intermittent systems. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, April 2020.

[35] Bashima Islam and Shahriar Nirjon. Zygarde. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(3):1–29, September 2020.

[36] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. QUICKRECALL: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*. IEEE, January 2014.

[37] Mustafa Emre Karagozler, Ivan Poupyrev, Gary K. Fedder, and Yuri Suzuki. Paper generators: Harvesting energy from touching, rubbing and sliding. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, page 23–30, New York, NY, USA, 2013. ACM.

[38] Vito Kortbeek, Souradip Ghosh, Josiah Hester, Simone Campanoni, and Przemysław Pawełczak. WARio: efficient code generation for intermittent computing. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, June 2022.

[39] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. Time-sensitive intermittent computing meets legacy software. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, March 2020.

[40] LLVM. The llvm compiler infrastructure project. `https://llvm.org/`, 2023. Last accessed: Jul. 8, 2023.

[41] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2015.

[42] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, October 2017.

[43] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 129–144, Carlsbad, CA, October 2018. USENIX Association.

[44] Kiwan Maeng and Brandon Lucia. Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2020.

[45] Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Discovering the hidden anomalies of intermittent computing. In *Proceedings of the 18th ACM International Conference on Embedded*

Wireless Systems and Networks (EWSN), Delft (The Netherlands), February 2021., 2021.

[46] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Transactions on Sensor Networks*, 16(1):1–24, February 2020.

[47] Alessandro Montanari, Manuja Sharma, Dainius Jenkus, Mohammed Alloulah, Lorena Qendro, and Fahim Kawsar. ePerceptive. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*. ACM, November 2020.

[48] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos. *ACM SIGARCH Computer Architecture News*, 39(1):159–170, March 2011.

[49] Emily Ruppel and Brandon Lucia. Transactional concurrency control for intermittent, energy-harvesting computing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2019.

[50] Saleae. Logic 8 data sheet. `https://downloads.saleae.com/specs/Logic+8+Data+Sheet.pdf`, 2018. Last accessed: Aug. 1=21, 2023.

[51] SEGGER Microcontroller GmbH. J-link – the market-leading debug probe. `https://www.segger.com/products/debug-probes/j-link/`, 2023. Last accessed: Mar. 14, 2023.

[52] Sivert T. Sliper, Domenico Balsamo, Nikos Nikoleris, William Wang, Alex S. Weddell, and Geoff V. Merrett. Efficient state retention through paged memory management for reactive transient computing. In *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, June 2019.

[53] STMicroelectronics. Datasheet - stm32f103xc, stm32f103xd, stm32f103xe. `https://www.farnell.com/datasheets/2815794.pdf`, 2018. Last accessed: Jul. 8, 2023.

[54] Sysprogs. msp430-gdbproxy. `https://github.com/sysprogs/msp430-gdbproxy`, 2022. Last accessed: Jul. 8, 2023.

[55] Vamsi Talla, Bryce Kellogg, Benjamin Ransford, Saman Naderiparizi, Shyamnath Gollakota, and Joshua R. Smith. Powering the next billion devices with wi-fi. *CoRR*, abs/1505.06815, 2015.

[56] Texas Instruments. Msp430 spy-bi-wire with simplelink™ mcus. In *SLAU320AJ*. 2017.

[57] Texas Instruments. Embedded emulation module (eem). In *SLAU367P*, pages 1017–1023. 2020.

[58] Texas Instruments. Msp430fr58xx, msp430fr59xx, and msp430fr6xx family user's guide. In *SLAU367P*. 2020.

[59] Texas Instruments. Msp430 programming with the jtag interface. In *SLAU320AJ*. 2021.

[60] Texas Instruments. Msp-fet hardware programming tool. `https://www.ti.com/tool/MSP-FET`, 2023. Last accessed: Mar. 14, 2023.

[61] Texas Instruments. Msp430-gcc-opensource. `https://www.ti.com/tool/MSP430-GCC-OPENSOURCE#description`, 2023. Last accessed: Jul. 8, 2023.

[62] Texas Instruments. Mspds. `https://www.ti.com/tool/MSPDS`, 2023. Last accessed: Jul. 8, 2023.

[63] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 17–32, Savannah, GA, November 2016. USENIX Association.

[64] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. InK. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. ACM, November 2018.

[65] Eren Yıldız, Lijun Chen, and Kasim Sinan Yıldırım. Immortal threads: Multithreaded event-driven intermittent computing on Ultra-Low-Power microcontrollers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 339–355, Carlsbad, CA, July 2022. USENIX Association.

[66] Joseph Yiu. Interrupt latency on the cortex-m processor family. `https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/beginner-guide-on-interrupt-latency-and-interrupt-latency-of-the-arm-cortex-m-processors`, 2016. Last accessed: Jul. 8, 2023.