HAW HAMBURG

BACHELOR THESIS

Tobias Westphal

Design and Evaluation of a Task Characterization Model for Performance Control of Embedded Devices at Runtime

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Tobias Westphal

Design and Evaluation of a Task Characterization Model for Performance Control of Embedded Devices at Runtime

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of Science Informatik Technischer Systeme*
at the Department Computer Science
at the Faculty of Computer Science and Engineering
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Thomas Schmidt
Supervisor: Prof. Dr. Franz Korf

Submitted on: 06.11.2022

Tobias Westphal

Title of Thesis

Design and Evaluation of a Task Characterization Model for Performance Control of Embedded Devices at Runtime

Keywords

DVFS, ScaleClock, RIOT, Embedded Devices, PMU, Cortex-M4, Debug and Tracing Components, Energy Efficiency

Abstract

When the CPU performance is not fully used by the software, energy is wasted. The goal of this thesis is to improve assessing task properties at runtime to be able to sense the software utilization of the CPU performance. This allows selecting more energy efficient frequency and voltage settings by applying Dynamic Voltage and Frequency Scaling (DVFS). Debug and trace features are available on multiple different Cortex-M CPUs. This thesis evaluates the debug and trace features as a resource for assessing task properties. A feedback mechanism is designed and implemented that uses the trace features without dedicated hardware debug probes. 69 different Bristol/Embecosm Embedded Benchmark Suite (BEEBS) tasks are selected to measure task properties with the task characterization model and to measure the energy efficiency at different hardware configurations. Lastly, the measured task properties are evaluated by their ability to sense the performance utilization by the use of energy measurements. The task characterization model detects whether tasks have a high flash, low RAM or special peripheral access. It can also track the number of cycles the CPU is sleeping or how many instructions are executed for a specific task. A threshold technique that uses the task property *cycles per instruction* selects the most energy efficient or a more energy efficient frequency setting for 78.7% of all processing tasks. Thereby, the task property only needs to be traced at a single CPU frequency. The BEEBS tasks use up to 35% less energy with 30% longer execution time by reducing the CPU frequency. Tracing the task property *instructions executed* additionally increases the power consumption of BEEBS tasks by 6.32% to 8.01%, but tracing does not always need to be active.

Tobias Westphal

Thema der Arbeit

Design und Evaluation eines Models zur Task-Charakterisierung für die Leistungskontrolle von eingebetten Systemen zur Laufzeit

Stichworte

DVFS, ScaleClock, RIOT, Eingebettete Systeme, PMU, Cortex-M4, Debug und Tracing Komponenten, Energieeffizienz

Kurzzusammenfassung

Nutzt eine Software die Leistung der CPU nicht vollständig, so wird Energie verschwendet. Ziel dieser Thesis ist es, das Zuweisen von Task-Eigenschaften während der Laufzeit zu optimieren, sodass die Nutzung der CPU-Leistung erkannt wird. Dabei können mit Dynamic Voltage and Frequency Scaling (DVFS) energieeffizientere Frequenz- und Spannungseinstellungen ausgewählt werden. Auf mehreren Cortex-M CPUs sind "debug and trace features" vorhanden. Letztere werden in dieser Arbeit evaluiert, um Task-Eigenschaften zu messen. Es wird ein Feedback-Mechanismus entworfen und implementiert, der die "trace features" ohne dedizierte Debug-Adapter nutzt. 69 verschiedene Bristol/Embecosm Embedded Benchmark Suite (BEEBS) Tasks werden ausgewählt, um Task-Eigenschaften sowie die Energieeffizienz mit verschiedenen Hardwareeinstellungen zu messen. Zuletzt wird unter der Hinzunahme von Energiemessungen evaluiert, ob durch die gemessenen Task-Eigenschaften die Nutzung der CPU-Leistung erkennbar ist. Das Modell zur Task-Charakterisierung zeigt an, ob Tasks einen hohen Flash, eine niedrigen RAM oder Peripherie-Zugriff besitzen. Es kann außerdem detektieren, für wie viele Zyklen eine CPU schläft oder wie viele Instruktionen für einen Task ausgeführt werden. Mit der Task-Eigenschaft "Zyklen pro Instruktionen" können für 78,7 % aller Tasks die effizienteste oder eine effizientere Frequenzeinstellung erkannt werden. Hierbei muss die Task-Eigenschaft nur bei einer einzigen Frequenzeinstellung gemessen werden. Durch Reduzierung der CPU-Frequenz nutzen einige BEEBs Tasks bis zu 35 % weniger Energie bei nur 30 % längerer Laufzeit. Der Verbrauch der BEEBS Tasks wird beim Messen der ausgeführten Instruktionen um 6,32 % bis 8,01 % zusätzlich erhöht, doch das Messen der Task-Eigenschaften muss nicht durchgehend aktiv sein.

Contents

List of Figures ix

List of Tables xv

Abbreviations xvii

1 Introduction 1

1.1 Motivation . . . 1
1.2 Outline . . . 2

2 Background and Related Work 4

2.1 Source and Composition of Energy Consumption . . . 4
2.1.1 CMOS Transistor Structure and Behavior . . . 4
2.1.2 Propagation Delay . . . 6
2.1.3 Static Power Consumption . . . 7
2.1.4 Dynamic Power Consumption . . . 7
2.1.5 Total Energy Consumption . . . 8
2.1.6 Energy Efficiency . . . 9
2.2 Related Work . . . 9
2.2.1 Types of Energy-Saving Techniques . . . 9
2.2.2 DVFS Spectrum . . . 10
2.3 Debug and Trace Features with Arm Cortex-M . . . 13
2.3.1 Common Use . . . 13
2.3.2 Debug Features . . . 14
2.3.3 Trace Features . . . 14

3 Scope of Thesis 16

3.1 Potential Performance-Controlling Approach . . . 16
3.2 Goals and Requirements . . . 17

4 Debug and Trace Feature Evaluation for Usability **19**
4.1 Data Watchpoint and Trace unit (DWT) 19
4.1.1 Profiling Counters 19
4.1.2 Comparators 22
4.2 Embedded Trace Macrocell (ETM) 23
4.3 Instrumentation Trace Macrocell (ITM) 24
4.4 Trace Packet Path and Trace Port Interface Unit (TPIU) 24
4.5 Summary of Feature Evaluation 25

5 Concept of Task Characterization Model **26**
5.1 Accessibility Discussion of Trace Features 26
5.1.1 Register Polling 26
5.1.2 Deserialization 27
5.1.3 Counting of Trace Packet Flanks 28
5.2 Feedback Mechanism with Timer Counter 29
5.3 Concept of Software Trace 31
5.4 Spread of Concept Use 32

6 Implementation of Task Characterization Model **35**
6.1 Board Choice for Implementation 35
6.2 Hardware Assembly on the Nucleo-L476RG 36
6.2.1 Requirements for Access to Serial Wire Output (SWO) Line 36
6.2.2 Wiring for Feedback Mechanism 37
6.3 Software Implementation of Trace Utility 38
6.3.1 Memory Coverage with Comparators on STM32L476RGT6 40
6.3.2 Peripheral Device Coverage with Comparators on STM32L476RGT6 41
6.4 Summary of Implementation constraints 42

7 TPIU Configuration **43**
7.1 TPIU encoding 43
7.2 TPIU prescaler 44
7.2.1 TPIU Prescaler for ECP 45
7.2.2 TPIU Prescaler with DTP 46
7.3 Flanks Per DTAOP 48

8 Methodology of Measurements 51
8.1 Measurement Principles . . . 51
8.1.1 Reproducibility . . . 51
8.1.2 Soundness . . . 51
8.1.3 Automation . . . 52
8.1.4 Verifiability . . . 52
8.2 Measurement Setup . . . 53
8.2.1 Software Components . . . 53
8.2.2 Experiment Procedure and Interaction . . . 55
8.2.3 Time Measurements . . . 56
8.2.4 Power and Energy Measurements . . . 57
8.3 Selecting a Benchmark Suite for Embedded Devices . . . 59
8.3.1 SPEC CPU . . . 59
8.3.2 MediaBench . . . 59
8.3.3 MiBench . . . 60
8.3.4 ERCBench . . . 60
8.3.5 BEEBS . . . 60
8.4 Measurements and Data Processing . . . 61
8.4.1 Frequency, Voltage and Flash Wait State (FWS) Configuration . . 61
8.4.2 Data Processing for Traced Task Properties . . . 63

9 Overhead of Task Characterization 65
9.1 Delay of ECP and DTAOP . . . 65
9.2 Overhead in Time . . . 68
9.3 Overhead in Power Consumption . . . 71

10 Evaluation of Tracing and Energy Results 79
10.1 Register, RAM or Flash intensive Workloads . . . 79
10.2 I/O Workload . . . 84
10.3 Inactivity . . . 89
10.4 Application-focused Tasks with BEEBS . . . 90
10.4.1 Model-Independent Task Properties . . . 91
10.4.2 Tracing Inaccuracies . . . 95
10.4.3 Tracing the Cycle Amount . . . 96
10.4.4 Tracing with Comparators . . . 100
10.4.5 Tracing with Profiling Counters . . . 110

10.4.6 Counter Combinations to Improve the Most Energy Efficient CPU Frequency (MEECF) Selection 115
10.4.7 Cycle Tracings at Different CPU Frequencies 119
10.5 Tracing Overhead for Selected Task Properties in Practice 125
10.5.1 Cycles Per Instruction 125
10.5.2 Cycles Saved 126
10.5.3 Flash Access 126

11 Conclusion and Outlook **127**
11.1 Achievements 127
11.2 Problems 128
11.3 Future Work 129

Bibliography **131**

Glossary **139**
Declaration of Autorship 140

List of Figures

2.1 CMOS schematic, based on [1, sec. 5.2]. 4
2.2 MOS transistor cross-section, based on [2]. 5
2.3 Behavior of a switching CMOS inverter, based on [1, sec. 5.2]. 6
2.4 Overview of trace and debug connections in the MCU and the connection to the host PC, based on [3, 4]. 13

3.1 High-level schematic of a potential performance-controlling approach. . . . 16

4.1 Format of an Event Counter Packet, based on [4, sec. D.3.1]. 20
4.2 Format of a Data Trace Address Offset Packet (DTAOP), which is a type of Data Trace Packets (DTPs). It is dependent on the comparator ID and the offset of the matched address. Based on [4, sec. D.3.4]. 22
4.3 Relation of the Microcontroller Profile of Version 7 of the Arm Architecture (ARMv7-M) trace components and the path of different packet streams, based on [4, sec. C1.7.1]. 24

5.1 Exemplary CPI overflow Event Counter Packet (ECP) packet represented as a signal with flanks and the reading with a timer counter. 29
5.2 Existing connection of relevant trace components (blue) and additional connection of components that enable the feedback of the trace packets (green). 30
5.3 Procedure concept of tracing a task with a profiling counter. Exemplary measuring the *LSU* counter is shown. 31
5.4 Component diagram of software components for the task characterization model. 32
5.5 RIOT (2021.07 version) boards with Cortex-M processors that are potentially suitable for task characterization model. Boards analyzed via Kconfig via the *cpu* and *boards* directory. 34

6.1 Debug component and connection overview with Serial Wire / JTAG Debug Port (SWJ-DP) zoomed in (left) for the STM32L476RGT6. Figure based on [5]. 36
6.2 Connection of pins with cables on the Nucleo-L476RG board to count generated trace packets with the feedback mechanism. Figure based on [6]. 37
6.3 Class diagram of trace utility as designed in Figure 5.4. 39
6.4 RAM and flash memory address space in comparison to the sum of all mask-able address ranges of the comparators on the STM32L476RGT6 MCU. 40

7.1 Comparison of CYCLE ECP signals between the *Non Return to Zero (NRZ)* and *Manchester* encoding. The signals are measured with an oscilloscope (see Section 8.2.3) at a CPU frequency of 13 MHz and a configured Trace Port Interface Unit (TPIU) prescaler of 2. 44
7.2 Measured flanks of fixed number of generated *CYCLE* ECPs. Experiment performed with different TPIU prescaler values and at a CPU frequency of 53 and 80 MHz. 46
7.3 Measured Data Trace Address Offset Packet (DTAOP) flanks per iteration of flash data access (left) vs theoretically generate-able DTAOP flanks (right) at a CPU frequency of 80 MHz and different TPIU prescaler values. 48
7.4 Frequency of Flanks per Packet of DTAOPs across all available comparators (0 to 3) and all available data address offset values (0x0 to 0xffff) at a CPU frequency of 80 MHz. 49

8.1 Component diagram of the measurement setup. 53
8.2 Sequence diagram of the interaction of different experiment setup components. The interaction of benchmarking a task running on a microcontroller and measuring the current with the multimeter is shown. 55
8.3 Measured delay between two GPIO flanks dependent on their polarity and number of GPIO used at different CPU frequencies. 57
8.4 Connection of the Nucleo-L476RG board and the Digital Multimeter (DMM) to measure the microcontroller current, Figure based on [7]. 58
8.5 Relation between Flash Wait State (FWS), CPU voltage and CPU frequencies dependent on the configured Dynamic Voltage Scaling Policy (DVS Policy) and enabled Dynamic Voltage Scaling (DVS). 62

9.1 Delay between the generation of different hardware source packets and the arrival of the first packet flank at timer counter. Measured at different CPU frequencies and a TPIU prescaler value of 0. 66
9.2 Delay between the generation of *CYCLE* ECPs and the arrival of the first flank at the timer counter. Measured at different CPU frequencies and different TPIU prescaler values. 66
9.3 Packet length (delay between first and last flank) for trace packets (ECP(left), DTAOP(right)). Measured at different TPIU prescaler values and different CPU frequencies. 67
9.4 Delay of Figure 9.3 combined with the inaccuracy of using General Purpose Input/Outputs (GPIOs) pins for triggering (see Figure 8.3). 68
9.5 Overhead delay performing the tracing initialization steps. 69
9.6 Overhead delay of tracing control functions dependent on trace method (profiling counter, comparator). 70
9.7 Visualization of different trace configurations measured for power consumption in Figure 9.8. 71
9.8 Overhead power consumption of different trace configurations in proportion to overhead power consumption of *TRACE_NO_SWO*, grouped by different workloads and measured at a CPU frequency of 80 MHz. 72
9.9 Overhead power consumption of tracing different task properties and corresponding traced counter flanks. The trace has been performed at a CPU frequency of 80 MHz. 74
9.10 Overhead power consumption of tracing flash data accesses at different CPU frequencies, with enabled Flash Wait State Adaption and selected *Fast Flash* DVS Policy. 76
9.11 Overhead power consumption of enabled tracing in proportion to the power consumption of workloads without enabling tracing. Illustrated with different workloads and CPU frequencies. Measured with Flash Wait State Adaption has been enabled and the *Fast Flash* DVS Policy has been selected. 77

10.1 Normalized profiling counter results measured with tasks of different memory access types and math operations. The measurements were performed at a CPU frequency of 80MHz. 80
10.2 Normalized flash/ Random-Access Memory (RAM) data access measured for tasks with different memory access types and math operation types. The task properties are measured at a CPU frequency of 80MHz. 81

10.3 Profiling counter and cycle trace results of tasks performing ADD operation with different memory access types. The measurements were performed at different CPU frequencies, with enabled Flash Wait State Adaption and *Fast Flash* DVS Policy. 82
10.4 Energy consumption at different CPU frequencies in proportion to the energy consumption at 80MHz. The proportion results are grouped by the math operation type and memory access type, and separated by the DVS Policy. The energy consumption is measured with enabled FWSA and DVS. 83
10.5 Trace results for the SPI intensive workload executed at different Serial Peripheral Interface (SPI) frequencies and CPU frequencies. The task properties are measured with enabled Flash Wait State Adaption and *Fast Flash* DVS Policy. 85
10.6 RAM/flash data access per cycle measured with the SPI workload. The results are grouped by SPI frequency and CPU frequency. The task properties are traced with enabled FWS and *Fast Flash* DVS Policy. 87
10.7 Energy consumption at different CPU frequencies in proportion to the energy consumption at 80 MHz measured with the SPI workload. The proportion results are grouped by the DVS Policy and the selected SPI frequency. The energy consumption is measured with enabled FWSA and DVS. 88
10.8 Peripheral data accesses per cycle of the SPI intensive workload. The data accesses are measured with DAM, as seen in Section 6.3.2. 89
10.9 SLEEP intensive workload properties measured at different CPU frequencies and different DVS Policy. 90
10.10 Execution time per iteration at a CPU frequency of 80 MHz. The y-axis is shown in logarithmic scale and the tasks are sorted by the highest value. 91
10.11 Average power consumption at different CPU frequencies. Grouped by CPU frequency and the DVS Policy. The power consumption was measured with enabled Flash Wait State Adaption. The tasks are sorted by the highest power consumption at 80MHz. 92
10.12 Energy consumption at different CPU frequencies and different DVS Policy in proportion to energy consumption at 80 MHz. The energy was measured with enabled Flash Wait State Adaption and DVS. The tasks are first sorted by the biggest energy consumption saving and secondly by the biggest energy consumption increase. 93

10.13 Execution time increase of different CPU frequency to 80 MHz. The execution time has been measured with enabled Flash Wait State Adaption and different DVS Policys (upper/lower figure). The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 94
10.14 Standard deviation of measured counters in proportion to mean of the measured counters that is calculated with ten repetitions. The proportion is grouped by the counter type and the CPU frequency. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 96
10.15 Cycle count at different CPU frequencies in proportion to the cycle count at 80 MHz. The measurements were performed with enabled Flash Wait State Adaption and the *FAST FLASH* DVS Policy. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 97
10.16 Cycle count at different CPU frequencies in proportion cycle count at 80 MHz. The cycles were measured with disabled Flash Wait State Adaption and configured *FAST FLASH* DVS Policy. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 98
10.17 Energy consumption at different CPU frequencies in proportion to energy consumption at 80 MHz. The measurements were performed with disabled Flash Wait State Adaption, different DVS Policys and enabled DVS. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 99
10.18 Memory accesses per cycle measured at a CPU frequency of 80 MHz. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 100
10.19 Difference of normalized comparator counter results (13MHz - 80MHz) with enabled *FAST FLASH* DVS Policy and enabled Flash Wait State Adaption (FWSA). The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 103
10.20 Comparison between the count of saved cycles at 13MHz with different flash cache states and the normalized flash access at an CPU frequency of 80 MHz. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 105
10.21 Amount of instructions per task assembler file. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 107

10.22 Amount of float and double operation instructions per task assembler file. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 108
10.23 Energy consumption at different CPU frequencies in proportion to the energy consumption at 80MHz measured with the *REG* task of Section 10.1, but with float and double variables. The plot is grouped by the math operation and used variable types. The data was measured with enabled FWSA and *Fast Flash* DVS Policy. 109
10.24 LSU, FOLD and CPI profiling counter in proportion to cycles. Measured at a CPU frequency of 80 MHz. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 111
10.25 NCC of Equation 10.1 at a CPU Frequency of 80 MHz. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 116
10.26 LSU and CPI counters in proportion to cycles for tasks with lower counters than 0.2. Measured at a CPU frequency of 80 MHz and The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 117
10.27 Cycles in proportion to the calculated instruction count (see Equation 4.1) at a CPU frequency of 80 MHz. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 118
10.28 Calculated number of cycles per CPU frequency in proportion to the number of cycles at 80 MHz. The calculation is based on Equation 10.3 and uses cycle measurements at 80 and 53 MHz. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 121
10.29 Average static power consumption per CPU frequency of all tasks of Figure 10.11 and extrapolated power consumption to a CPU frequency of 0 MHz. 122
10.30 Calculated $\alpha \cdot C$ factor at different CPU frequencies with Equation 10.5. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 123
10.31 Estimated Energy Consumption as shown in Equation 10.4 with cycle measurements of 80 and 13 MHz, grouped by CPU frequency. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12. 124

List of Tables

3.1 Thesis goals and requirements. 18

4.1 Description of available profiling counters and CYC counter of the DWT, based on [4, 8]. 20
4.2 Summary of debug and trace features as potential indicators of performance utilization and constraints for usability at runtime. 25

5.1 Supported trace features of Cortex-M processors. Table based on [3, table 9]. 33

6.1 Address space and the number of comparators needed to trace the data access of different peripherals on the STM32L476RGT6 Microcontroller Unit (MCU). 41
6.2 Summary of tracing constraints with the Nucleo-L476RG board. 42

7.1 Summary of ideal TPIU configuration in regard to transmission accuracy and highest possible throughput. 50

8.1 Properties to look for when searching for a suitable benchmark suite for this thesis. 59
8.2 Trace method properties to calculate the original counter reading from measured flanks. 63

9.1 Summary of the task characterization overhead. 78

10.1 Potential indicators for more energy efficient frequency settings. 84
10.2 Actual SPI frequency dependent on the CPU and selected SPI frequency. . 86
10.3 Linear correlation of all BEEBS tasks between the saved cycle proportion of Figure 10.15 and the energy proportion of Figure 10.12 and grouped by the CPU frequency. 97

10.4 Findings for selecting a higher/lower MEECF with Data Address Matching (DAM) and the BEEBS tasks. 110
10.5 Highest linear correlation between profiling counter results of Figure 10.24 and the saved cycles of Figure 10.15. 111
10.6 Profiling counter findings with BEEBS tasks for selecting a higher/lower MEECF. 115
10.7 Highest correlations between the *cycles per instructions* task property at a CPU frequency of 80 MHz and the saved cycles of Figure 10.15, grouped by the task subgroups of Figure 10.12. 119

Abbreviations

ARMv7-M Microcontroller Profile of Version 7 of the Arm Architecture.

ART Accelerator Adaptive Real-Time Memory Accelerator.

BEEBS Bristol/Embecosm Embedded Benchmark Suite.

CMOS Complementary Metal-Oxide-Semiconductor.

DAM Data Address Matching.

DMM Digital Multimeter.

DTAOP Data Trace Address Offset Packet.

DTP Data Trace Packet.

DVFS Dynamic Voltage and Frequency Scaling.

DVS Dynamic Voltage Scaling.

DVS Policy Dynamic Voltage Scaling Policy.

DWT Data Watchpoint and Trace unit.

ECP Event Counter Packet.

ETM Embedded Trace Macrocell.

FWS Flash Wait State.

FWSA Flash Wait State Adaption.

GPIO General Purpose Input/Output.

IAM Instruction Address Matching.

IoT Internet of Things.

ITM Instrumentation Trace Macrocell.

MCU Microcontroller Unit.

MEECF Most Energy Efficient CPU Frequency.

NRZ Non Return to Zero.

PMU Performance Monitoring Unit.

RAM Random-Access Memory.

SPI Serial Peripheral Interface.

SWD Serial Wire Debug.

SWJ-DP Serial Wire / JTAG Debug Port.

SWO Serial Wire Output.

TPIU Trace Port Interface Unit.

1 Introduction

1.1 Motivation

The diversity in applications of embedded devices is high. Home automation, industrial controls or ubiquitous urban sensing [9] are just some examples of these applications [10]. While the embedded devices differ in their availability of energy from permanently wired devices to energy-harvesting devices [9], they share the same goal of maximizing energy efficiency to be conformed to green computing or as it extends battery life [11].

As embedded systems often perform a few predefined tasks with specific requirements, many embedded Internet of Things (IoT) operating systems only implement "the clock configuration in static code that is only configurable before compilation [10]". A static clock frequency compromises energy efficiency [12] because a single clock configuration can not be most energy efficient for every task [10]. A widely known technique to improve this situation is to dynamically change the system clock frequency and voltage to trade-off between energy and performance at runtime [13]. This technique is called Dynamic Voltage and Frequency Scaling (DVFS) and can potentially save energy when the CPU performance is not fully used [10].

This thesis focuses on finding resources that point to "mismatches between the performance configuration of the hardware and the utilization by software [10]" at runtime to potentially increase energy efficiency. Sensing the utilization at runtime is often achieved by comparing the execution time of the idle process to the non-idle execution time [14] or by using hardware performance counters that can be used to detect memory bottlenecks [15]. While the idle metric is susceptible to tasks that only appear to require high performance [10], hardware performance counters are not common on low-power embedded devices.

The IoT operating system RIOT [16] offers the module ScaleClock [17] to enable dynamic clock reconfiguration at runtime. At the time of this research the boards Nucleo-L476RG

by STMicroelectronics [18] and the SLSTK3402A EFM32 Pearl Gecko PG12 by Silicon Labs [19] were already ported to the ScaleClock implementation. Each Microcontroller Unit (MCU) of both boards is based on the ARM Cortex-M4 CPU, which offers debug components that trace performance statistics, data access or even instruction fetching [4, sec. C1.1].

The question of this thesis is whether the features of the debug components can be used as a resource to trace task properties at runtime and to which extent the traced task properties indicate tasks which improve their energy efficiency by lowering the CPU frequency. Hence, the debug and trace features are evaluated for usability, a feedback mechanism is designed to get access to the trace features at runtime and the mechanism is used to trace different task properties of 69 different tasks of the Bristol/Embecosm Embedded Benchmark Suite (BEEBS)[20].

1.2 Outline

Chapter 2 shows the background of energy consumption for Complementary Metal-Oxide-Semiconductor (CMOS) circuitry, the spectrum of devices that use Dynamic Voltage and Frequency Scaling (DVFS) and an introduction to the Cortex-M debug and trace features. Chapter 3 goes into detail about the goal of this thesis and which steps are taken to achieve this goal. The first step is Chapter 4 where the trace and debug features of the ARM Cortex-M4 CPU will be evaluated regarding their usability to trace task properties, which potentially indicate the performance utilization. After that, Chapter 5 outlines possible ways of enabling access to the trace features without the use of a dedicated debug probe. A feedback mechanism is designed that has a low realization cost and is not invasive in terms of CPU load. Chapter 6 goes into detail about the implementation of the feedback mechanism in hardware on the Nucleo-L476RG, which is the chosen target board. Furthermore, an implementation of the tracing utility in software is presented. The TPIU configuration, which affects the tracing accuracy, will be examined in Chapter 7. Chapter 8 shows the measurement methodology regarding measurement setup, selection of a benchmark task suite and measurement data processing. In Chapter 9 the trace features and the implementation of the feedback mechanism will be examined in terms of time and energy overhead. The traced task properties are evaluated in Chapter 10 on synthetic tasks and further on tasks of a more representative benchmark suite. Chapter

11 summarizes the approach, highlights problems and gives further note on what could be done to improve the designed task characterization model.

2 Background and Related Work

2.1 Source and Composition of Energy Consumption

DVFS is said to be a trade-off between high performance and low power [13]. Thereby, it alters the clock frequency (f) and the system voltage (V). To understand why this manipulation alters the performance and power consumption of a system like a microcontroller, it is useful to first zoom from this high-level understanding to the elements that a circuit is made of. Thereby, the Complementary Metal-Oxide-Semiconductor (CMOS) circuitry is only inspected schematically.

2.1.1 CMOS Transistor Structure and Behavior

"The electrical behavior of complex circuits [such as NAND, NOR, or XOR, which in turn form the building blocks for processors,] can be almost completely derived by extrapolating the results obtained for inverters [1, sec. 5.1]".

Fig. 2.1 shows the schematic for a genuine CMOS inverter. The inverter consists of two MOS transistors (a NMOS and a PMOS transistor) with complementary behavior regarding the V_{in} level.

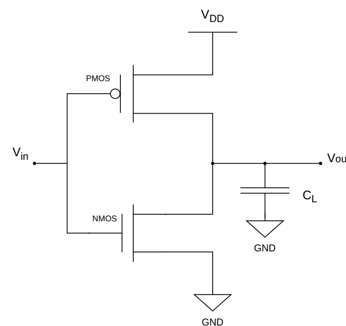


Figure 2.1: CMOS schematic, based on [1, sec. 5.2].

When V_{in} is equal to 0, the PMOS transistor is on, while the NMOS transistor is off. This enables a connection between V_{out} and the V_{DD}, resulting in a steady-state high output voltage. When V_{in} is equal to V_{DD}, the inverter's V_{out} is connected to ground and therefore has a "steady-state value of 0[1, sec. 5.2]".

The underlying MOS transistors are simple switches with an infinite resistance for $|V_{in}| < |V_T|$ and a finite on-resistance for $|V_{in}| > |V_T|$ [1, sec.5.2].

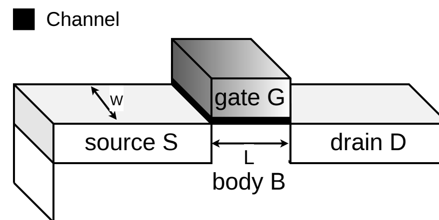


Figure 2.2: MOS transistor cross-section, based on [2].

A MOS transistor consists of a source, drain, body and a gate. The source and the drain are isolated from each other by the body, through which current ideally can not flow. A current flow I_D between the source and drain is controlled by the voltage at the gate terminal relative to the source V_{GS} [1, sec.3.3.1].

Because the gate is isolated from the rest, with more voltage an electrical field builds up, influencing the material between the source and the drain. This electrical field acts as a capacitor (C_L) to its driving circuitry [2, sec.2].

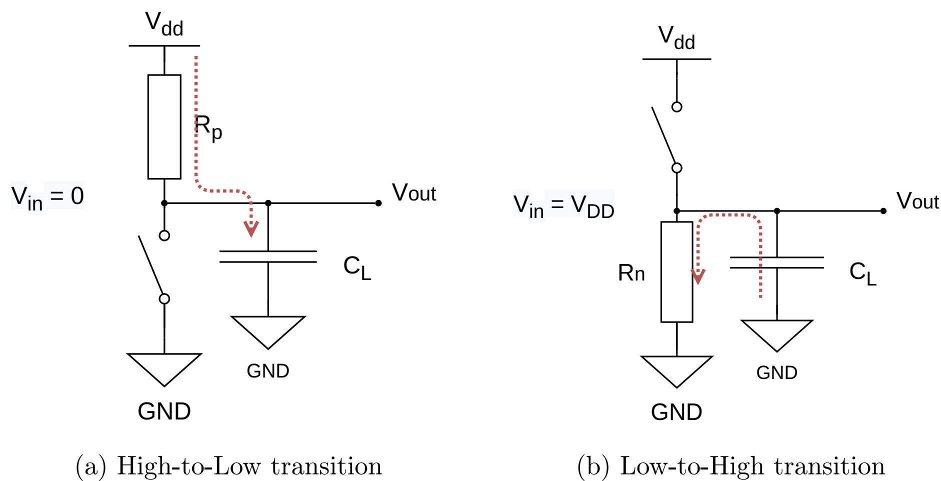
The voltage V_{GS} required to turn on the transistor is the threshold voltage V_T. Once the gate capacitor is fully charged to its desired state, no further current is ideally required to maintain that state [2, sec. 2].

The threshold voltage V_T which gives a transistor the switch-like behavior is dependent on many material constants. In contrast to the switch-like behavior, the current I_D that flows between the drain and the source rises exponentially with the increase of V_{GS}. Unfortunately, with $V_{GS} < V_T$ an unwanted current flow is present, which is the subthreshold leakage that is part of the static power consumption [1, sec.3.3, 5.5.2].

2.1.2 Propagation Delay

In normal operation in a circuitry, inverters are switched off or on many times in a short timespan. The propagation delay is the duration it takes to switch an inverter. This duration is determined by how long it takes to charge or discharge the load capacitance C_L of the NMOS and PMOS transistors through a resistor R_{eq}, which is voltage-dependent [1, sec.5.4.2, sec.1].

Figure 2.3 shows the inverter switching states which consist of a low-to-high or a high-to-low transition. The overall propagation delay of an inverter is defined as the average of these two phases. The propagation delay rises with the reduction of V_{DD} and has a more significant increase starting below $\approx 2V_T$, which therefore should be avoided [1, sec.5.4.2].



(a) High-to-Low transition (b) Low-to-High transition

Figure 2.3: Behavior of a switching CMOS inverter, based on [1, sec. 5.2].

As the propagation delay rises with the reduction of V_{DD} and the clock frequency defines how often an inverter is switched per second, the clock frequency has to be reduced if the voltage is reduced. Otherwise, the time to charge/discharge the load capacitance will be too short to guarantee the proper functioning of the inverter.

2.1.3 Static Power Consumption

The static power dissipation is a result of various leakage modes of the MOS transistors, with its highest component being conflicted with the *subthreshold leakage* [2, sec. 2]. Therefore, static power consumption is not dependent on the currently performing workload but is always present if the system is powered on and is dependent on technology parameters [2, sec. 1].

The overall static power consumption of CMOS circuitry is modelled by the formula [2, sec. 1]:

$$P_{leak} = V_{DD} \cdot I_{leak} \cdot N \cdot k_{design} \tag{2.1}$$

With k_{design} representing a device design factor, N being the number of transistors in the design, V being the power supply voltage and I_{leak} representing the current that flows between the supply rails in the absence of switching activity (subthreshold leakage). [2, abstract]

2.1.4 Dynamic Power Consumption

The dynamic power dissipation mainly takes place when the transistor is switching. Each switching cycle consists of a Low-to-High and a High-to-Low inverter transition. During each inverter transition, the load capacitor C_L gets charged or discharged, and its voltage rises from 0 to V_{DD}. Thereby, both transitions are dissipating energy and collectively consume the amount of energy equal to $C_L V_{DD}^2$ [1, sec. 5.5.1,].

Equation 2.2 formulates the dynamic power consumption which also considers a time component on how often the inverter switching is happening:

$$P_{dyn} = V^2 \cdot \alpha \cdot C_L \cdot f \tag{2.2}$$

With f being the tunable system clock frequency, C_L being the total load capacitance of the circuitry, which is hardware dependent, and α being the switching activity, which is task dependent. The switching activity is a factor taking into account that in the actual circuit not all inverters are switching in each cycle. Therefore, α represents a value between 0 and 1. [21, 2, 11, 12]

2.1.5 Total Energy Consumption

When measuring the energy a system is consuming, the current that the system draws from the power supply with a certain voltage is measured over a certain amount of time.

This energy consumption is the combination of static and dynamic power consumption over a certain amount of time t, which can simplified be formulated as: [1, sec. 5.5.3, sec. 5.7]

$$P_{total} = P_{dyn} + P_{leak} \tag{2.3}$$

$$E_{total} = t \cdot P_{total} = t \cdot P_{dyn} + t \cdot P_{leak} \tag{2.4}$$

Considering a system is performing a task at a certain clock frequency for a certain amount of time, the number of cycles needed to perform that task can be formulated by:

$$t = \frac{CYC}{f} \tag{2.5}$$

with CYC being cycles [22].

Combining Equation 2.4 with Equation 2.5 forms the following:

$$E_{total} = CYC \cdot V^2 \cdot \alpha \cdot C_L + \frac{CYC}{f} \cdot P_{leak} \tag{2.6}$$

With the knowledge of how the total energy consumption of a system is composed of, a better explanation can be made on why DVFS trades off performance for power and energy reduction [23, sec. 1].

A reduction in frequency reduces dynamic power consumption, which results in a lower total power consumption, but also more time is needed to perform a certain number of cycles. An increase in clock frequency increases the total power consumption but reduces the task duration with a constant number of cycles, meaning a higher performance.

2.1.6 Energy Efficiency

Reducing the frequency reduces the dynamic power consumption but increases the execution time. Considering Equation 2.6, if the number of cycles stays the same and the execution time increases, the proportion of the static energy consumption compared to the total energy rises. Hence, if the same number of cycles at lower CPU frequencies is needed, it is most energy efficient to configure the highest CPU frequency.

Only with a further voltage reduction at a low enough clock frequency [1, 24] the static power can also be reduced, which might result in lower total energy consumption. [25, sec. 1 Introduction] But the range available for voltage on modern MCUs is significantly more limited than for frequency [10].

Furthermore, derived from Equation 2.6, a lower and more energy efficient frequency might be possible if the number of cycles is reduced at lower clock frequencies, as this reduces both dynamic and static energy consumption.

Therefore, precise knowledge of the task behavior is key to selecting the optimal frequency [10] and the goal is to avoid or at minimum reduce needless cycles [2].

2.2 Related Work

To better understand the field of energy saving techniques and especially the field of DVFS, a short overview is given.

2.2.1 Types of Energy-Saving Techniques

The potential of DVFS is fundamentally to reduce the electrical energy consumption without significantly compromising performance at runtime [26]. Even though this thesis concentrates on DVFS, the spectrum of CMOS techniques that also share the same basic goal is wide and should be surveyed first. Mittal [11, sec. 3] categorizes the spectrum of embedded energy consumption reduction techniques into 4 classes with one being DVFS.

The second class are low power modes of device hardware. Different modes disable or reduce the functionality of different parts of the system, for example through clock

gating, disabling the clock for clock tree branches [27, sec. Background]. Thereby, different modes save different amounts of energy but also take different time to return to normal mode. In comparison to DVFS, a MCU can not perform work while a low power mode disables the CPU and the performance in normal mode is not controllable [10, sec. 4.1.1]. Power modes should not be seen as a competitor technology but could be combined with DVFS [11, sec. 4.2].

The third class is characterized by techniques that exploit application behavior and apply for specific components. The stated are mainly techniques for RAM/cache usage, like RAM compression [11, sec. 4.3].

The last class is categorized as using special hardware like GPUs, FPGAs, ASICs that are especially efficient for a specific application type. For example, matrix multiplication is very efficient on GPUs in comparison to CPUs [11, sec. 4.4].

2.2.2 DVFS Spectrum

This thesis is focused on DVFS at the CPU-level, but more precisely about single core, low-power embedded devices with no real-time constraints. First, an overview of the DVFS research is given to strengthen the differentiation of this thesis to other DVFS topics.

The range of platforms performing DVFS is wide. It ranges from multicore server systems [28, 29] over general purpose systems [24, Introduction] to the embedded, low power systems [30, 31, 25, 32, 33, 34, 35, 36].

On high performance platforms DVFS reduces power consumption due to increasing energy and cooling costs or affected chip reliability with high die temperatures [37]. Research regarding embedded DVFS ranges from real-time constraints [30, 32, 21, 38, 39, 31, 40, 33] to energy harvesting and battery life constraints [41, 12, 42, 43, 9, 11]. The constraints heavily influence the behavior of applied algorithms, for example with real-time constraints with lower frequency a higher execution time is to be expected, which conflicts with task deadlines. For energy harvesting constraints the performance is adjusted to the remaining battery life to keep the system functional [44].

What are requirements of DVFS?

To perform the scaling of voltage hardware-wise, software programmable DC-DC converters are needed [11]. Scaling frequency requires multipliers and dividers that can manipulate the clock signal [10].

Frequency scaling is more complex, as MCUs are built and managed through a clock tree. Manipulating certain components of the clock tree may require complex transitions as many system components are dependent on a certain clock tree node. The complexity further rises if the frequency has to be configurable at runtime [10].

Regarding different MCUs, clock trees can be very different and unique. The usage of a more generic architecture is preferable like Intel's SpeedStep for desktop or laptop architectures [45], or in this thesis ScaleClock, the work of Rottleuthner et al. [10] for constrained embedded devices.

What are limitations of DVFS?

The time overhead to perform voltage or frequency scaling is different. As described in [10], the time overhead of voltage scaling is relatively static in the order of several µs per 10mV. The time overhead of frequency scaling ranges from being instantaneous (e.g., when switching a mux) to multiple ms (e.g., cold-starting an oscillator).

Further, apart from energy and time overhead, the software enabling the clock tree transition or voltage adjustments also incorporates ROM overhead. This is especially the case when using more generic solutions, as it is the case with ScaleClock [10, sec. 2.4].

Where energy efficiency can be improved

The CPU performance can be limited by memory bandwidth rather than CPU speed. In those cases, a frequency reduction will have a small effect on CPU performance, but reduce energy consumption, since memory performance is not affected by a change in CPU frequency. This phenomenon is called *sub-linear performance slowdown*, which can be exploited by reducing the CPU performance when the *CPU-boundedness* of a code region is low [23].

Further, energy efficiency can be improved in situations where the CPU is idling for long periods of time and a higher CPU performance is only required occasionally. This is especially the case for IoT applications that have low duty cycle processing [35].

Rottleuthner et al. [10] state, a region of high CPU utilization is dominated by instructions requiring mostly access to CPU registers or Random-Access Memory (RAM). Regions that are limited by slow peripherals, I/O access or other asynchronous interactions have a lower CPU utilization.

More generally, energy is wasted whenever there is "a mismatch between the hardware configuration of the CPU and the utilization of the software [10]".

Indicators of Software Utilization

CPU utilization knowledge can be based on off-line source code profiling that quantifies regions of CPU or memory activity coupled with compiler transformations to adjust the CPU frequency at runtime [46, 22]. The need for source code and compiler support makes these approaches less practical. As input data sets might change at runtime and result in different program behavior, energy savings are limited [23].

Another group of approaches bases their evaluation on past program activity at runtime. The calculation of the CPU utilization has been primary based on the ratio of processor idle to busy time in a given time interval [14, 47, 48, 49, 25]. But this does not consider tasks appearing to utilize the CPU that are in fact limited by operations that do not require high performance. Rottleuthner et al. [10] therefore base their performance utilization metric on the context switching count and average CPU-time measured at different clock frequencies to also detect situations where the CPU might be bottlenecked by low performance operations.

Further research bases their CPU utilization on Performance Monitoring Units (PMUs) that are included in many modern processors [15, 50, 51, 38, 52]. PMUs count events that occur within different components of the processor like cache hit/miss ratio [38] or memory access counts [52]. They enable the knowledge on hardware resource usage and are used to detect performance bottlenecks or detect memory related performance issues. Beneficial to the usage of PMUs is that the reading of counters does not imply a high performance overhead for the CPU [52].

2.3 Debug and Trace Features with Arm Cortex-M

The IoT operating system RIOT [16] offers the module ScaleClock [17], which implements a generic interface to alter the clock tree to change the system clock frequency and to alter the system voltage.

At the time of writing this thesis, two boards were already supported to be used with ScaleClock, the STM32 Nucleo-64 development board with STM32L476RGT6 MCU [18] and the SLSTK3402A starter kit with the EFM32PG12B500F1024GL125 MCU [19]. Both MCUs are based on the ARM Cortex-M4 CPU, which is part of the Microcontroller Profile of Version 7 of the Arm Architecture (ARMv7-M) [3]. Even though they do not specify hardware counters that are intended to be used at runtime, the ARMv7-M specifies non-invasive debug components that trace performance statistics, data access or even instruction fetching [4, sec. C1.1].

2.3.1 Common Use

Usually the debug and trace features are intended to help debugging source code during development. For that, the developer has to connect their PC to the board via a debug adaptor. This enables communication with the debug and trace modules of the CPU over JTAG or Serial Wire Debug (SWD), as seen in Figure 2.4.

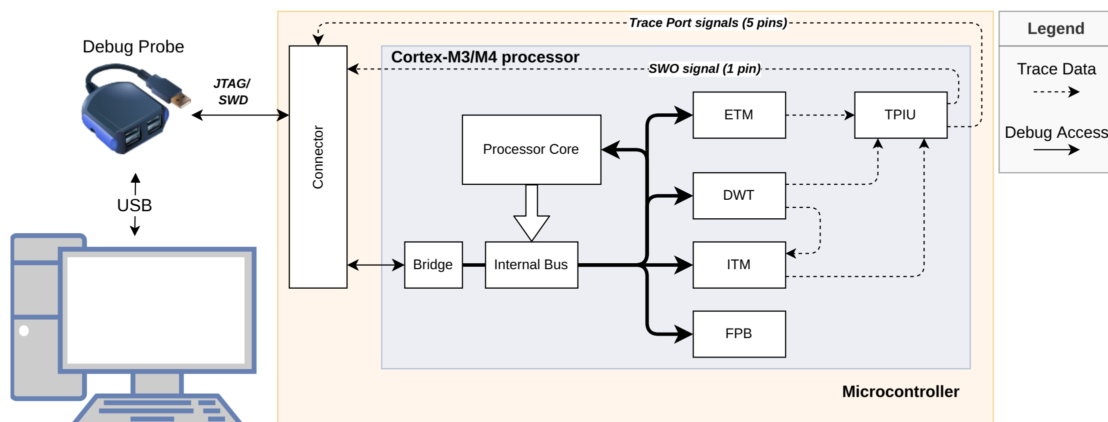


Figure 2.4: Overview of trace and debug connections in the MCU and the connection to the host PC, based on [3, 4].

The trace features are typically used to analyze code for coverage or performance and can be visualized in several ways. One example is the Keil µVison debugger, which is part of the Keil Microcontroller Development Kit [53]. There is a difference between debug and trace features, which will be examined in the following sections [3].

2.3.2 Debug Features

A debug connection allows an external debugger to:

- access debug/ trace feature registers
- access core registers (only when the processor is halted)
- access the memory map (possible while the processor is running)
- add breakpoints via the Flash Patch and Breakpoint Unit (FPB)

The debug connection is handled by 2 different connection types. Those are the industry standard JTAG with 4-5 pins and the newer SWD with only 2 required pins but which handles all features of JTAG [3].

2.3.3 Trace Features

A "trace connection allows an external debugger to collect information about program execution in real time (with a small delay) during program execution [3, sec. 6.3]".

The information collected could be from [3]:

- **Data Watchpoint and Trace unit (DWT)** - enables data trace of selected memory address ranges and profiling trace with counters that track the number of cycles the CPU used in different operations
- **Embedded Trace Macrocell (ETM)** - enables to access the instruction execution history
- **Instrumentation Trace Macrocell (ITM)** - enables applications to send logging or event words to the debugger (via the Trace Port Interface Unit (TPIU)) and provides control of timestamp packets

To provide external visibility to the debugger for these 3 described modules, a MCU implementation typically includes a TPIU. The trace connection of the TPIU can be handled by 2 different connection types [4, sec. C1.10]:

- **Serial Wire Output (SWO)** - provides asynchronous trace of DWT and ITM (single pin)
- **Trace Port** - provides high bandwidth parallel trace port and supports trace of DWT, ITM and ETM (multi data path pins, clock pin, optional control pin)

Nevertheless, first the processor family and more important the MCU implementation determines whether certain debug or trace components are implemented [4]. Therefore, this can only be determined by looking at the MCU specific reference manuals like for the STM32L476RGT6 [5] or the EFM32PG12B500F1024GL125 [54].

3 Scope of Thesis

3.1 Potential Performance-Controlling Approach

Figure 3.1 shows a schematic of a potential module that optimizes a system for energy efficiency by controlling the CPU performance. This schematic has been designed to give an outlook on what already might be possible on modern processors with dedicated hardware performance counter. This thesis evaluates the usability and overhead of certain debug components in terms of similar usage to hardware performance counter.

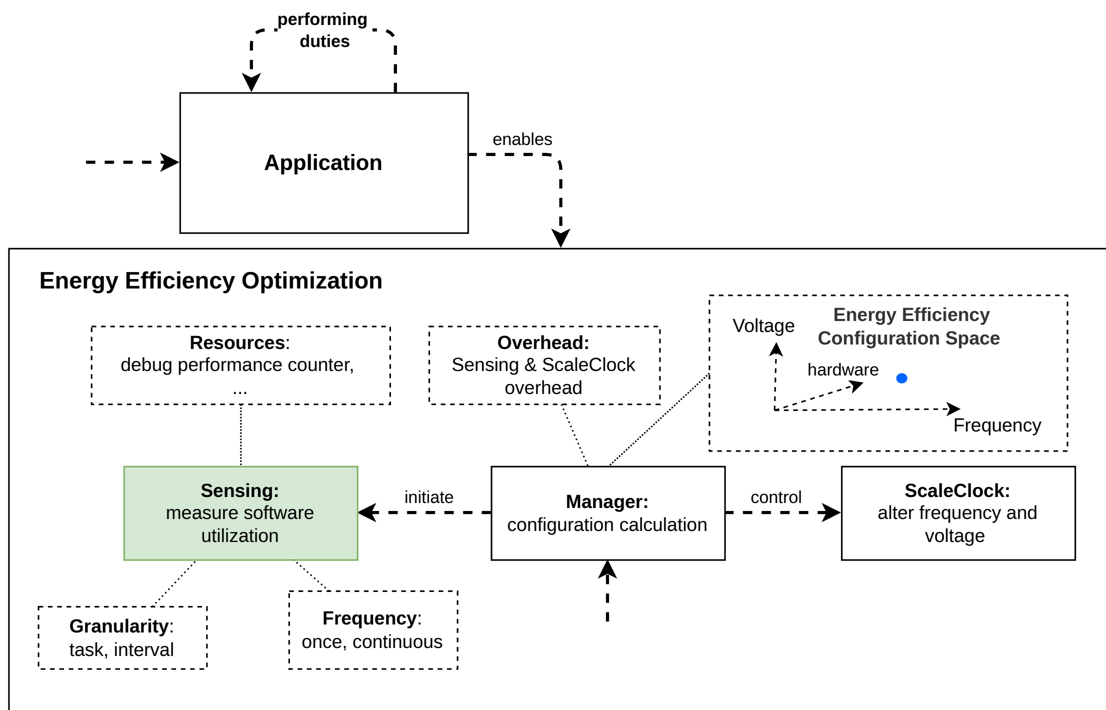


Figure 3.1: High-level schematic of a potential performance-controlling approach.

Shown is a schematic of a module that enables optimization for energy efficiency by applying DVFS. The module can be enabled by an application and performs measurements of the application state. The measurements are based on resources that are suitable to assess the software utilization like, e.g., performance counters. The manager module further has to decide at which granularity and frequency these measurements need to be performed. With the gained knowledge, a calculation estimates the most energy efficient voltage and frequency setting for the currently running tasks (blue dot). The calculation should also consider the overhead of measuring the software utilization and the upcoming time and energy overhead performing DVFS.

3.2 Goals and Requirements

The focus of this thesis is to design a way of measuring task properties with debug or trace components at runtime with the aim of realizing a similar module to Figure 3.1 in the future. Using the measured task properties as an indicator for performance utilization will be evaluated. The evaluation should be based on an appropriate set of tasks. With that in mind a list of requirements is formed as:

No.	Description	Importance
1	**Evaluate Usability of Debug and Trace features to trace task properties at runtime**	High
1.1	Not invasive in terms of CPU load and program execution	High
1.2	Retrieval of task properties without task customization or prior task knowledge	High
1.3	Task properties potential usable as indicators of performance utilization	High
2	**Design and Implementation of a model that enables to trace task properties with the debug and trace features**	High
2.1	Preserve requirement 1.1	High
2.2	Simple deployment by other users without special hardware	High
2.3	Assessment of overhead in time and energy	High
2.4	Comprehensive and modular software API	Medium
3	**Select a set of tasks to evaluate the implemented model**	High
3.1	Task features that expose different energy consumption (e.g., via I/O tasks, flash or RAM memory access)	High
3.2	It should be representative for embedded use	Medium
4	**Evaluation of the traced task properties as indicators for performance utilization**	High
4.1	Which task behavior and energy consumption characteristics are observable?	High
4.2	Which sensed task properties point to a lower Most Energy Efficient CPU Frequency (MEECF) setting?	High
4.3	What is the overhead in terms of tracing steps for selected task properties?	Medium

Table 3.1: Thesis goals and requirements.

4 Debug and Trace Feature Evaluation for Usability

In the following, the trace features of the ARMv7-M will be evaluated regarding their usability to trace task properties at runtime. The evaluation adheres to the requirements of goal 1. of Table 3.1. Thereby, the task properties should potentially be usable as indicators of performance utilization.

Debug features like halting the processor via the Flash Patch and Breakpoint Unit (FPB) will not be considered, as they stop the program execution (see 1.1 of Table 3.1). The FPB also allows remapping specific instruction addresses from the code region of system memory to addresses in the SRAM region [4, sec. C1.11]. This could potentially be used to instrument applications to retrieve specific task properties, but this violates with requirement 1.2 (see Table 3.1).

4.1 Data Watchpoint and Trace unit (DWT)

The Data Watchpoint and Trace unit (DWT) provides non-intrusive profiling counters. The DWT comparators enable Data Address Matching (DAM) and Instruction Address Matching (IAM) [4, C1.8]. In the following, these features will be examined in more detail.

4.1.1 Profiling Counters

Table 4.1 shows the available profiling counters. Each counter tracks different CPU behavior and is readable/ writable by software via memory-mapped registers. Except the *CYC* counter, each counter has a size of 8 Bit [4, sec. C1.8.7].

Counter Type	Description	Size (Bit)
CPI	Counts additional cycles required to execute multi-cycle instructions, except those recorded by LSU counter, and counts any instruction fetch stalls.	8
EX	Counts the total cycles spent in exception processing.	8
SLEEP	Counts the total number of cycles that the processor is sleeping.	8
LSU	Increments on any additional cycles required to execute load or store instructions.	8
FOLD	Increments on each instruction that takes 0 cycles.	8
CYC	Increments on each processor clock cycle.	32

Table 4.1: Description of available profiling counters and CYC counter of the DWT, based on [4, 8].

"The profiling counter size and the DWT event generation model are designed for non-intrusive operation, where the DWT generates information for remote tracing, without the system overhead of software reads and processing by the processor itself [4, sec.C1.8.4]". Thereby, if the counters are remotely traced, they preserve the requirement 1.1 of Table 3.1. Remote tracing is the case when using a debug probe as seen in Figure 2.4.

The counters can be individually enabled by the *DWT_CTRL* register. If a counter reaches the value 256 it overflows to 0 and generates an Event Counter Packet (ECP). This 2-Byte packet is part of a set of DWT hardware source packets and its format is described in Figure 4.1 [4, sec.C1.8.4].

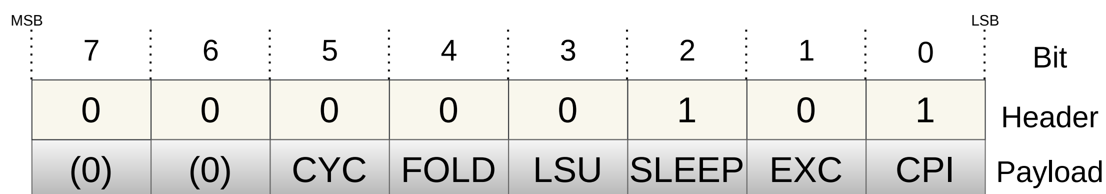


Figure 4.1: Format of an Event Counter Packet, based on [4, sec. D.3.1].

Every overflowed counter can be represented in the payload by an appropriate flag set to 1. The DWT can generate packets with multiple counter bits set to 1, indicating a

combination of counters wrapping to zero, if multiple counters are active and overflow at the same time. This reduces the number of packets that have to be generated [4, sec. D.3.1].

Regarding the accuracy, the ARMV7-M Architecture Reference Manual describes that "the counters provide approximately accurate performance count information, but the architecture accepts a reasonable degree of inaccuracy in the counts [4, sec. C1.8.4]".

Regarding the usability for performance utilization, the profiling counters are similar to hardware performance counters as mentioned in Section 2.2.2 and therefore very interesting to be evaluated with actual tasks. The *SLEEP* counter could be used to detect applications that have a low duty cycle. The *LSU* counter might give insights on the frequency of memory accesses, which might detect slow peripherals, as mentioned in Section 2.2.2. Nonetheless, the downside is that they are hard to be accessed at runtime by their memory-mapping, as a counter overflow happens every 256 counter cycles. Even though the overflow generates a packet, these are not tracked anywhere else with a further counter.

Executed Instructions

Implementations that do not have access to the Embedded Trace Macrocell (ETM) module, where every instruction is reported in the exported stream, a combination of the profiling counters can be used to calculate the total executed instructions [55]. This combination is seen in Equation 4.1:

$$INSTR_{executed} = CYC_{CNT} - CPI_{CNT} - EX_{CNT} - SLEEP_{CNT} - LSU_{CNT} + FOLD_{CNT} \tag{4.1}$$

The calculated total executed instructions might also be used in combination with the cycle counter to create the ratio *cycles per instructions.* This ratio and further dedicated hardware counters have been used by Choi et al. [51] to create DVFS technique, which saves up to 70% energy for memory bound programs on a XScale platform.

4.1.2 Comparators

Dependent on the MCU implementation, a DWT can include up to 15 comparators. A comparator is defined by three different registers (*DWT_COMP, DWT_FUNCTION, DWT_MASK*). The DWT_COMP register is used to compare the held address with one of the following CPU activities [4, sec. C.1.8.1]:

- access to an instruction address
- access to data address
- access to a data value
- the cycle count value

For address range comparison the DWT_MASK register defines how many addresses the comparator observes. The register DWT_FUNCTION defines the type of CPU activity that is observed. It also defines which type of access (read, write, both) should be tracked and which event or Data Trace Packet (DTP) is generated on a successful comparator address match [4, sec. C.1.8.1]. A DTP is also part of the set of DWT hardware source packets [4, sec. D.3].

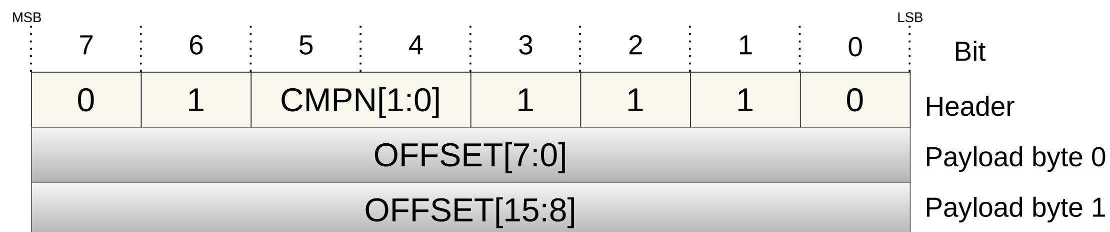


Figure 4.2: Format of a Data Trace Address Offset Packet (DTAOP), which is a type of DTPs. It is dependent on the comparator ID and the offset of the matched address. Based on [4, sec. D.3.4].

There is a difference between packet and event generation. Packets are forward from the DWT to the Instrumentation Trace Macrocell (ITM), while *CMPMATCH[N]* events are only forwarded to the ETM [4, sec. C.1.8.1]. When the CPU accesses a data structure from memory and an address match occurs, either a *CMPMATCH[N]* event or a DTP (see Figure 4.2) can be generated. When the CPU, for example, loads an instruction from flash and an address match occurs, only the generation of an *CMPMATCH[N]* event is permitted [4, table. C1-21]. The reason for this constraint has not been described in the reference manuals but could be related to the higher frequency of instruction fetches

compared to data accesses. While DTPs can be serialized and output via the SWO line, packets generated by the ETM are only accessible via the trace port connection, which offers higher bandwidth than the SWO line [4, sec. C1.3]. Therefore, boards that prevent access to the trace port connection also lose the opportunity of tracing Instruction Address Matching (IAM).

Regarding usability for performance utilization, comparing for a single data value or the cycle count value is not considered to be useful. Data value comparison can only observe one address per comparator and would predefine application knowledge (see Table 3.1). The cycle count value is redundant to the already existing *CYCCNT* that has a high bit size of 32 Bit and can be read directly by the software. A match for a data or instruction address range might be useful to analyze the access to RAM, flash or peripherals via their memory addresses. But the comparator matching only generates packets or events that are not tracked by any counter, which means that this information is not directly accessible by the microcontroller software.

4.2 Embedded Trace Macrocell (ETM)

As already mentioned, the ETM module accepts *CMPMATCH[N]* events as a trigger input. Thereby, it enables to trace IAM and more advanced address matching filtering [56, sec. 2.6].

Cortex-M4 CoreSight ETM implementations [57] only accept input from the DWT comparators, other specifications include their own comparators that also trace data addresses. The ETM can also include sequencers to enable more complex multi-stage trigger schemes [56, sec. 2.2.3]. Packets transmitted by the ETM are only put out over the parallel trace port of the TPIU [4, sec. C1.3].

As the ETM enables to match against access for a range of instruction addresses, it might be useful to detect memory bottlenecks of the flash memory. Unfortunately, the IAM is not coupled with any counter to be read by the software but needs even stricter requirements for the TPIU to sent packets to a debug probe.

4.3 Instrumentation Trace Macrocell (ITM)

The ITM can on one side be used as a memory-mapped register interface to enable application logging to a trace sink, like the TPIU. Furthermore, it controls the generation of timestamps or synchronization packets and merges them with events of other trace components into a single trace stream [4, sec. C1.7.1] (see Figure 4.3).

Regarding usability for performance utilization, the application logging could be useful to signal the start of certain application parts. For example, application parts that are known to not fully use the CPU performance like data persisting via slow peripherals. Unfortunately, the generated packets are not tracked by any counter and are only sent to the TPIU for remote tracing use. Furthermore, the application logging requires inserting certain commands into the application code to write to the ITM registers, which conflicts with the requirement 1.2 of Table 3.1.

4.4 Trace Packet Path and Trace Port Interface Unit (TPIU)

Figure 4.3 shows the relation and packet path between the trace feature components.

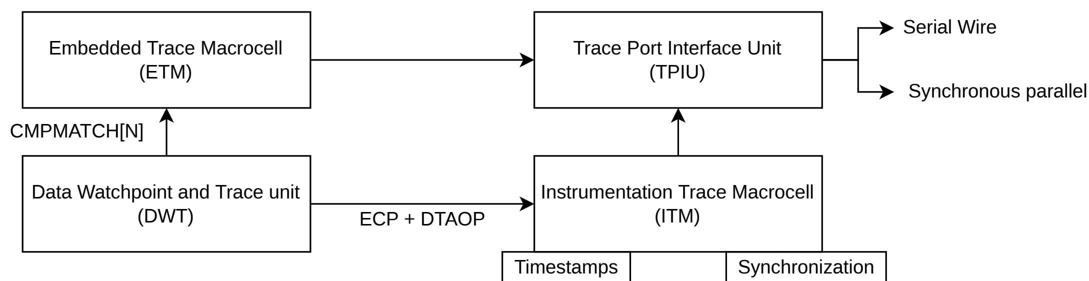


Figure 4.3: Relation of the ARMv7-M trace components and the path of different packet streams, based on [4, sec. C1.7.1].

The DWT sends a trace packet in the case of an overflow of a profiling counter (ECP) or a comparator data address match (DTAOP) to the ITM which in turn can send packets to any suitable trace sink. The ITM merges packets from the ITM (local/ global timestamp packets or synchronization packets) and DWT (source packets) and forwards them to the TPIU as a single data stream. The TPIU merges this data stream with the data

from the ETM (if the ETM is enabled) and provides external visibility through a trace interface connection (see 2.3.2). An asynchronous SWO or synchronous parallel trace port is provided [4, sec. C1.7]. An output path for the packet stream from the ITM is minimally supported by the TPIU support for the ARMv7-M [4, sec. C1.10].

4.5 Summary of Feature Evaluation

Table 4.2 shows a summary of the debug and trace features of the Arm Cortem-M4 processor and their usability as resource for obtaining characteristic task properties at runtime (as described in the prior subsections):

Feature	**Potential usability as indicator of performance utilization**	**Constraint**
FPB - Remapping & Breakpoints	None	halts the processor, needs task specific address knowledge
DWT Profiling Counters	High	8 Bit per counter, except CYCCNT
DWT Comparators - Data Address Matching (DAM)	High	Not directly accessible via software
ITM - Application Logging	Low	Not directly accessible via software
ETM - Instruction Address Matching (IAM)	Medium	Not directly accessible via software

Table 4.2: Summary of debug and trace features as potential indicators of performance utilization and constraints for usability at runtime.

5 Concept of Task Characterization Model

The term task characterization model describes a component that dispenses task properties for a given task by tracing the task. It uses resources like, for example, cortex-M trace features to trace task properties. The resources are made accessible with a feedback mechanism of some kind, for example with a timer counter. Software defines the start- and endpoint of a trace. The aim of the model is to capture task properties that indicate the performance utilization of the traced task.

The term task characterization describes the process to trace a set of task properties for a given task with the task characterization model.

The concept of the task characterization model in this thesis uses the Cortex-M trace features as a resource and a timer counter as feedback mechanism. Thereby, with the usage of the term task characterization model this specific model is meant.

5.1 Accessibility Discussion of Trace Features

In the following a handful of possible ways to get access to the profiling counters (see 4.1) or comparator address matching at runtime by the firmware will be discussed. Thereby, the realization cost and the level of invasiveness in terms of CPU load will be examined.

5.1.1 Register Polling

As it is possible to directly read the profiling counter registers, a trivial idea might be to periodically check the count of each interesting profiling counter. This procedure has a low realization cost but tracing a longer period of time without missing any counter information means reading any profiling counter every 256 cycles continuously.

To implement this kind of behavior without instrumenting the application software directly would be achievable in two ways. First, by a concurrent thread that polls the cycle count for n (n $<$ 256) cycles and then reads the DWT profiling counter. Secondly, by a timer that is instrumented to generate an interrupt every n (n $<$ 256) cycles, which further triggers an Interrupt Service Routine (ISR) and checks the DWT registers. Therefore, the realization is considerably small.

Regarding the level of invasiveness, this very frequent reading of the DWT registers by polling or interrupts generates a very intrusive situation. In a system that has multi-cores and uses polling, a single core would spend all of its execution time for a thread reading the profiling registers but preserves the other cores from polling. Still, this approach violates with requirement 2.1 of Table 3.1 and the Cortex-M4 is only equipped with a single core. The interrupt approach avoids polling, but the context-switch overhead, in terms of pushing/popping registers, interferes heavily with the normal application execution. Therefore, in both cases the non-invasive DWT property would be destroyed.

Furthermore, the CPU processing power is not only wasted by continuously polling or plagued by the interrupt overhead, but the named approaches probably also skew the measurements of the counters and affect the accuracy. This would disguise the actual benefit of using the profiling counters. Lastly, this approach would only enable to use the profiling performance counters. The access to comparator address matching would still need to be enabled on an additional way.

5.1.2 Deserialization

Another approach would be to use the available generation of trace packets and to deserialize the packets in other ways than with a debug probe. This would enable to use both the profiling counters and the generated packets of the comparators.

Similar to a debug probe, an external hardware shield could be designed, which deserializes the ECPs/DTP that are being output by the TPIU. It would further consist of counters bigger than 8 Bit for each profiling counter/comparator matching. To access the designed counters by the application, the external hardware would need to be accessible via a serial communication interface like Serial Peripheral Interface (SPI).

A positive aspect of this approach is that the deserialization process is completely decoupled from the microcontroller CPU, preserving the non-invasiveness property. Further-

more, querying the profiling counter values via a peripheral communication interface is probably less invasive than the *Register Polling* approach discussed above.

Without considering the time overhead generated by the deserialization and peripheral communication, or the additional power consumption incurred by powering the external shield, this approach has some significant limitations regarding realization cost. First of all, this piece of hardware does not exist yet. Secondly, designing an external shield is a huge effort for a potential low reward, considering the trace features of Section 4 are not usable for performance utilization tracings. Before investing this effort, the usability should be proved first.

Further, usability-wise after designing and producing this piece of hardware, an outside user would need to buy or build extra hardware to potentially save energy for their applications.

5.1.3 Counting of Trace Packet Flanks

While the deserialization approach has the benefit of translating all information hidden in the encoded hardware source packets, the question has to be asked whether this is actually necessary. As the event generation for each profiling counter is individually configurable, and known by the firmware, just the presence of a single ECP/ DTP shows whether an overflow or match has occurred.

Therefore, given the trace packet stream of the deserialization approach without the need to actually translate the encoding of a packet, enables to simply forward the packets into a unit that only counts the flanks of an analog hardware source packet.

Regarding the realization cost, on many boards this can be achieved via a peripheral timer module that is configured as a counter with an external clock source. The external clock source of the counter is a channel that is connected to a GPIO pin. With signals arriving to the GPIO pin, the counter counts up or down when a rising or falling flank is detected. In hardware, a cable needs to forward the output signal of the TPIU to a GPIO pin of the timer counter. Software-wise, the firmware only has to keep track on the enabled profiling counter, which then can be associated with the timer counter.

The downside of this approach is that an ECP can represent multiple counter overflows at the same time. To count any type of trace feature requires enabling only this one

counter type at a time and has the downside of exclusively using the DWT profiling counters or the comparator matching.

Regarding the level of invasiveness, the only thing that affects the CPU operation is the timer counter register read. This should only be necessary very infrequently when choosing a timer with a bigger counter than 8 Bit. Gandraß et al. [58] performed a survey for timer peripherals on MCUs. It covers 43 MCU device families, which are supported by RIOT. Thereby, they identified that all platforms provide at least 16 Bit timers, 90% of all platforms provide a counter width of 32 Bit. Furthermore, 92% of all timers can be driven by one external clock source.

To conclude, as this approach should preserve the non-invasiveness of the CPU operation and produces a low realization cost compared to the deserialization approach, it will be realized in the following sections.

5.2 Feedback Mechanism with Timer Counter

While the trace packets are usually sent to a debug adapter to be viewed by dedicated Desktop software tools, the approach is to simply count the outgoing ECPs/ DTPs packet flanks by a timer counter and read the memory-mapped timer counter register with the application firmware.

Figure 5.1 shows the basic idea of flank counting, as a packet is sent over the line as a signal with rising and falling flanks.

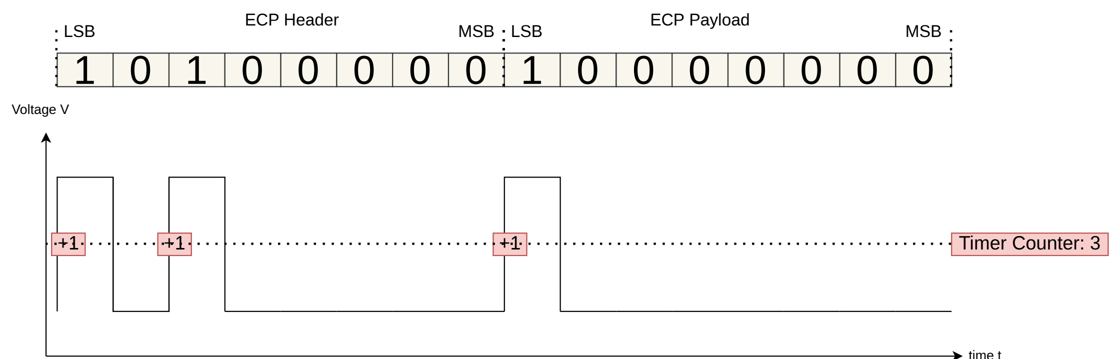


Figure 5.1: Exemplary CPI overflow ECP packet represented as a signal with flanks and the reading with a timer counter.

To achieve this signal reading, the output of the TPIU, which is externally accessible by a GPIO pin, simply needs to be connected to the input of a timer counter, as seen in 5.2.

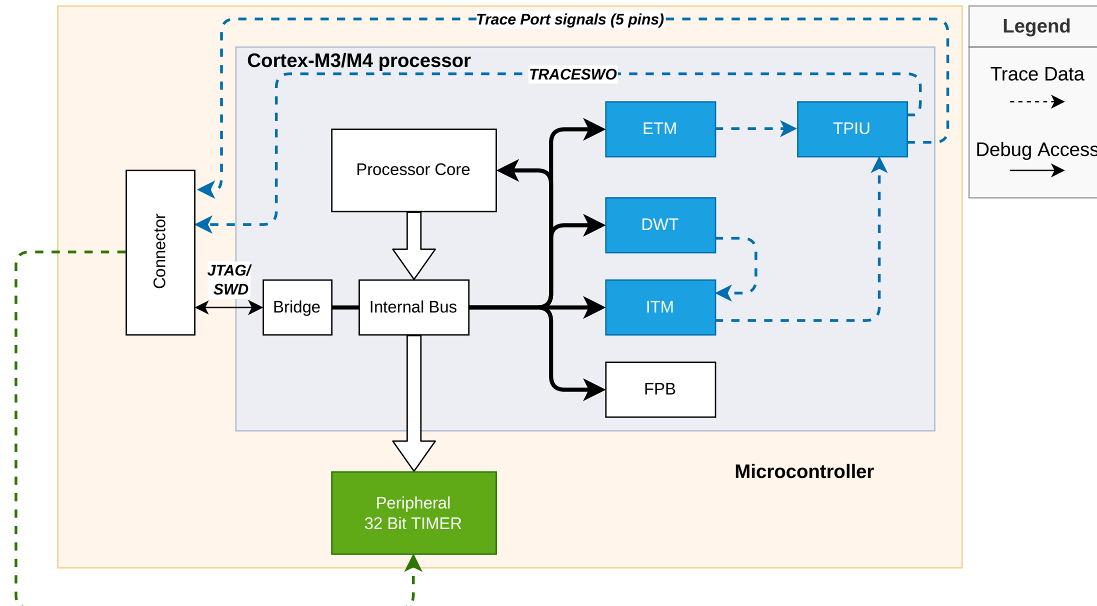


Figure 5.2: Existing connection of relevant trace components (blue) and additional connection of components that enable the feedback of the trace packets (green).

This feedback mechanism enables to access the profiling counters, Data Address Matching (DAM) and Instruction Address Matching (IAM) features independent of the TPIU port (trace port or SWO). The concept is applicable for many cortex-M processor (see Section 5.4). The presented mechanism might also be used with trace features of other processors that were designed to be used with a debugger.

Size of Timer Counter

A profiling counter overflow event stands for a counter that was incremented 256 times. The comparator matching approach does not have a granularity reduction but generates a DTP every time an address match occurs.

Therefore, the timer counter size should preferably be bigger than 8 or 16 Bit. The bigger size on one hand maximizes the timespan before the timer counter should be read. On

the other hand, because the timer counter logic is as simply as reading the flanks of a packet on the trace line, any ECP or DTP increments the timer counter by a number that is similar to the packet byte size (see formats in Figure 4.1 and 4.2).

5.3 Concept of Software Trace

Software-wise a task or code section should be able to be measured with different profiling counters or configured comparator address ranges. Prior to starting a trace, the trace components (DWT, ITM, TPIU) should be enabled. To save energy, it should also be possible to disable the trace components. Figure 5.3 shows the trace concept at an exemplary sequence of a *LSU* trace.

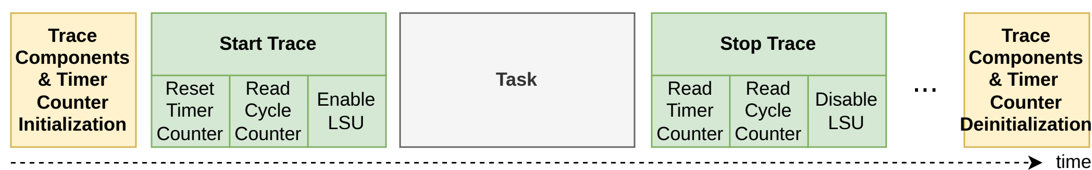


Figure 5.3: Procedure concept of tracing a task with a profiling counter. Exemplary measuring the *LSU* counter is shown.

This trace concept should be able to be accomplished with a user-friendly interface by a single component.

Figure 5.4 shows the software components of the task characterization model conforming to separation of concern.

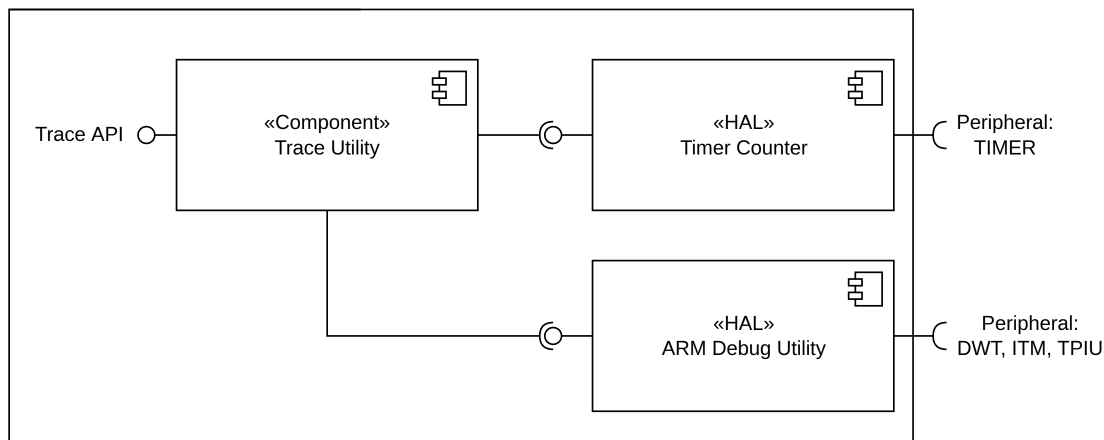


Figure 5.4: Component diagram of software components for the task characterization model.

Trace Utility This component is concerned about offering an application-focused interface to initialize/start/stop a task trace. Therefore, it needs access to the *Arm Debug Utility* and the *Timer Counter* hardware abstraction layers.

Timer Counter This component abstracts the memory-mapped timer counter register interface to comprehensive functions specifically for trace usage. Further, the focus on the separation of concern enables to easily adjust the implementation to timers of different boards without modifying other components.

ARM Debug Utility This component focuses on the abstraction of memory-mapped debug unit registers of the DWT, ITM and TPIU. It is mainly responsible for the initialization of the packet generation and ensures the debug component collaboration.

5.4 Spread of Concept Use

Regarding wider usability of the task characterization model, Table 5.1 points out which cortex-M CPUs also support the evaluated trace features of Section 4.

PLATFORM FEATURE\	Cortex-M0/M1	Cortex-M0+	Cortex-M3/M4	Cortex-M7	Cortex-M23	Cortex-M33
Architecture	Armv6-M	Armv6-M	Armv7-M/ Armv7E-M	Armv7E-M	Armv8-M Baseline	Armv8-M Mainline
Protocol of trace connection	-	-	Trace port/ Serial Wire Viewer	Trace port/ Serial Wire Viewer	Trace port/ Serial Wire Viewer	Trace port/ Serial Wire Viewer
Profiling Counter Trace (using DWT)	-	-	Yes	Yes	-	Yes
Selective Data Trace (using DWT)	-	-	Yes	Yes	-	Yes
Instruction Trace (using ETM)	-	-	Yes	Yes	Yes	Yes
DWT Comparators	Up to 2	Up to 2	Up to 4	Up to 4	Up to 4	Up to 4

Table 5.1: Supported trace features of Cortex-M processors. Table based on [3, table 9].

As shown by Table 5.1, the Cortex M3, M4, M7 and M33 support the DWT profiling counters, data address matching (DAM) and the instruction address matching (IAM) by the ETM module. Based on that, Figure 5.5 shows how many boards of RIOT use one of the mentioned CPUs and can thereby potentially be instrumented to obtain access to the trace features at runtime.

The data of Figure 5.5 has been collected by resolving the path from a board Kconfig file in directory *boards* to the used CPU Kconfig file defined in directory *cpu*. It is shown that the designed task characterization model can potentially be used by 49.23% of all RIOT boards.

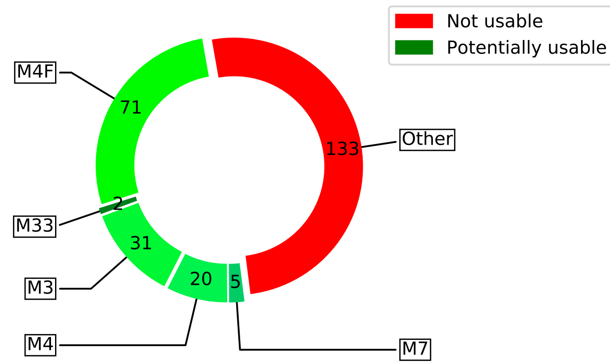


Figure 5.5: RIOT (2021.07 version) boards with Cortex-M processors that are potentially suitable for task characterization model. Boards analyzed via Kconfig via the *cpu* and *boards* directory.

6 Implementation of Task Characterization Model

6.1 Board Choice for Implementation

The choice of a board to implement the concept of Section 5 with, has been between the Nucleo-L476RG [18] and the SLSTK3402A [19] board, as they are already ported to the ScaleClock implementation. The Nucleo-L476RG board has been chosen, as it has ARDUINO Uno V3 connectivity support [6], which makes extending the functionality with specialized shields easy. It has been used by the ScaleClock paper for the networking case study [10], which increases the research base for this board. Furthermore, the Nucleo development board is six times cheaper to purchase than the SLSTK3402A.

This choice has been made at the beginning of this research, when only the profiling counters as a resource for obtaining task properties were sighted. Using the DWT comparators to track the access to RAM or flash has been discovered late in the research process. While the Nucleo-L476RG board is well suited for enabling the access to the DWT profiling counters and the DAM with comparators, it is not suited to track IAM with the DWT comparators. This limitation is caused by the package of the STM32L476RGT6 MCU, which does not provide access to the parallel trace port connection. But IAM only allows to generate *CMPMATCH[N]* events, which are only sent to the ETM module. The ETM only allows to output packets over the parallel trace port of the TPIU [4, sec. C1.3].

Regarding future work, the SLSTK3402A board with the EFM32PG12B500F1024GL125 MCU provides the DWT and ETM trace features [54]. Furthermore, the package of the MCU provides access to the trace connections SWO over the *DBG_SWO* alternate function and the parallel trace port connection over *ETM_TD0* and *ETM_TCLK* alternate functions [59].

6.2 Hardware Assembly on the Nucleo-L476RG

6.2.1 Requirements for Access to SWO Line

To get access to the trace data by the TPIU block on the STM32L476RGT6 MCU, the asynchronous SWO (here *TRACESWO*) port is used, which only requires a single line. The use of the parallel trace port is not possible with the package of the MCU, but it should also be possible to get access to trace data with MCUs that offer the trace port connection.

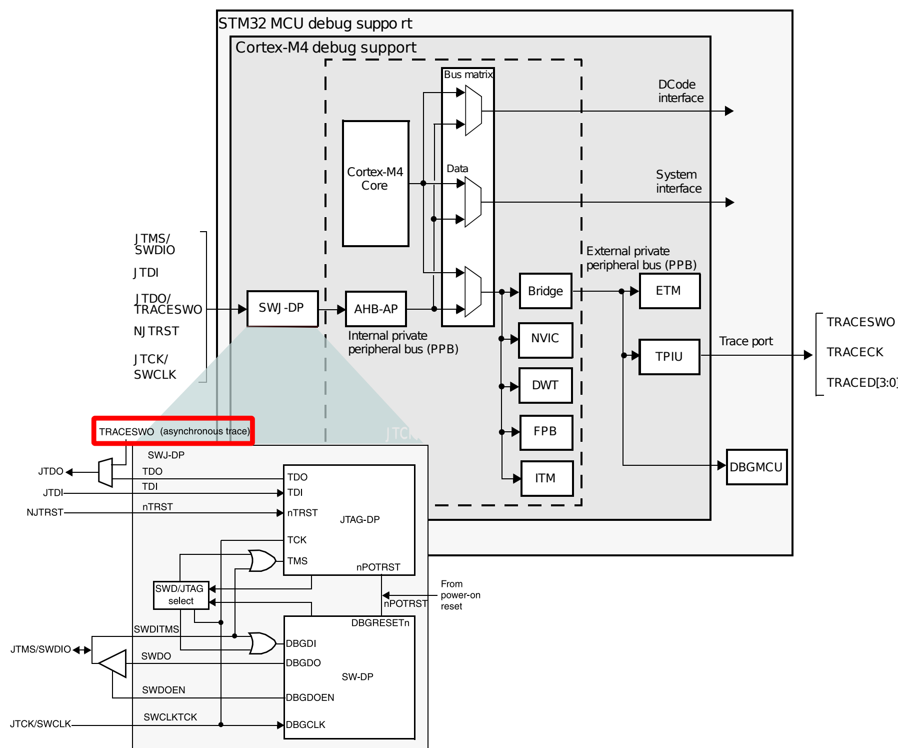


Figure 6.1: Debug component and connection overview with SWJ-DP zoomed in (left) for the STM32L476RGT6. Figure based on [5].

Figure 6.1 shows the bus connections between the system and debug components. Further, the component SWJ-DP is zoomed in. It should be noticed that there are two *TRACESWO* ports shown (left and right). The right one is only accessible over the trace port connection, which the LQFP64 package of the STM32L476RGT6 does not provide [60, sec. 4 Pinouts and pin description].

The left *TRACESWO* line is connected to the SWJ-DP and multiplexed with the *JTDO* line. Therefore, the asynchronous trace can only be used with Serial Wire Debug Port enabled and not JTAG Debug Port. By default, the JTAG Debug Port is active. If the external debugger host wants to switch to Serial Wire Debug Port, it must provide a dedicated JTAG sequence on the *JTMS/SWDIO* and *JTCK/SWCLK* lines. This sequence can be seen at [5, sec. 48.3.1].

6.2.2 Wiring for Feedback Mechanism

Figure 6.2 shows the low hardware overhead to enable the feedback mechanism of reading the flanks of debug trace packets with a timer counter. Thereby, only three cables and two 1k resistors are needed.

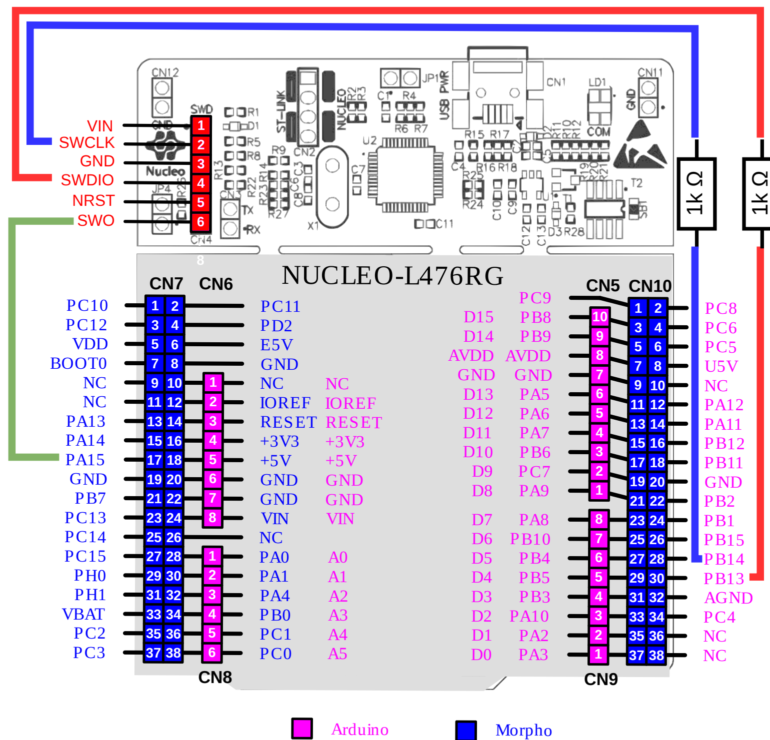


Figure 6.2: Connection of pins with cables on the Nucleo-L476RG board to count generated trace packets with the feedback mechanism. Figure based on [6].

The Nucleo-L476RG has a MCU part and ST-Link part by default (see Figure 6.2). When the ST-Link part is not cut, the SWD/JTAG pins are connected to the *CN4* connector (see red pins of Figure 6.2) [6].

The *TRACESWO* line, which is accessible via connector *CN4 pin 6*, is connected to the GPIO pin *PA15*, which is input to the timer counter via GPIO alternate functions [60]. The used wire should be as short as possible to reduce the resistance and parasitic capacitance of the wire, which reduces the fault susceptibility of the connection for data transfer at higher frequencies.

Usually an external debugger is needed to switch the debug port from JTAG mode to SWD to get access to the *TRACESWO* line [5, sec. 48.3.1]. But this can be avoided by sending the switch sequence over any synchronous bus interface, like for example SPI. Using a SPI module for the short JTAG sequence might not be possible for certain boards as they do not have a dedicated module or the module is are already occupied with other sensors. Therefore, the synchronous communication can be emulated with GPIO pins over bit-banging, which reduces the hardware requirements.

By connecting the GPIO pin *PB14* to *SWCLK/JTCK* and GPIO pin *PB13* to *SWDIO/JTMS*, the SWJ-DP can be switched to SWD mode by sending the sequence [5, sec. 48.3.1] at runtime via bit-banging. The two cable connections each have a resistor of 1k ohms inserted as a current limiter. This is a safety precaution as the board might switch each pin on both connection ends to mode *GPIO_ OUT* at reboot, which would create a short circuit without a resistor.

6.3 Software Implementation of Trace Utility

Figure 6.3 shows the class diagram of the trace utility.

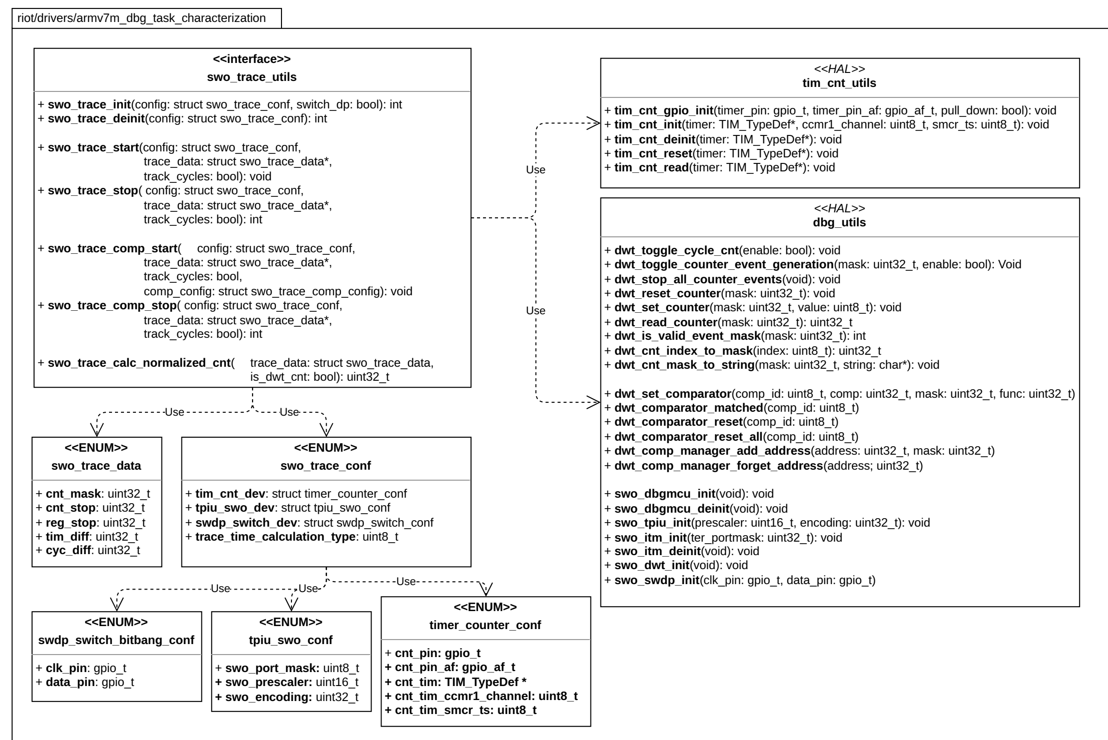


Figure 6.3: Class diagram of trace utility as designed in Figure 5.4.

In the following only the application-focused interface of the trace utility component, depicted in Figure 5.4, will be described further:

swo_trace_init / swo_trace_deinit Initializes/Deinitializes the debug units for packet generation and packet forwarding. It triggers switching the SWJ-DP and sets the timer counter up to start listening for packet flanks. The init and the following functions use a config struct as an input to simplify the usage.

swo_trace_start / swo_trace_stop Starts/Stops a trace using a selected performance counter, tracks the timer counter value dependent on the given timer device at the end of the trace and saves the trace data in the struct swo_trace_data.

swo_trace_start_comp / swo_trace_stop_comp Starts/Stops a trace using selected comparator address range. It tracks the timer counter value of the given timer device at the end of the trace and saves the trace data in the struct swo_trace_data.

swo_trace_calc_normalized_cnt Normalizes the counter readings to compare task properties between tasks with different trace lengths. The normalization will be explained in detail in Section 8.4.2.

6.3.1 Memory Coverage with Comparators on STM32L476RGT6

As already mentioned in Section 4.1.2, the comparators can be used to trace access to RAM or flash memory. Figure 6.4 visualizes the size of the flash and RAM memory in comparison to the total address range that can be observed on a single trace with the comparators of the STM32L476RGT6 MCU.

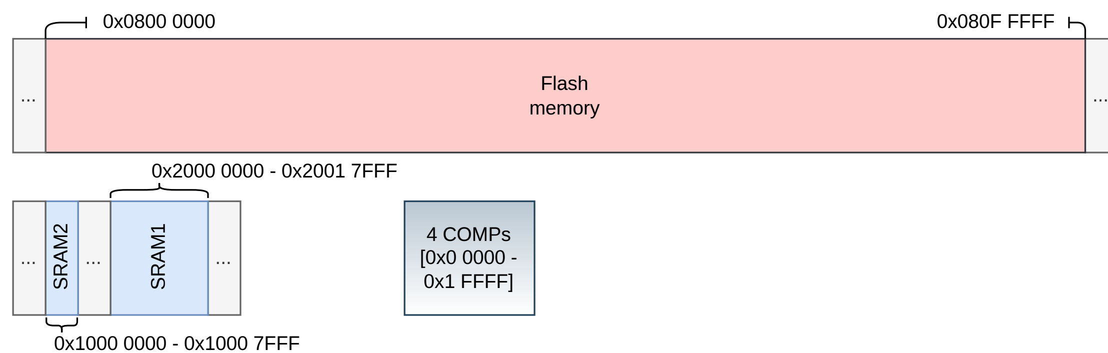


Figure 6.4: RAM and flash memory address space in comparison to the sum of all maskable address ranges of the comparators on the STM32L476RGT6 MCU.

The total address range of the DWT comparators that can be observed on a single trace depends on the maximum address mask size and the number of comparators. The number of comparators and the comparator mask size is dependent on the implementation of the MCU [4, sec. C1.8.1]. The number of comparator can be determined by the *NUMCOMP* field of the *DWT_CTRL* register [4, sec. C1.8.7]. The comparator mask size can be determined with the *DWT_MASK* register.

On the STM32L476RGT6 MCU four DWT comparators and a maximum mask size of 4 Bit per comparator are available. Together the comparators can compare a total address range of 0x20000 addresses at a time (each 0x8000 addresses), which enables to easily match against RAM usage in one trace. To cover the whole flash memory of 1MByte, eight different traces are needed.

The division into separate traces complicates the comparator tracing of the flash memory, a MCU implementation with more than four comparators or a bigger mask size per comparator could remedy these complications.

6.3.2 Peripheral Device Coverage with Comparators on STM32L476RGT6

The comparators can also be used to track the access to peripherals via their memory-mapping. This can be useful if many accesses to a certain peripheral are an indicator for a performance bottleneck. Table 6.1 lists a few exemplary devices that can be traced, compared with how many comparators are required to observe the complete address range of the device type.

For example, to fully trace the access of all Low Power Timers (LPTIM) minimally two comparators are needed. This is because the addresses mapped to the timer registers are not next to each other and the DWT comparators can only cover an address space with a start and an end point. More complex address matching with excluding areas in-between two addresses is only possible with comparators of the module ETM [56, sec. 2.6].

In Section 10 the peripheral SPI will be traced.

Peripheral Device	**Address Space**	**Required Comparators**	**Mask Size Per Comparator**
SPI 1 - 3	0x40013000-0x400133FF, 0x40003800-0x40003FFF	2	10, 11
LPTIM 1 - 2	0x40007C00-0x40007FFF, 0x40009400-0x400097FF	2	10, 10

Table 6.1: Address space and the number of comparators needed to trace the data access of different peripherals on the STM32L476RGT6 MCU.

6.4 Summary of Implementation constraints

The following table summarizes the tracing constraints of the implemented task characterization model over the SWO output connection:

Trace Method	**Constraint**	**Description**
Profiling counter	limited simultaneous tracing of different profiling counters	When multiple profiling counters are enabled to generate events, the DWT can merge multiple events into one packet.
Profiling counter & Comparator	no deserialization	Counting packet flanks makes different encoded packets in the same data stream indistinguishable. Only one trace source (profiling counters or comparators) at a time should be configured.
Comparator	total range of 0x20000 addresses observable per trace	The STM32L476RGT6 is only equipped with four comparators and a maximum address masking range of 0x8000 addresses per comparator

Table 6.2: Summary of tracing constraints with the Nucleo-L476RG board.

7 TPIU Configuration

Before taking measurements of the overhead and before tracing tasks with the task characterization model, the encoding of the packet transmission and the transmission speed of the TPIU should be configured.

Ideally the TPIU should be configured so that the transmission has the properties of a high accuracy (no packets are lost), low latency, high throughput, and a low energy overhead (the fewer flanks, the better).

7.1 TPIU encoding

The SWO transmission encoding should be configured as standard UART (NRZ) mode instead of the *Manchester* encoding. While both provide the same information of a single data packet transmitted, the *Manchester* encoding produces packets with 3 times more flanks (5 for *NRZ* and 15 for *Manchester* encoding with an ECP). A higher number of flanks per packet increases energy consumption of the packet transmission, increases the transmission time per packet (see Figure 7.1) and the timer counter is incremented more quickly.

Figure 7.1 shows the difference in transmission time of a packet encoded with *NRZ* mode (orange) and the same packet encoded with *Manchester* mode.

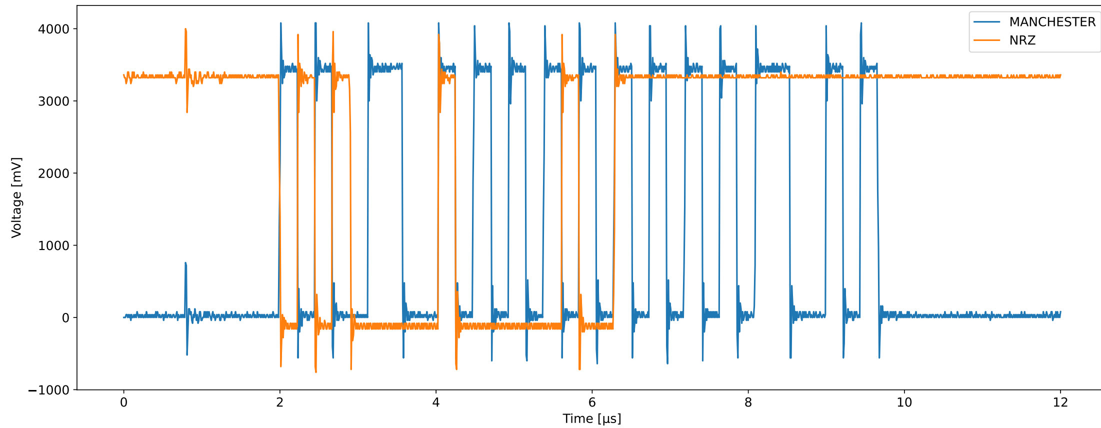


Figure 7.1: Comparison of CYCLE ECP signals between the *NRZ* and *Manchester* encoding. The signals are measured with an oscilloscope (see Section 8.2.3) at a CPU frequency of 13 MHz and a configured TPIU prescaler of 2.

7.2 TPIU prescaler

The TPIU prescaler divides the *TRACECLKIN* clock domain, which is the clock input of the TPIU, and therefore defines the data output speed. The higher the prescaler value, the slower the data is transmitted.

On the STM32L476RGT6 MCU, the *TRACECLKIN* connection is tied to the *HCLK* clock domain [5, p. 1862], which also clocks the CPU. With enabled *NRZ* mode, a configured prescaler value of 0 (*TPI_ ACPR*register) and a CPU frequency of 80 MHz, the time between two neighboring flanks was measured at 25ns. This stands for a transmission frequency of 40 MHz.

The choice of the prescaler configuration is on one hand constraint by the timer counter ability to recognize flanks at high signal frequencies. On the other hand, a slower TPIU output signal frequency might cause packet congestion in the TPIU with a high rate of generated ECPs/DTAOPs by the DWT.

Regarding the timer capability, the datasheet of the STM32L476RGT6 MCU [60, sec. 6.3.27 Timer characteristics] defines the maximum detectable timer channel input frequency as 40 MHz (f_{EXT}).

The packet congestion is dependent on the selected trace method. Profiling counters generate ECPs and comparators generate DTP. In the following, each trace method will be examined separately:

7.2.1 TPIU Prescaler for ECP

Regarding the congestion of packets, the highest frequency of generated ECPs can be calculated. An overflow of a single profiling counter is generated at the shortest timespan of 256 cycles and an ECP has a length of 16 Bit. Therefore, at a CPU frequency of 80MHz an overflow is generated at the highest frequency of 312.5 kHz and the TPIU would need 16 clock cycles to transmit a single ECP (see Equation 7.2). Meaning the TPIU prescaler should be low enough so that the TPIU output frequency is not smaller than 5 MHz for profiling counters. Therefore, with a prescaler of 0 (40MHz) or 1 (26MHz), a congestion should not be a problem with ECPs.

$$f_{maxECP} = \frac{1}{CYCLES} \cdot f_{CPU} \cdot Bits_{packet} \tag{7.1}$$

$$= \frac{1}{256} \cdot 80 \cdot 10^6 \cdot 16 = 312.5kHz \cdot 16Bit = 5 \cdot 10^6 Hz \tag{7.2}$$

To prevent a possible source of error with task tracing, an experiment (see Figure 7.2) was further performed to be certain about the minimal possible prescaler value for ECPs, as the maximum detectable timer channel input frequency is 40 MHz. The experiment involved configuring the DWT *POSTCNT* to a reset value of 4 and the *SYNCTAP* register to increment the *POSTCNT* every 64 cycles. This effectively generates a *CYCLE* ECPs every 256 cycles by the DWT. The ECP encoded with the cycle flag (see Figure 4.1) is representative for all profiling counters, as all profiling counter ECPs have the same transmission time per packet and are generating 5 rising flanks per packet. To compare the results at different TPIU prescaler values and different CPU frequencies the CPU polls an incrementing counter for the same fixed number of counter values for all measurements. The generated ECPs are lastly traced by the timer counter and the resulting timer counter values can be evaluated.

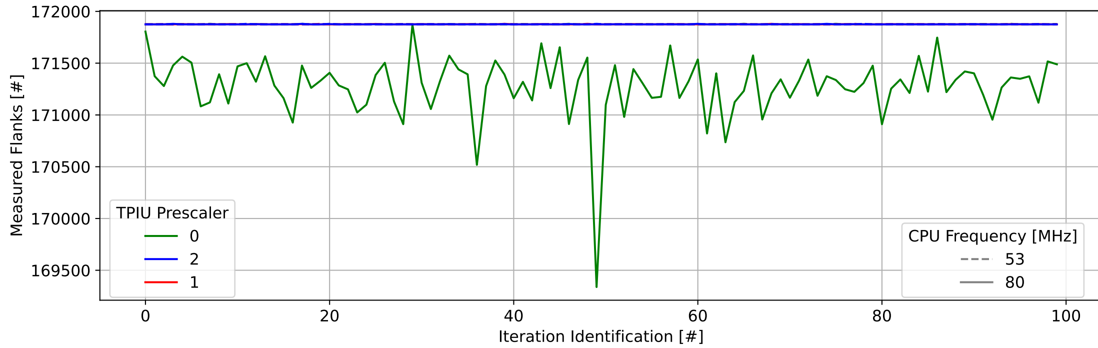


Figure 7.2: Measured flanks of fixed number of generated *CYCLE* ECPs. Experiment performed with different TPIU prescaler values and at a CPU frequency of 53 and 80 MHz.

Figure 7.2 shows that the timer counter readings for the prescaler 0 configuration and at a CPU frequency of 80 MHz fluctuate with a standard deviation of 297,52 flanks. This behavior can not be caused by the conveying congesting caused by a slow transmission speed, because the flank counter readings with the prescaler values of 1 or 2 show almost no fluctuations. Here the total flank counter readings show a standard deviation of maximum 1,18 flanks, which is negligible considering the value height of 172000 flanks. With an oscilloscope it is visible that the packet flanks per packet are still clearly visible at a prescaler value of 0 and a CPU frequency of 80MHz. The reason for the lost flanks might have to do with the maximum timer input frequency of 40 MHz, which is present at a TPIU prescaler value 0 and a CPU frequency of 80 MHz.

Therefore, the decision has to be made between a prescaler of 1 with a slower packet transmission and potential lost packets or a prescaler value of 0 with faster packet transmission and lost flanks per packet. As the measurements with lower prescaler values than 0 show no lost packet flanks and because in a real scenario a profiling counter overflows at a higher timespan than 256 cycles, potential of lost packets can be excluded. Therefore, the TPIU prescaler should minimally be configured to a value of 1 for profiling counters, which prioritizes the transmission accuracy over the transmission speed.

7.2.2 TPIU Prescaler with DTP

Before defining the TPIU prescaler value for comparator address matching, a *COMP_FUNCTION* has to be selected. As the TPIU prescaler choice depends on the

packet size, a *COMP_FUNCTION* has been chosen that generates the smallest packets. The smallest DTP is the DTAOP with a packet size of 24 Bit, as seen in the format Figure 4.2.

Regarding the packet congestion, the maximum necessary frequency of a DTAOP generation can be calculated with the cycle count of a load/store instruction, which is minimally 2 (see [61, sec. 3.3]).

With the Equation 7.2, a RAM intensive task can generate DTAOP at a frequency of 40 MHz, if the CPU clocks at 80 MHz. This would require the transmission speed to be minimally 960 MHz (24 Bit · 40MHz). Therefore, the output frequency of 40 MHz of the TPIU with one data output line is definitely a bottleneck.

To fully catch all DTAOP packets with an available transmission frequency of 40 MHz, a DTAOP should only be generated at a frequency of 1.6MHz (see Equation 7.3)

$$f_{DTAOPGEN} = \frac{f_{maxTPIU}}{Bits_{packet}} = \frac{40 * 10^6}{24} = 1.6MHz \tag{7.3}$$

Choosing the most useful TPIU prescaler value for DTAOPs is conflicted with the talked about inaccuracy of ECPs at a TPIU prescaler value of 0 and the bottleneck of the transmission frequency. Therefore, a further experiment was performed.

For n iterations, the experiment repeats the access to a data array that was prior configured to be matched by a single comparator. The looped data access provokes a situation where every next instruction should produce a DTAOP. Concurrently to the data access loop, the resulting packet flanks are counted by the timer counter to evaluate the bottleneck at different TPIU prescaler values and different CPU frequencies. (see Figure 7.3).

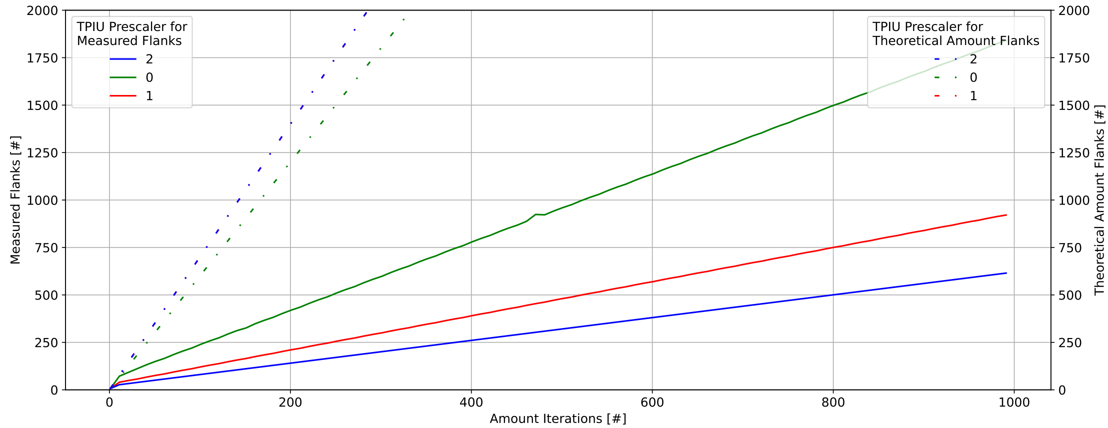


Figure 7.3: Measured DTAOP flanks per iteration of flash data access (left) vs theoretically generate-able DTAOP flanks (right) at a CPU frequency of 80 MHz and different TPIU prescaler values.

Figure 7.3 shows that with decreasing TPIU prescaler value, higher number of DTAOP are measured. In comparison to the theoretically calculated flanks, even at the lowest prescaler value a bottleneck is present.

Highlighted by the different dotted prediction lines, which are extrapolated with the measured flank count of the first measured DTAOP packet, a lost flank can still occur.

The congestion of the packets due to the bandwidth bottleneck makes a bigger difference in total read packet flanks than the inaccuracy at a CPU frequency of 80 MHz. Therefore, for the DTAOP a TPIU prescaler value of 0 should be used. Regarding the task property measurements by counting the data access to RAM and flash, the RAM matching is probably going to generate less useful results than flash data access matching. RAM can be accessed faster than flash, therefore RAM usage probably creates a higher rate of data access in the same amount of time.

7.3 Flanks Per DTAOP

With ECPs, the number of flanks per packet stay the same even with different profiling counters. Therefore, it is fairly easy to calculate the exact cycles tracked by any profiling counter based on the measured flanks (see Section 8.4.2).

To calculate the exact number of data accesses by tracking the flank count of DTAOP is more challenging. As seen in the format Figure 4.2, the flank number per packet changes because it depends on the used comparator identification and the data address offset a match occurred with.

Tracing the data access to RAM or flash results in different number of flanks per packet, because all 4 comparators are used and a range of addresses will be observed (see Section 6.3.1). To at least get a sense of how many data access are made with a certain timer counter value, the flanks per DTAOP can be approximated. To approximate the average flanks per packet, an experiment has been performed that traced all packet combinations dependent on the finite number of address offsets (0x0000 to 0xFFFF) and comparator identifications (0 - 3).

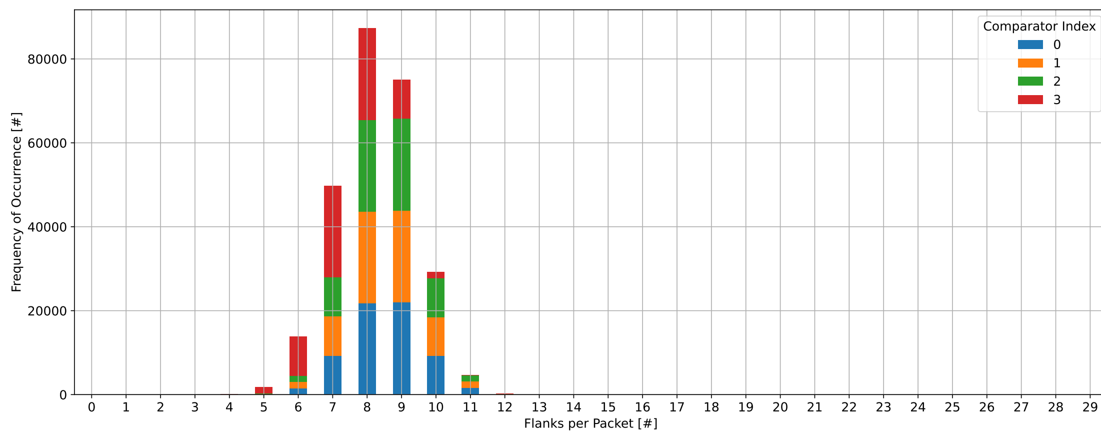


Figure 7.4: Frequency of Flanks per Packet of DTAOPs across all available comparators (0 to 3) and all available data address offset values (0x0 to 0xffff) at a CPU frequency of 80 MHz.

Figure 7.4 shows the frequencies of occurrence of flank counts per DTAOPs. The flanks per packet differ as the experiment generated DTAOPs with different matched address offsets (0x0000 to 0xFFFF) and different comparator identifications (0 - 3). The Figure shows the frequencies of flank occurrences at a CPU frequency of 80 MHz but was also performed with lower CPU frequencies that gave the same results.

Calculating the average of flanks per packet weighted by the flank occurrence across all used comparators results in 8.254 flanks per DTAOP.

Type	**ECP**	**DTAOP**
TPIU Encoding	NRZ	NRZ
Prescaler Value	1	0
Flanks per Packet	5	avg. 8.254

Table 7.1: Summary of ideal TPIU configuration in regard to transmission accuracy and highest possible throughput.

8 Methodology of Measurements

In this chapter, the measurement setup will be presented, the selection of the Bristol/Embecosm Embedded Benchmark Suite (BEEBS) will be examined and the data processing of traced task properties will be highlighted.

8.1 Measurement Principles

8.1.1 Reproducibility

When measuring metrics or testing certain behaviors of a system, the system should ideally respond to a set of inputs with a set of outputs which will not change, even if the process is repeated multiple times. To achieve this reproducibility, each measurement should always start from a system state that has not been changed by prior measurements.

To achieve this system behavior with a microcontroller, each measurement is minimally started by a *microcontroller reboot*. While there are system properties that, when changed, still persist after a system reboot, the optimal clean system state would be established by a full power down. As the full power down fails on the lack of automation, whenever a situation forms which can only start from a clean system state by a power down, a note is given.

8.1.2 Soundness

In the subsequent chapters, certain properties of the tracing utility are analyzed by measuring current and debug counter readings on the basis of running workloads that demand the system in different ways. To be certain that these properties are sufficiently captured, each workload is at least measured for 1 second and with an individual number

of iterations. The keyword **iteration** is used to describe how many times a workload is iterated for a single measurement.

As it is still possible that some trace recordings fail, e.g., when remote controlling an oscilloscope or a Digital Multimeter (DMM), and to be sure that identified properties are no outliers, a measurement is minimally repeated 10 times. Outliers that find their way into data for evaluation can be pointed at and will be removed. Therefore, **repetitions** is the keyword for how many times a measurement is repeated.

The combination of iterations and repetitions create the possibility of identifying some statistical properties and support the soundness of the findings.

8.1.3 Automation

As the implications of the property *Reproducibility* induces a microcontroller reboot before each measurement and the property *soundness* induces to minimally run a measurement for 1 second repeated at least 10 times, an accomplishment by manually performing the data collection would increase the total benchmark time overhead and would make the benchmarking process infeasible.

Therefore, a benchmark setup has been implemented to automate the data retrieval, which also increases the ability to be performed in the same way by other developers.

8.1.4 Verifiability

The measured data is saved in an object format for persistence. This enables the data to be verifiable and to be used by other researchers.

A further level of verifiability is to use an open-source set of tasks to measure the task properties with. Therefore, a benchmark suite has been searched and ported to RIOT, as seen in Section 8.3.

8.2 Measurement Setup

8.2.1 Software Components

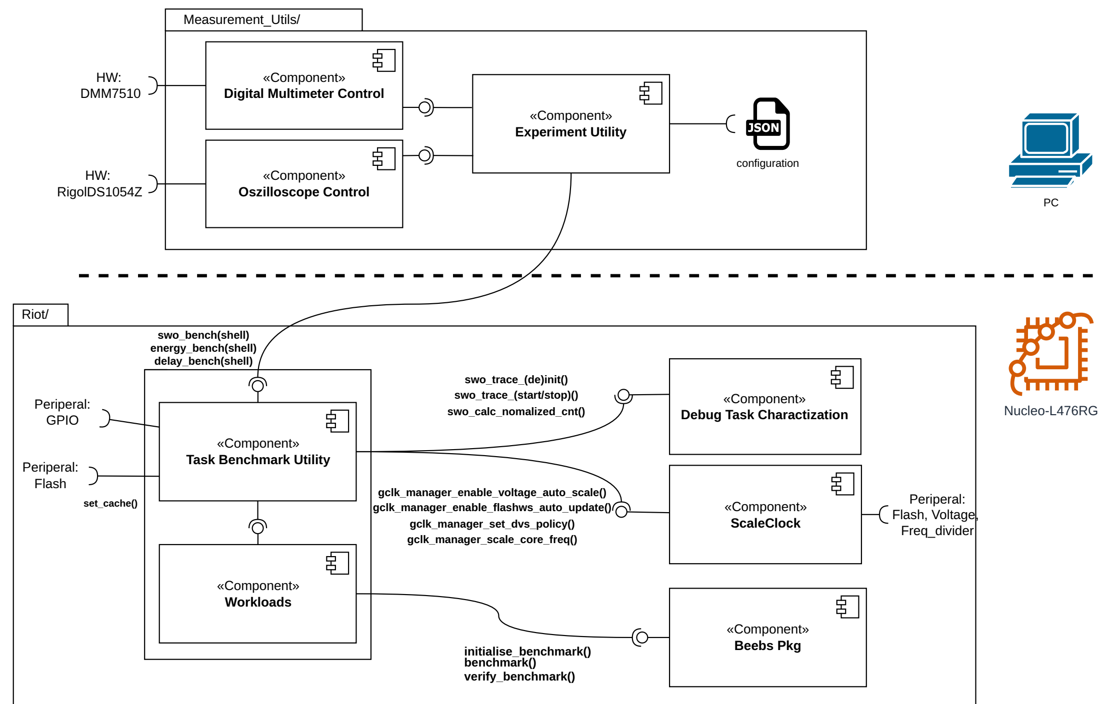


Figure 8.1: Component diagram of the measurement setup.

Figure 8.1 shows the cooperation of components which enable to create an automation setup for benchmarking the energy/ task characterization or delay measurements. The icons on the right and the separation line in the middle illustrate the execution location of software components. The source code of the measurement setup and the implementation of the task characterization model is available on GitHub [62].

Experiment Utility This is a python script, which is the point of contact for the user to start an automated task characterization benchmark and energy or delay measurement. It implements a shell argument interface with the python module *argparse* and its behavior is controllable with the input of JSON files.

Dependent on the configured behavior, the script accesses a hardware abstraction layer module for the digital multimeter DMM7510 (see Section 8.2.4) or the oscilloscope RigolDS1054Z (see Section 8.2.3).

The microcontroller is connected to the PC over the STLink USB connection, to communicate with the USART of the MCU. The script communicates with the software component *Task Benchmark Utility* via the RIOT *sys* module *Shell*.

After a measurement procedure has been finished, the data is acquired from the microcontroller, DMM or oscilloscope. The acquired data is made persistent to an object file format of the python module *pickle* for further data analysis and data illustration.

Digital Multimeter Control This python module communicates with the *Keithley* 7.5 Digit Graphical Sampling Multimeter (Model DMM7510) over *USBTMC* commands. It has already been implemented prior to this thesis work and could be used immediately.

Oscilloscope Control This python module communicates with the *RigolDS1054Z* oscilloscope over *USBTMC* by sending SCPI commands [63]. The module has been implemented during the work of this thesis.

Task Benchmark Utility This component implements a shell argument interface. It is the contact point to the automation script but can also be used by the user to quickly test or perform workloads for measuring the task properties with the task characterization model (swo_bench), energy consumption (energy_bench) or delay between two events (delay_bench).

To signal the start/end of a workload for energy or delay measurements, GPIO pins are triggered to communicate with the DMM or oscilloscope.

To start/stop the task characterization the access to the module *debug_task_characterization* is needed (see Section 5 and 6).

To alter the system state regarding Flash Wait State Adaption, core voltage, CPU frequency or Dynamic Voltage Scaling Policy (DVS Policy) the access to the module Scale-Clock [10] is needed. Further, to alter the Flash Cache state, the peripheral registers of the flash is used.

Any traced/measured workload is defined by the module *workloads* and accessible by a simple uniform interface to execute the task with a certain number of iterations and workload specific arguments. The module *Workloads* offers, among synthetic tasks, the workloads of the Riot package *Beebs* (see Section 8.3).

8.2.2 Experiment Procedure and Interaction

Figure 8.2 illustrates an experiment sequence to measure energy consumption of a workload dependent on the given JSON experiment configuration.

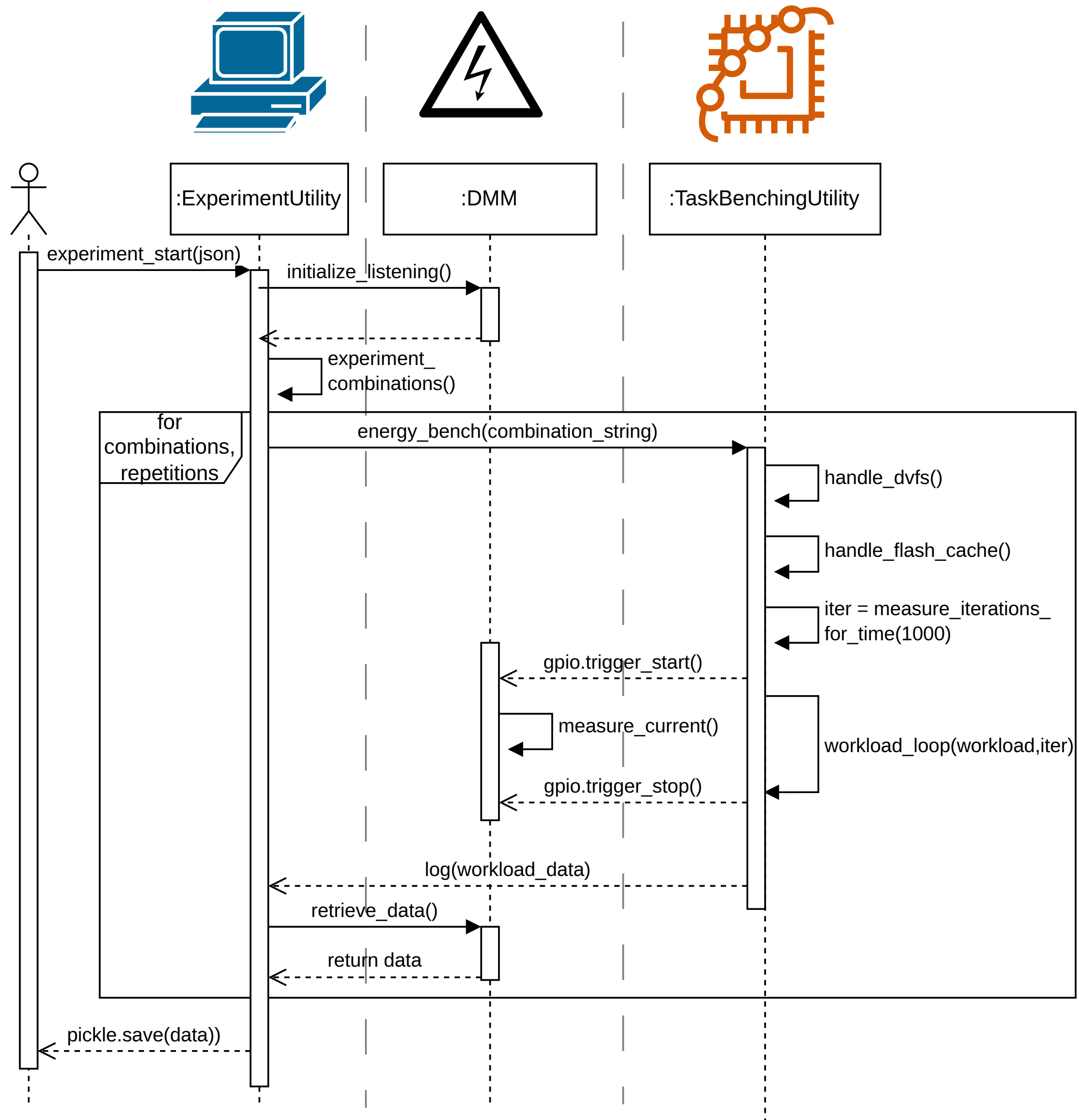


Figure 8.2: Sequence diagram of the interaction of different experiment setup components. The interaction of benchmarking a task running on a microcontroller and measuring the current with the multimeter is shown.

This is an example of the communication with the DMM. An experiment measuring the delay would look similar, only the DMM would be replaced by an oscilloscope and the oscilloscope would measure the delay. An experiment tracing a task property with the task characterization model would look similar too. Only the DMM would be removed completely and the GPIO trigger functions would be replaced by *swo_trace_start* or *swo_trace_stop* functions, where the results are returned by the MCU.

8.2.3 Time Measurements

To accurately measure time between two events in software, the RigolDS1054Z oscilloscope [64] is used. It offers a maximum samplerate of $500\,\mathrm{MS\,s^{-1}}$ for two input channels and $1\,\mathrm{GS\,s^{-1}}$ for one input channel.

Measuring the delay between events in software is made possible by mapping these events to signals on GPIO pins. Therefore, the time difference between generating high or low signals with GPIO pins can be accurately measured with an oscilloscope.

When using a GPIO trigger pin as a start point for time measurements, there is also some delay between setting the GPIO by software and finally measuring a low or high signal on the pin. This delay can be neglected when a time measurement uses the same trigger pin method for the start and end point, as this adds the delay on both sides of the measurement. An inaccuracy might exist where the measurement only starts or ends with a single GPIO trigger pin. To expose this inaccuracy, a GPIO delay measurement was performed as seen in Figure 8.3, which uses the same software trigger functions that are also used for successive delay experiments. This measurement involves measuring the time between two flanks (rising or falling) of one or two pins with the oscilloscope. The triggering of the pins has been performed with successively executed gpio_set() or gpio_clear() functions.

The samplerate of minimally $500\,\mathrm{MS\,s^{-1}}$ with 2 input channels should suffice the experiment needs as the minimal GPIO delay is 116ns, meaning with the smallest GPIO delay, a worst-case deviation of 4% exists, which is acceptable.

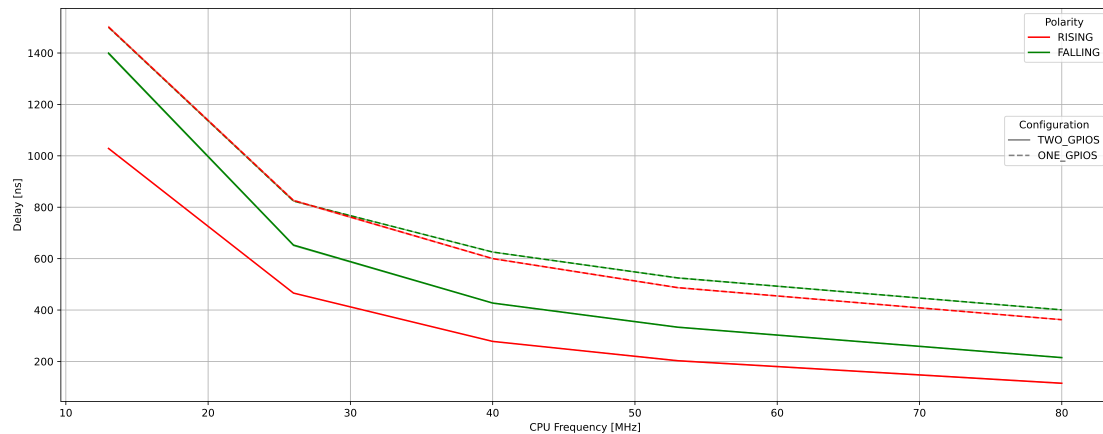


Figure 8.3: Measured delay between two GPIO flanks dependent on their polarity and number of GPIO used at different CPU frequencies.

Each repetition of the GPIO delay measurement precedes a reboot of the microcontroller. This is especially important when measuring the delay with GPIO pins where the first access to memory-mapped registers involves loading addresses into the CPU or cache which takes additional time. This fact is of magnitude as successive delay measurements also only trigger the GPIO pin once or two times every measurement cycle and are therefore affected by that loading delay as well.

8.2.4 Power and Energy Measurements

The metrics power and energy are based on current measurements. All current measurements were performed with *Keithley*'s 7.5 Digit Graphical Sampling Multimeter (Model DMM7510) [7].

To measure the MCU current, the DMM probe cables are connected to the jumper connection JP6 (IDD) [6] by replacing the prior jumper (see Figure 8.4). Internally the DMM connects a precision-resistor to the shown front panel inputs and samples the voltage that drops on the resistor. As the voltage and resistance of the precision-resistor is known by the DMM, the current can easily be calculated with Ohm's law: $I = \frac{U}{R}$.

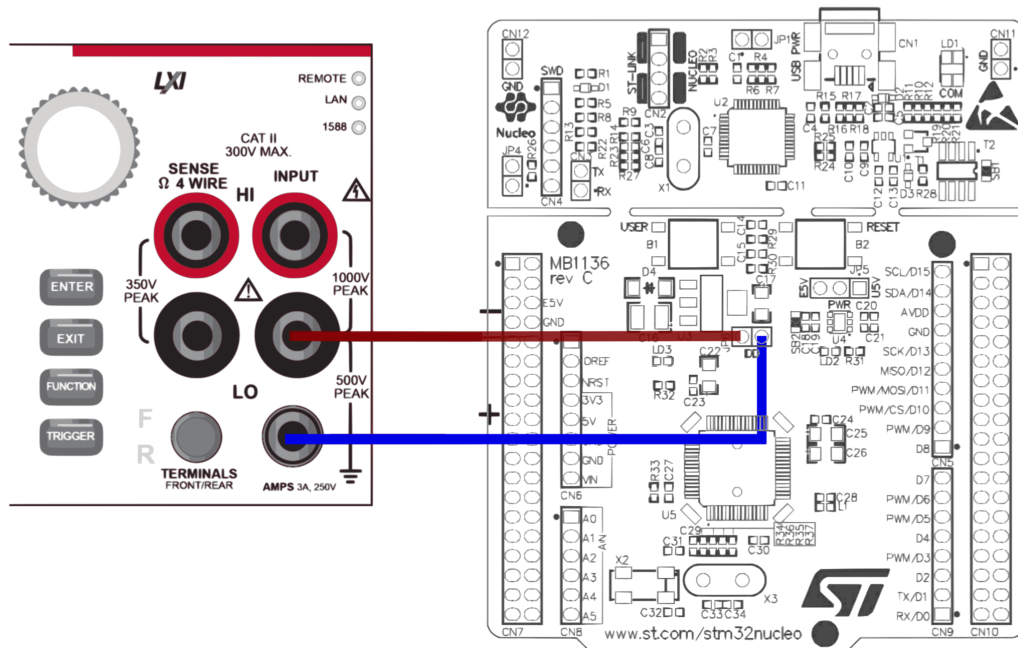


Figure 8.4: Connection of the Nucleo-L476RG board and the DMM to measure the microcontroller current, Figure based on [7].

The current is measured for the duration of a workload by signaling the measurement start and end of the DMM via a GPIO pin that is triggered by the MCU. Further, the current is sampled at a minimal rate of 10000 samples per second.

When the current is measured for all kinds of workload configurations, the power, which is $P = I * V$, and the energy, which is $E = I \cdot V \cdot t$, can be calculated. Thereby, the current values relate to a static supply of 3.3V.

A power comparison will be used, for example, to identify the consumption overhead of the tracing utility (see 9.3). Here the exact number of workload iterations or tracing length is not important and an average of the traced power consumption is viewed at.

When trying to find the most energy efficient frequency setting for a specific workload configuration, the metric of total energy is of interest. To correctly compare the different energy consumption for one workload configuration, the workload should be performed with the same number of iterations across different CPU frequencies.

8.3 Selecting a Benchmark Suite for Embedded Devices

The premise for this section is to select a broad set of different tasks to evaluate the task characterization model with. Thereby, creating not all tasks saves a lot of time and energy, avoids creating unrealistic tasks and enables other researchers to more easily verify the results presented in Section 10.

The selection for a suite is accompanied by the following requirements:

Property	Description
Comprehensible code base	This means among others things no debug messages with *printf* for example
Open source	Porting the benchmark suite as package to RIOT, a free and open source license is preferable
Embedded use	Represents a wide variety of embedded application areas
Variety of system stressing properties	Has tasks with different RAM and Flash memory access, tasks that provide different CPU loads or tasks that use I/O

Table 8.1: Properties to look for when searching for a suitable benchmark suite for this thesis.

8.3.1 SPEC CPU

Widely used benchmark suites are the Standard Performance Evaluation Corporation (SPEC) CPU Benchmarks [65, 66]. Unfortunately, they are designed for general-purpose computing performance and are not designed for embedded devices [67].

8.3.2 MediaBench

MediaBench [68] and MediaBenchII [69] contain complete software applications for the application areas of video, graphics, image compression, audio, speech and security.

Even though the application areas appear interesting, they are not designed for the embedded / IoT area. Furthermore, source code is available on their website [69], but an open-source license statement can not be found.

8.3.3 MiBench

The MiBench suite [66], in the form of the repository MiBench2 [70], is a free and open-source suite that divides the code base into six different areas of the embedded market.

Unfortunately, it uses file I/O and usability-wise the source code still consists of many prints. The source code is not well documented and contains huge datasets, which makes the the suite unusable for lower ROM/RAM systems.

8.3.4 ERCBench

ERCBench [67] is an open-source benchmark suite for embedded and reconfigurable computing. It covers the application areas, audio processing, communications, cryptography, and image and video processing. It not only offers software but also hardware benchmarks for reconfigurable devices like FPGAs.

Unfortunately, source code for the package implementation analysis has not been found.

8.3.5 BEEBS

The Bristol/Embecosm Embedded Benchmark Suite (BEEBS) [20] is open-source, optimized for embedded devices and contains a large set of workloads. The variety of tasks evaluated mostly originate from other benchmark suites. The tasks are categorized by the same application areas as the MiBench paper. The application areas Automotive, Consumer, Network, Telecommunication and Security are covered.

In the paper [20], Pallister et al. select 10 different tasks justified for resulting in different energy consumption. Nonetheless, the paper appendix lists all benchmarks that were evaluated to choose the set of 10 tasks from. The list structures the tasks by their task properties targeted at evaluating energy consumption, fit in MCU ROM and gives insights on the embedded applicability. The embedded applicability represents the likelihood the task functionality would be used in real embedded systems. The task properties that the group of 10 selected tasks differ at, and which also affects energy consumption, are the usage of integer operations, usage of floating point operations, the memory access intensity and the branching frequency. The paper tests the benchmarks on different processors, with the Cortem-M3 being verified to compile and run with the benchmarks.

To conclude, this benchmark suite satisfies most of the targeted requirements. The code base does not use debug messages and could be implemented as a RIOT package with reasonable effort. I/O or peripherals are excluded from the benchmark tasks for portability reasons, but a task will be created as a replacement, as seen in Section 10.2. Even though the selected 10 tasks of the paper [20] have the highest suitability of their used metrics, tracing the less suitable benchmarks with the task characterization model increases the quality of a comprehensive evaluation (see Section 10.4).

8.4 Measurements and Data Processing

In Chapter 10 energy measurements at different frequencies of tasks are examined. Thereby, changing the CPU frequency enables to also change the CPU voltage and Flash Wait State (FWS). This relation is illustrated in Section 8.4.1.

Furthermore, Chapter 10 also tries to correlate certain energy measurements with task properties traced with the task characterization model. For this correlation the trace measurements are normalized, which is shown in Section 8.4.2.

8.4.1 Frequency, Voltage and Flash Wait State (FWS) Configuration

Chapter 10 evaluates measurements on the most energy efficient setting regarding CPU frequency and choice of DVS Policy. Reducing the CPU frequency reduces the dynamic power consumption, whereas a lower voltage reduces dynamic and static power consumption. Many MCUs are equipped with an embedded flash memory peripheral for persistent data storage, whose access latency is dependent on the configured CPU frequency and voltage. As the CPU often runs at a higher frequency than the flash memory, the number of Flash Wait State (FWS) must be correctly programmed to read valid data from flash memory [5, sec. 3.3.3 Read access latency].

The voltage range for the STM32L476RGT6 MCU can be configured to two different ranges. A high-performance range, where the main regulator provides a typical output voltage of 1.2V and the system clock frequency can be configured to up to 80 MHz. Secondly, a low-power range, where the main regulator provides a typical output voltage of 1.0V and the system clock frequency can only be configured to up to 26 MHz [5, sec. 5.1.8 Dynamic voltage scaling management].

The direction of dependency between voltage, frequency and FWS depends on the perspective. From the perspective of wanting to scale the CPU frequency, the lower the frequency, the lower the FWS can be configured, and the more CPU cycles can be saved when accessing the flash memory. On the other hand, the lower the CPU frequency, the lower the voltage can be configured. When lowering the CPU frequencies, there are configurations where only the voltage or the FWS can be minimized. This context is handled by the selection of a Dynamic Voltage Scaling Policy (DVS Policy) of the ScaleClock implementation [17]. The scaling policy *Fast Flash* prioritizes a FWS reduction when lowering the frequency. The policy *Low Voltage* prioritizes a reduction of the voltage range, if the CPU frequency is in the bounds of the lower voltage range.

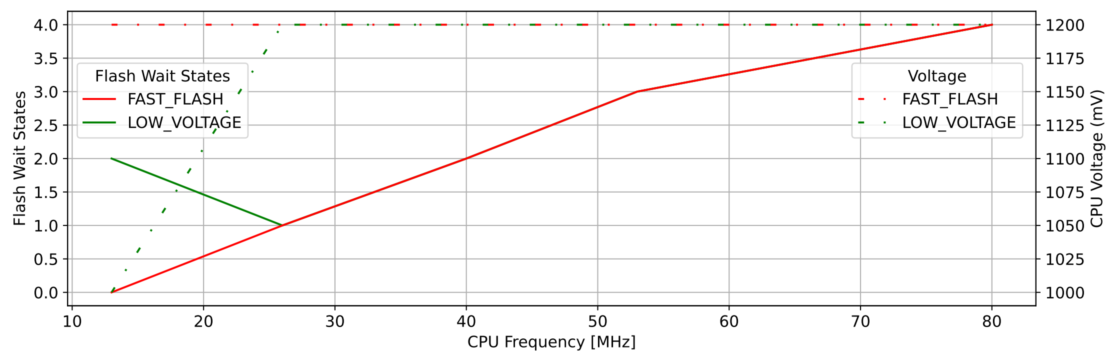


Figure 8.5: Relation between FWS, CPU voltage and CPU frequencies dependent on the configured DVS Policy and enabled DVS.

Figure 8.5 visualizes the two DVS Policys in terms of FWS, CPU voltage and CPU clock frequency. As the measurements of the following sections are performed at the frequencies of 13 MHz, 26MHz, 40MHz, 53MHz and 80 MHz, the two policies only differ at a CPU frequency of 13 MHz. To make use of the reduction of FWS when lowering the CPU frequency, the ScaleClock functionality of Flash Wait State Adaption (FWSA) has to be enabled. To make use of the reduction of voltage, the functionality of Dynamic Voltage Scaling (DVS) has to be enabled.

8.4.2 Data Processing for Traced Task Properties

Original Counter Reading

To be able to calculate the correct executed instructions (see Equation 4.1) with the profiling counter results, the timer counter readings need to be processed as they only represent hardware source packet flanks. As seen in Table 8.2 the profiling counters overflow at a value of 256, meaning one ECP is equal to a cycle count of 256. A DTAOP is generated per data access of the CPU, therefore a single DTAOP is equal to a single data access.

With the knowledge of flanks per packet, a scaling factor can be formed that calculates the original value (profiling counter cycles or data access) from the measured flanks dependent on the trace method.

Trace Method	**Unit per Event**	**Measured Flanks per Packet**	**Scaling Factor**
Profiling Counters	256 Cycles	5 (see Table 7.1)	256/5
Comparator Address Matching	1 Data Access	8.25 (see Table 7.1)	1/8.25

Table 8.2: Trace method properties to calculate the original counter reading from measured flanks.

Counter Normalization

What do we want? We want to compare tasks by their task properties that are traced with the task characterization model. Tasks need a different number of cycles to be executed, therefore the absolute values of the traced flanks per task property also depend on the number of cycles the trace lasted. The absolute tracing values need to be normalized to compare the task properties independent of the tracing length. This should enable to form sentences like, e.g., a task needs 20% of all tracing cycles to perform load/store operations.

How should the data be processed? The absolute tracing measurements are not only scaled to the original counter (see Table 8.2) but are also normalized by the number of cycles a trace lasts.

Normalizing only by the time a trace lasted would make comparing tracings at different CPU frequencies difficult. A trace at different CPU frequencies, but with the same number of cycles, would artificially normalize the results at lower CPU frequencies to lower values because cycles take longer at lower frequencies.

Traced task properties are normalized as follows:

$$P_{normalized} = \frac{CNT_{RAW}}{f_{CPU} \cdot t_{trace}} = \frac{CNT_{RAW}}{CYC} \tag{8.1}$$

What does the normalized data represent? The trace results originate from two different trace methods. The profiling counters represent cycles spent in different CPU operations (except *FOLD*) and the comparator counters represent absolute data accesses.

Inserting the raw counter readings of the two different trace methods into Equation 8.1 produces results of different meaning. With the profiling counters representing cycles, the normalization Equation 8.1 produces percentages and the normalized values should be in the range of 0 to 1. For example, inserting *LSU* measurements into the equation results in a percentage of load/store instruction cycles of total cycles.

The normalized DAM results represent data accesses per cycle. Therefore, the values do not represent percentages. The maximum normalized comparator counter value could not be calculated a priori as DAM generates packets with different flanks and as the packet stream is bottlenecked by the bandwidth (as seen in Section 7.3). Based on measurements conducted in Section 10, a range of values between 0 and 0.04 have been observed. (as seen in Figure 10.18). The normalized DAM results are not scaled to a range between 0 and 1 to keep the data processing uniform for both trace methods.

9 Overhead of Task Characterization

This Chapter discloses the delay between the generation of hardware source packets (ECP, DTAOP) and the recognition by the feedback mechanism. This delay is important if the task characterization model should be used for live feedback of different tasks. Secondly, this Chapter investigates the initialization duration of the tracing utility, which is overhead in terms of CPU usage. Lastly, the energy overhead of tracing certain task properties will be examined, as this reduces the energy savings when optimizing the performance utilization.

9.1 Delay of ECP and DTAOP

The time between the generation of a ECP or DTAOP and afterwards the detection by the timer counter is not zero. This delay is of importance considering the task characterization model is intended to be used for live feedback during the execution of different tasks. To correctly attribute a packet to the executed task, the delay should be known prior. Therefore, an experiment was conducted to give a worst-case estimate on the delay between an ECP/DTAOP generation and the full recognition by the timer counter.

The *CPI* and *LSU* overflow packets can't be precisely triggered as they count additional cycles for certain CPU instructions. The *CYCLE* counter on the other hand deterministically increments every cycle and can therefore be configured to trigger an event right after enabling the event generation. Although the different profiling counters result in different packet encoding, all packet signals should take the same route from the DWT block to the final timer counter input channel. Further all packets have the same first and last flanks on the line in common and have therefore the same transmission time.

As the comparator matching always triggers a DTAOP event, it is also possible to precisely generate a DTAOP when conducting the time measurement.

The estimate methodology consists of three steps: First, setting the specific internal profiling counter one counter increment (decrement for the cycle counter) before a counter overflow occurs. Second, triggering a GPIO pin, at best one instruction prior to enabling event generation, in order to measure the delay between the generation and the first packet flank by the oscilloscope. And lastly, enabling the overflow event or triggering a data address match.

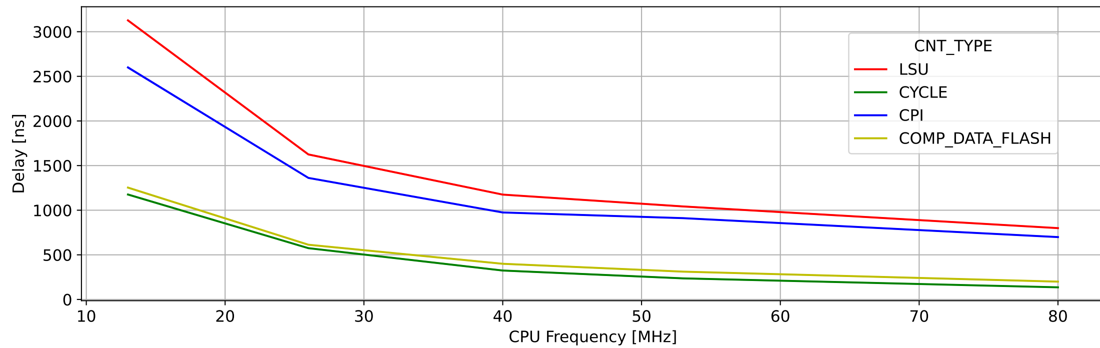


Figure 9.1: Delay between the generation of different hardware source packets and the arrival of the first packet flank at timer counter. Measured at different CPU frequencies and a TPIU prescaler value of 0.

Figure 9.1 shows that the cycle counter results in the smallest ECP delay, because it is triggered right after triggering the GPIO pin. The CPI and LSU ECPs probably have the same delay but are triggered later as they react only to specific CPU instructions. The actual time between ECP generation and the first flank seen by the timer counter is therefore best illustrated by the delay of the cycle counter packets.

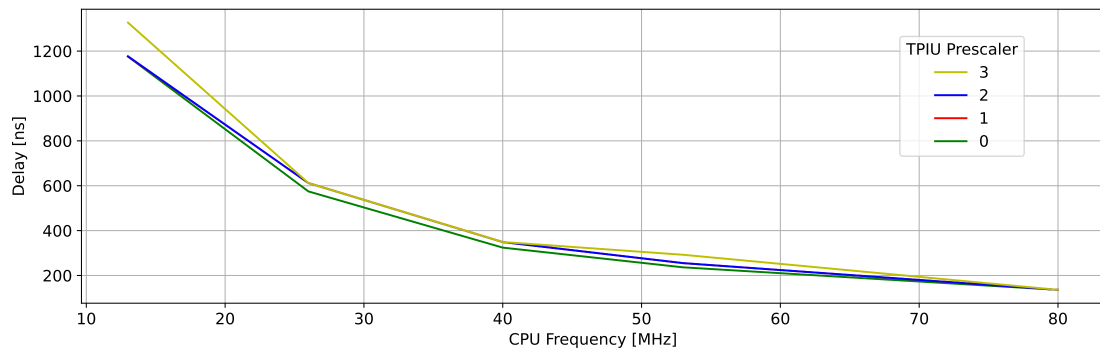


Figure 9.2: Delay between the generation of *CYCLE* ECPs and the arrival of the first flank at the timer counter. Measured at different CPU frequencies and different TPIU prescaler values.

Figure 9.2 shows that the delay between generation and arrival of the first flank of an ECP is not affected by the TPIU prescaler setting. Figure 9.3 shows that the packet length (delay between first and last flank) changes due to changing the TPIU prescaler value, as this changes the output frequency. For the ECPs the red line should be considered, as prescaler 1 has been proved to be the ideal configuration regarding accuracy. For the DTAOP, the green dotted line should be considered, as prescaler 0 provides the ideal configuration regarding most packets observed (see Table 7.1).

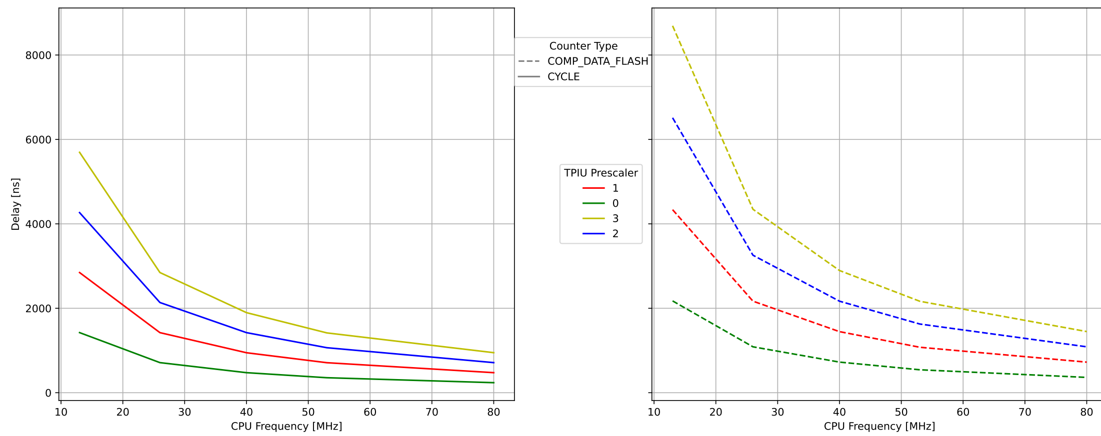


Figure 9.3: Packet length (delay between first and last flank) for trace packets (ECP(left), DTAOP(right)). Measured at different TPIU prescaler values and different CPU frequencies.

The total ECP delay is therefore calculated by adding the data of figs. 9.2 and 9.3 grouped by the TPIU prescaler and CPU frequency, and considering the GPIO delay of Figure 8.3 as a potential inaccuracy. The GPIO inaccuracy is given as a potential value that the ECP delay might fluctuate with.

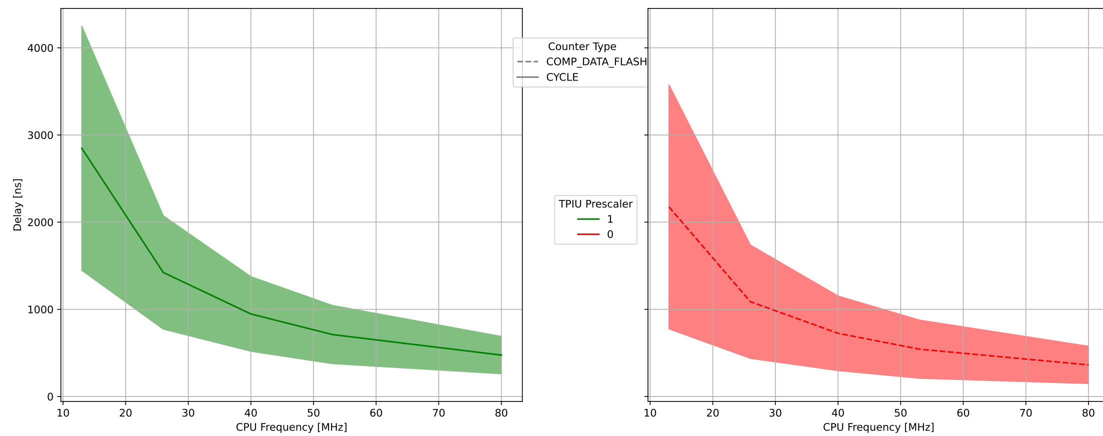


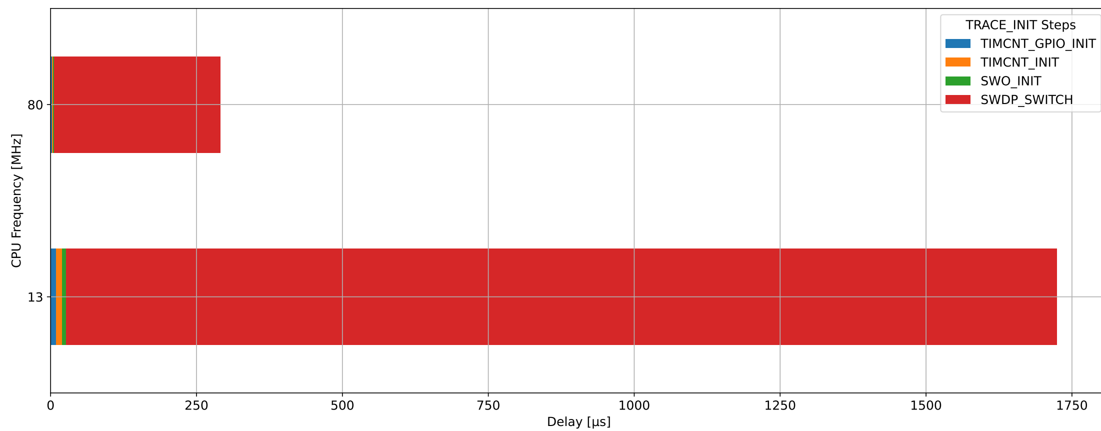
Figure 9.4: Delay of Figure 9.3 combined with the inaccuracy of using GPIOs pins for triggering (see Figure 8.3).

To put the ECP delay into perspective, a context switch at 80 MHz has been measured to take around 2760ns. Assuming a system does not only perform context switching, this delay is considered small enough to precisely attribute events to corresponding tasks or code sections.

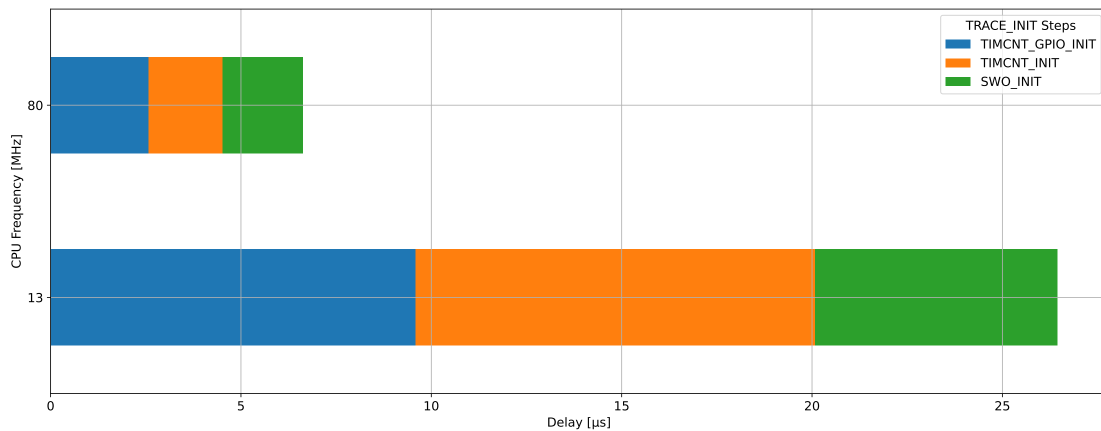
9.2 Overhead in Time

To quantify the time overhead of different operations needed for controlling the tracing, we examine the initialization of the used debug modules, and the starting and stopping of traces dependent on the trace method.

Figure 9.5 shows the time delay of the *swo_trace_init* initialization steps from enabling the packet conveying (*SWO_INIT*) to enabling the timer counter (*TIM_...*), as introduced in Section 6.3. Part (a) of the Figure shows all initialization steps and part (b) excludes the step of switching the SWJ-DP to SWD-Mode, as this step takes significantly longer.



(a) Initialization with switching SWJ-DP to SWD mode.



(b) Initialization without switching SWJ-DP to SWD mode.

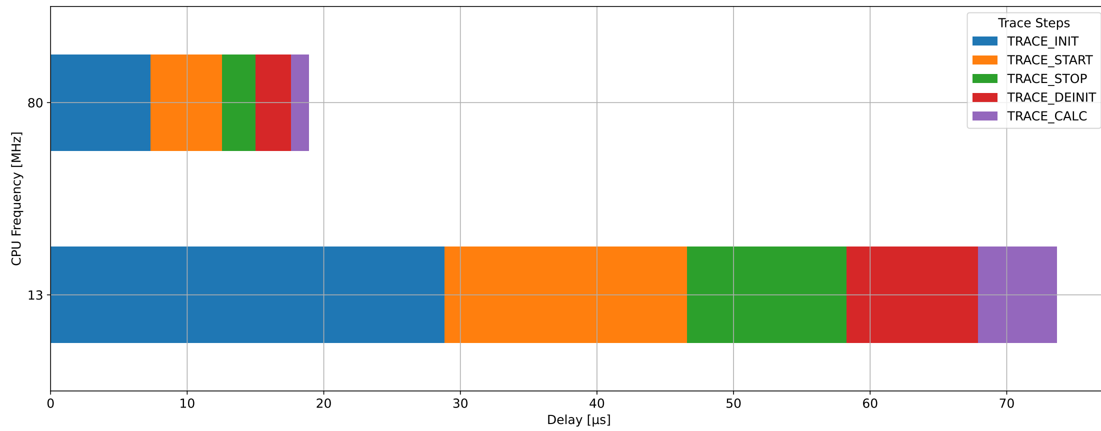
Figure 9.5: Overhead delay performing the tracing initialization steps.

While the first 3 initialization steps take only 7µs at 80MHz, switching the Debug Port from JTAG to SWD mode takes the longest time ($SWDP_SWITCH$). Ignoring the SWD-switch, the initialization of the debug components takes only 30% of the initialization duration, while the timer counter GPIO and registers take 70% of the initialization duration.

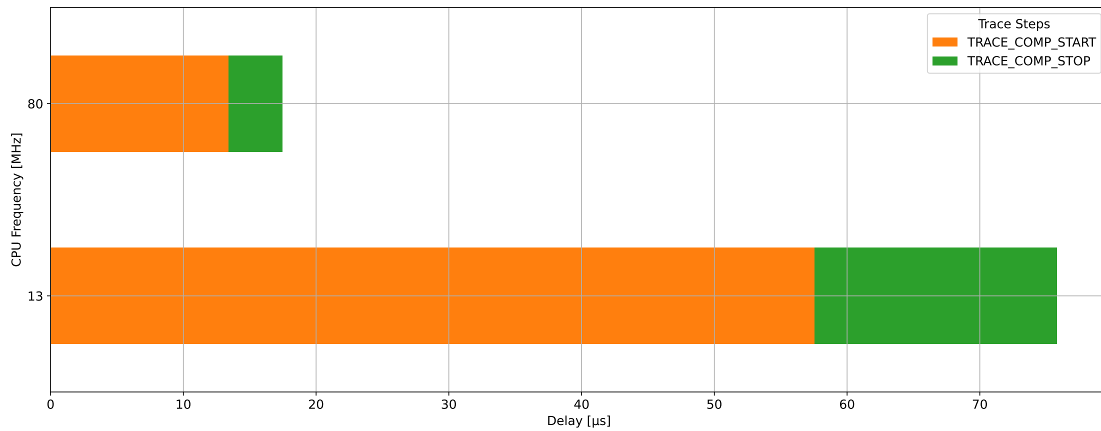
The reason for the high SWD-switch delay is the emulation of the synchronous bus interface via GPIO bit-banging and the slow transfer time of the specified switch sequence, which does not compensate for the difference in CPU frequency. While the switching represents 97% of the initialization delay, it is actually only necessary once per microcontroller power down and won't be switched back via a reset.

Figure 9.6 shows the rest of the necessary tracing steps as introduced in Section 6.3. The initialization over starting/stopping the event generation (*TRACE_[COMP_]_START/STOP*) dependent on the trace method is shown, to calculating the normalization by *TRACE_CALC* (as described in Section 8.4.2).

Further, the result *TRACE_INIT* in Figure 9.6 does not consider the delay created by the debug port switching.



(a) Profiling counter control function and the initialization, deinitialization, normalization functions.



(b) Comparator specific control functions.

Figure 9.6: Overhead delay of tracing control functions dependent on trace method (profiling counter, comparator).

Comparing part (a) and (b) of Figure 9.6 shows that starting and stopping tracing with comparators takes almost twice as long as tracing with the profiling counters. This is probably caused by the fact that multiple DWT comparators have to be configured.

9.3 Overhead in Power Consumption

Before examining the power consumption results, the methodology of the experiments should be noted. In the following, different trace configurations are measured for current while performing three different workloads. To compare the power overhead of the trace configuration per workload, a current measurement of the workload has been performed without activating any tracing (*reference current measurement*). Therefore, the current overhead of certain trace configurations can be calculated by subtracting the measured current of a workload with activated tracing by the *reference current measurement*.

Regarding the three different workloads, *FLASH_ADD* performs integer *ADD* math operations with values that are constantly read from Flash memory. *REG_DIV* performs integer *DIVIDE* math operations that are tried to be kept in CPU registers without memory loading. The workload *SLEEP* does a CPU *sleep* for a specified time. These workloads have been chosen to represent different aspects of system behavior.

Share of Tracing Configurations Figure 9.8 shows the overhead power consumption of the different tracing configurations (see Figure 9.7) in proportion to the overhead power consumption of only counting the *CYCLE* profiling counter, without activating the packet generation, packet conveying and timer counter.

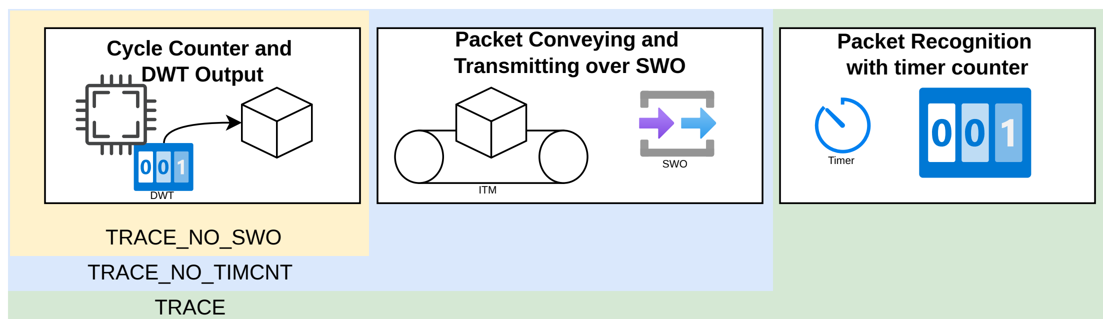


Figure 9.7: Visualization of different trace configurations measured for power consumption in Figure 9.8.

The trace configurations are shown in Figure 9.7. *TRACE_NO_SWO* only enables the *CYCLE* profiling counter and event generation, but disables the conveying of packets over ITM, over the SWO output or to the timer counter. The *TRACE_NO_TIMCNT* configuration enables a profiling counter, the event generation and packet conveying, but does not activate the timer counter. Lastly, the *TRACE* configuration represents a normal trace with enabled profiling counter, event generation, packet conveying and enabled timer Counter.

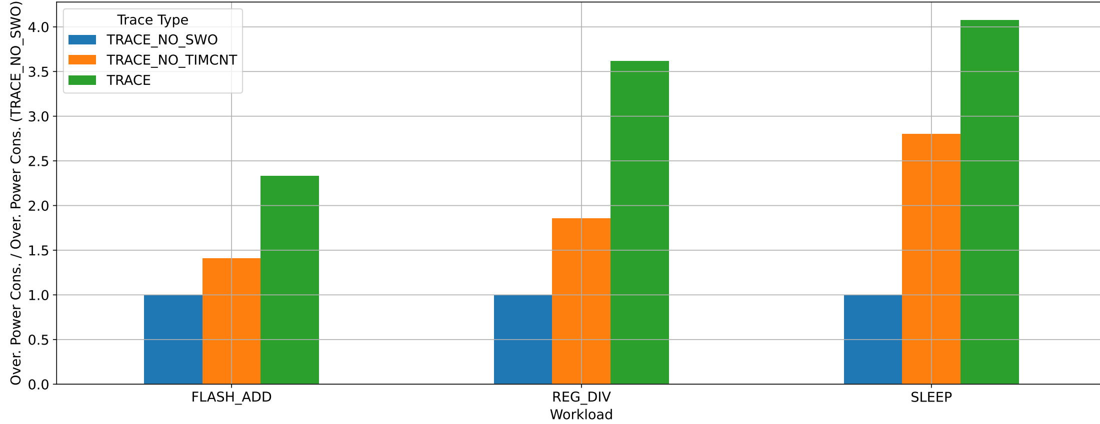


Figure 9.8: Overhead power consumption of different trace configurations in proportion to overhead power consumption of *TRACE_NO_SWO*, grouped by different workloads and measured at a CPU frequency of 80 MHz.

Figure 9.8 illustrates that the packet conveying and the timer counter are responsible for most of the overhead power consumption.

Figure 9.8 only shows the relation of overhead power consumption between the trace configuration. It is reasonable to mention the actual power consumption of the workloads without any tracing enabled and the proportion of the tracing overhead power consumption to the actual workload power consumption. The actual workload of *FLASH_ADD* has the highest mean power consumption of 46.95mW, the *REG_DIV* workload consumes 33.98mW and the *SLEEP* workload has the lowest mean power consumption of 21.18mW. Thereby, the *TRACE* configuration increases the *FLASH_ADD* power consumption by 7.62%, *REG_DIV* is increased by 9.63% and the *SLEEP* workload power consumption is increased by 24.11%.

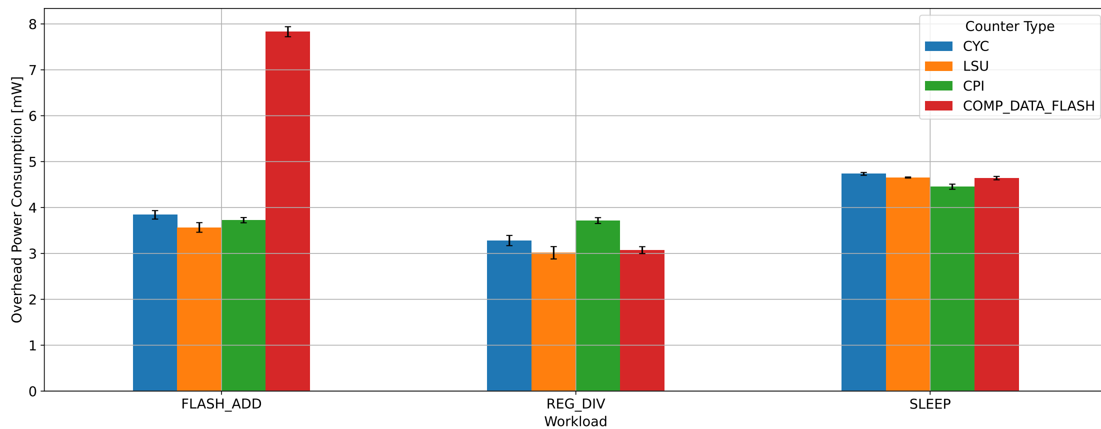
Considering the overhead power consumption relation between the tracing components shown in Figure 9.8 and the overhead power consumption in relation to the power

consumption of the actual workload, the event generation is responsible for 2.66% (*REG_DIV*) to 5.92% (*SLEEP*) overhead power consumption. The packet conveying is responsible for 1.34% (*FLASH_ADD*) to 10.67% (*SLEEP*) and the timer counter is responsible for 3.02% (*FLASH_ADD*) to 7.52% (*SLEEP*).

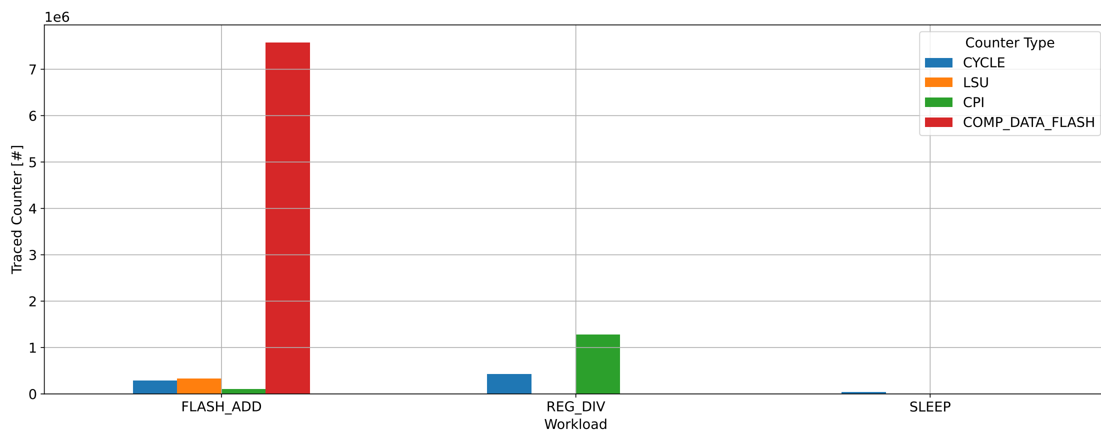
It is shown that the timer counter can be responsible to up to 50% of the overhead power consumption (see *REG_DIV* in Figure 9.8). Therefore, the task characterization model can benefit from timer optimizations to reduce the total overhead power consumption.

This section appointed power consumption measurements to different trace configurations with enabled *CYCLE* profiling counter, as it generates ECPs independent of the current workload. It would also have been interesting to appoint power consumption values to trace configurations with DAM, but the comparators can not observe addresses and generate packets independent on the traced workload. As the packet conveying and packet recognition by the timer counter is also used for DAM, the overhead power consumption should be similar to the mentioned values. Only the power consumption of matching data accesses by the comparators are missing. This might be exposed by future researchers.

Event Generation Dependency Another interesting question is whether the power consumption of the tracing utility is dependent on the number of generated packets. Therefore, measurements were performed with the mentioned 3 workloads and enabled tracing with profiling counters or DAM.



(a) Overhead Power Consumption with caps as standard deviation.



(b) Trace Counter Flanks.

Figure 9.9: Overhead power consumption of tracing different task properties and corresponding traced counter flanks. The trace has been performed at a CPU frequency of 80 MHz.

Before evaluating Figure 9.9, it has to be noted that each workload has been performed with a thousand iterations, which results in different execution times. As part (b) shows the total count of read flanks per counter type, the height is dependent on the performed workload. Further, when tracing the cycle counter the flank counter readings are not higher than all other counter types, because a cycle ECP is only generated every 1024 cycles dependent on the *POST_ CNT* and *SYNC_ TAP* field of the *DWT_ CTRL* register.

FLASH_ ADD of Figure 9.9 shows that the overhead power only correlates with the count of read flanks for the comparator counter, not for the profiling counters. On the other

hand, the overhead power consumption for profiling counters at the workload *REG_ DIV* does show this correlation.

When comparing the power consumption, the difference between no packet generation and packet generation is quite small for the profiling counter events. But there is a huge power increase when the DAM generates DTAOP in comparison to no packets. This proves that the packet generation has an effect on the overhead power consumption.

As Figure 9.9 shows only the overhead power consumption, it is reasonable to again mention the actual power consumption of the workloads without any tracing enabled, which are similar to Figure 9.8. The actual workload of *FLASH_ ADD* has the highest mean power consumption of 46.84mW, the *REG_ DIV* workload consumes 33.87mW and the *SLEEP* workload has the lowest mean power consumption of 21.49mW.

To summarize, the overhead power consumption for profiling counters ranges from 3mW to 3.8mW (8.85% to 8.1% in proportion to the actual workloads) for load intensive workloads (REG_DIV and FLASH_ADD) and can grow to 4.7mW (21.8%) with no CPU load (SLEEP). Thereby, the overhead power consumption increases by up to 0.7mW (2%) due to the generation and reading of the ECPs. The overhead power consumption for DAM ranges from 3.1mW (9.15%) for load intensive workloads to 4.6mW (21.4%) with no CPU load. Hereby, the overhead power consumption increases by up to 4.7mW (10.03%) due to the generation and reading of the DTAOPs. This increase of overhead power consumption comes from the significant higher number of packets transferred. DAM generates a trace packet directly after a match, whereas the profiling counters only generate a packet when the internal profiling counter overflows.

To also name the overhead power consumption at different CPU frequencies, Figure 9.10 shows the overhead power consumption with enabled flash DAM. The trace with comparator is chosen as it covers the whole power consumption range of the three workloads, as seen in Figure 9.9. Figure 9.10 shows that the overhead power consumption is proportional to the frequency selection.

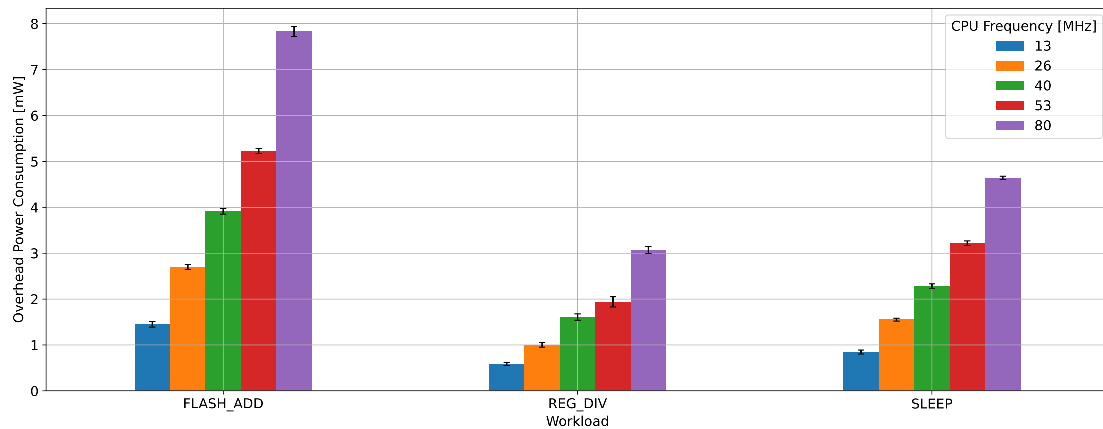
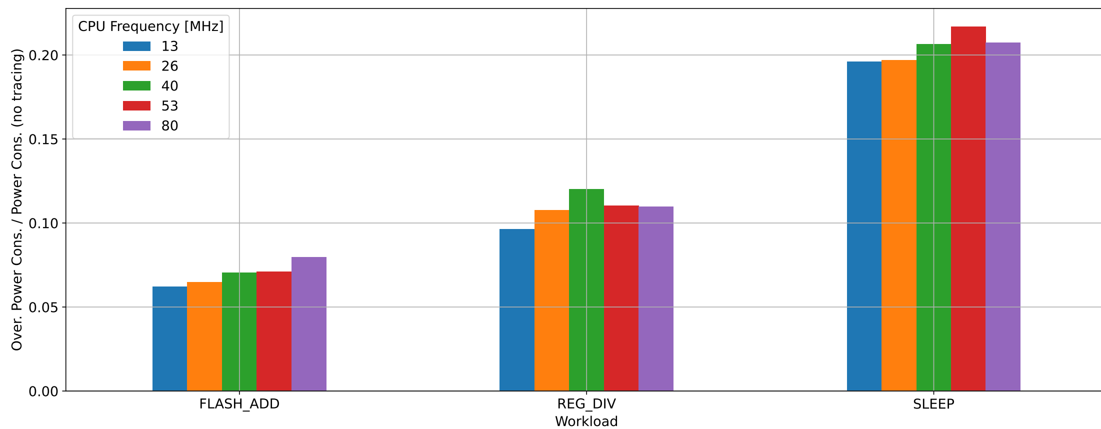


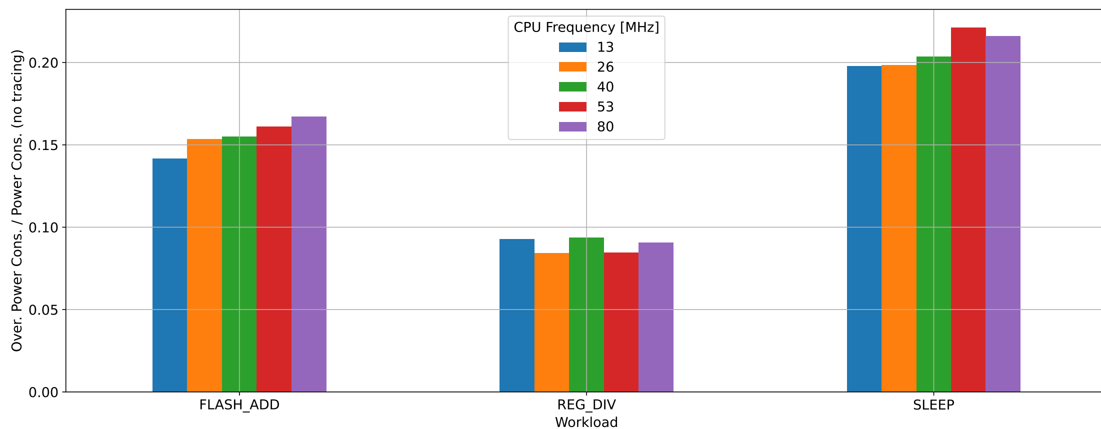
Figure 9.10: Overhead power consumption of tracing flash data accesses at different CPU frequencies, with enabled Flash Wait State Adaption and selected *Fast Flash* DVS Policy.

Share of Overhead Power Consumption at different frequencies Lastly, Figure 9.11 shows the magnitude of overhead power consumption of *CPI* and *COMP_DATA_FLASH* tracing in proportion to the power consumption without tracing. Only *CPI* and *COMP_DATA_FLASH* tracing is shown, as they cover the whole power consumption range of the three workloads.

This comparison is made to show the overhead power consumption share to the actual workloads at different CPU frequencies. Users that want to evaluate the usefulness of the task characterization model in combination with their application should use the absolute overhead power consumption values that have been shown in Figure 9.9.



(a) Tracing the *CPI* profiling counter.



(b) Tracing flash access with comparators.

Figure 9.11: Overhead power consumption of enabled tracing in proportion to the power consumption of workloads without enabling tracing. Illustrated with different workloads and CPU frequencies. Measured with Flash Wait State Adaption has been enabled and the *Fast Flash* DVS Policy has been selected.

Figure 9.11 shows that the share of overhead power consumption often gets smaller with a lower CPU frequency. Further, it is shown that the share is higher when the CPU has no load (*SLEEP*) and that the share is generally task dependent. Again, by looking at the workload *FLASH_ADD* it is visible that tracing with comparators has a big effect on the overhead power consumption if comparator matches occur.

Property	Profiling Counter (ECP)	Comparators (DTAOP)
Delay between Packet Generation and complete Timer Counter Recognition (min to max)	80MHz:260ns to 720ns and 13MHz:1440ns to 4260ns	80MHz:160ns to 570ns and 13MHz:800ns to 3600ns
Overhead Delay for Trace Start/Stop Utility Functions	80MHz: 5.26µs / 2.45µs	80MHz: 13.40µs / 4.06µs
Overhead Power Consumption Range at 80MHz	task dependence: 3 to 4.8 mW, increase due to counter activeness: 0.7mW	task dependence: 3.1 to 4.6 mW, increase due to counter activeness: 4.7mW

Table 9.1: Summary of the task characterization overhead.

10 Evaluation of Tracing and Energy Results

This section presents tracing and energy results of synthetic and the Bristol/Embecosm Embedded Benchmark Suite (BEEBS) tasks.

Thereby, the following questions will be examined further:

1. What do certain counter readings tell about a task behavior?
   - Is the task CPU intensive?
   - Is RAM and/or flash memory used significantly?
   - Are I/O units used?
   - Which operations are used?
2. Can certain counter readings alone or combinations be used to point to the Most Energy Efficient CPU Frequency (MEECF) setting?
3. What is the overhead in terms of tracing steps for selected task properties?

10.1 Register, RAM or Flash intensive Workloads

To start off analyzing the potential of the task characterization model to indicate the performance utilization, an experiment trying to mainly use a single "memory" type (RAM, Flash or no memory only Register) at a time is performed. The synthetic tasks are also performed with different math operations and an unsigned 32 Bit variable.

Task Information at the default CPU Frequency (80 MHz)

Figure 10.1 visualizes the profiling counter measurements in proportion to the CPU cycles at a CPU Frequency of 80MHz (see normalized counter calculation in Section 8.4.2).

Note: Due to the normalization with the total count of traced cycles, e.g., a normalized *LSU* counter of 0.5 implies that a task spends 50% of all cycles for load/store instructions. A normalized flash comparator counter of 0.03 implies 0.03 flash memory accesses per cycle.

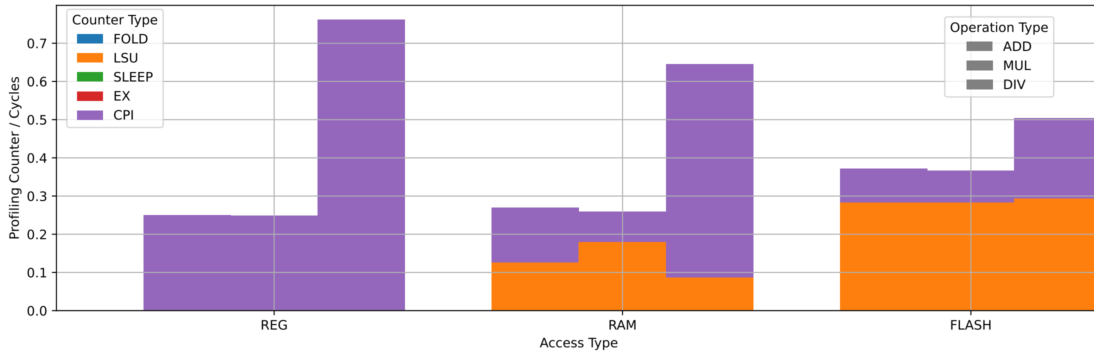


Figure 10.1: Normalized profiling counter results measured with tasks of different memory access types and math operations. The measurements were performed at a CPU frequency of 80MHz.

Looking at Figure 10.1, the *LSU* counter correlates with the access latency of **RAM** and **Flash**. With a higher memory access latency, the counter gets bigger as more cycles are needed for instructions, which use more cycles at higher frequencies. To be certain that the *LSU* counter results for the RAM benchmark are caused by the RAM access, the benchmark was traced via the comparator trace method, as seen in 10.2. Furthermore, even though the workload *REG* needs to load the workload instructions from flash, they are not tracked by the *LSU* counter. Looking at the *CPI* counter, it positively correlates with the cost of operations. The operation *divide* needs 2 to 12 cycles to complete, whereas *add* or *multiply* only need around 1 cycle [61, sec. 3.3.1].

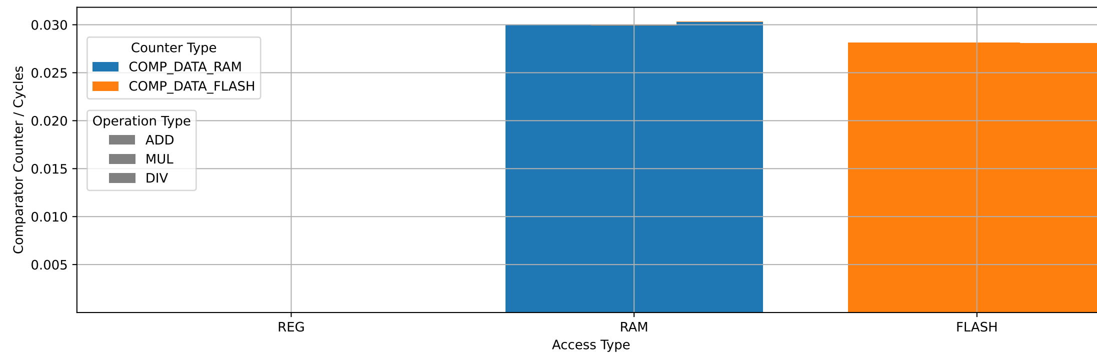


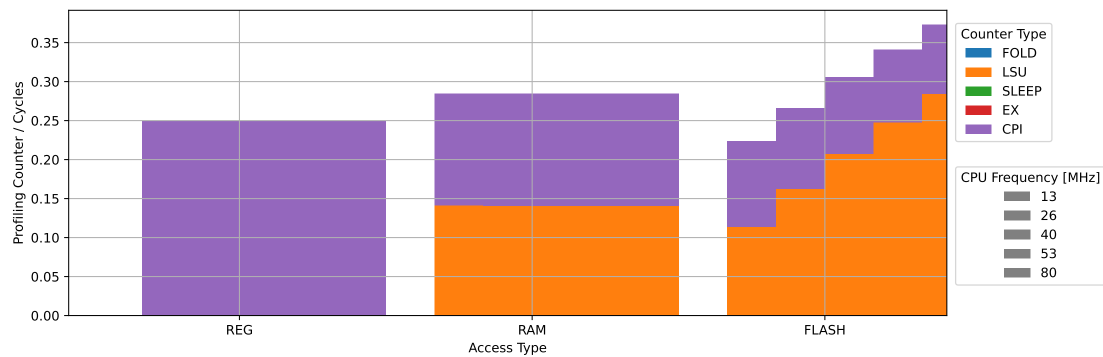
Figure 10.2: Normalized flash/ RAM data access measured for tasks with different memory access types and math operation types. The task properties are measured at a CPU frequency of 80MHz.

Comparing the results for flash or RAM access with Figure 10.2, only the flash benchmark shows flash matches and only the RAM benchmark shows RAM access. Meaning the *LSU* counter reacts to both RAM and flash access. This confirms the profiling counter description (see Table 4.1), as it counts on additional cycles needed for load/store instructions, which are needed to access both flash and RAM.

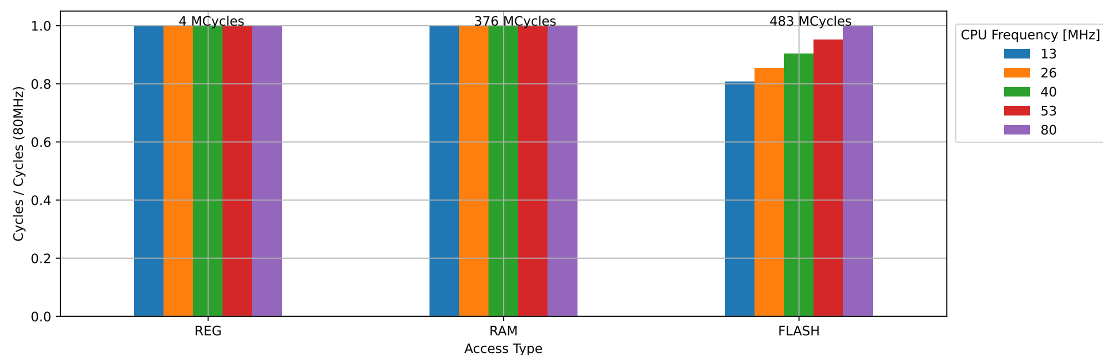
Figure 10.2 shows that the comparator counter results do not differ with the different math operations. This is probably because every math operation needs to load the same number of operands before the calculation and calculates a single result that also requires the same number of writes to *RAM*.

Comparing Tracing Results at different CPU Frequencies

Figure 10.3 shows the normalized counter values at different CPU frequencies in part (a). Part (b) shows the number of cycles needed to perform the respective tasks at different CPU frequencies in proportion to the cycles needed at a CPU frequency of 80 MHz. This proportion is used to highlight and compare the cycle reduction across different tasks and different CPU frequencies.



(a) Normalized profiling counter results.



(b) Cycles at different CPU frequencies in proportion to cycles at a 80MHz - with annotated number of cycles at 80 MHz.

Figure 10.3: Profiling counter and cycle trace results of tasks performing ADD operation with different memory access types. The measurements were performed at different CPU frequencies, with enabled Flash Wait State Adaption and *Fast Flash* DVS Policy.

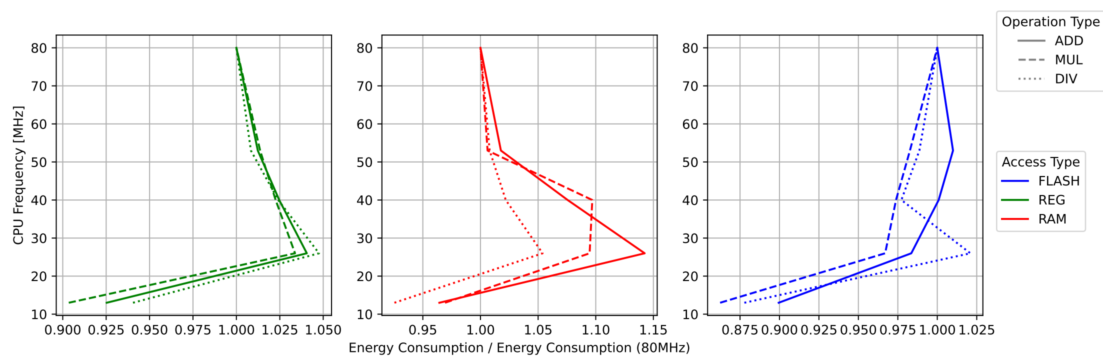
It is shown that with lower CPU frequencies, the *LSU* counter proportion and the total cycles are only reduced with workloads that intensively use the flash memory. The reduction in traced cycles is a result of reducing the FWS as described in Section 8.4.1. Per lowered FWS the total number of cycles is reduced by around 5%. This saves around 97 million cycles at the lowest frequency of 483 million cycles in the default configuration.

Most Energy Efficient Frequency and Policy

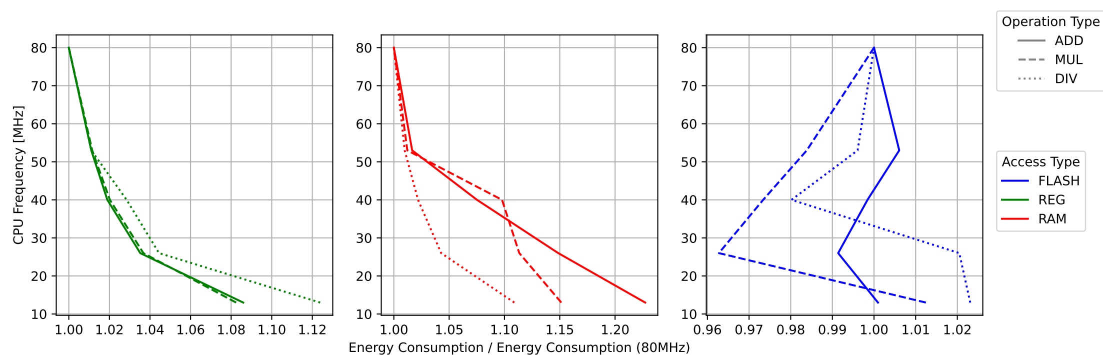
Before considering the meaning of Figure 10.4, it should be made clear which data is shown. The Figure is a result of current measurements with enabled Flash Wait State

Adaption and DVS. This is the same workload which is traced for Figure 10.1. All current measurements of this chapter are measured without performing task characterization. For each workload configuration the total energy is calculated. To show the MEECF setting, each energy result is divided by the total energy consumption at a CPU frequency of 80 MHz. An energy proportion value lower than 1 means the configuration is more energy efficient. Configurations with a higher value than 1 are less energy efficient.

NOTE: This workload intensively only provokes a certain behavior of the target system. The correlation between the traced task properties and the energy measurements might therefore diverge from workloads that represent more realistic tasks. For example, tasks that are frequently interrupted or do more a mix of instructions.



(a) DVS Policy: Low Voltage.



(b) DVS Policy: Fast Flash.

Figure 10.4: Energy consumption at different CPU frequencies in proportion to the energy consumption at 80MHz. The proportion results are grouped by the math operation type and memory access type, and separated by the DVS Policy. The energy consumption is measured with enabled FWSA and DVS.

Consequently, Figure 10.4 shows that tasks working mostly with RAM and no frequent memory access (*REG*) scale well with the CPU frequency. Meaning the highest frequency is executed most energy efficient. Only if the policy *LOW_ VOLTAGE* is used, the CPU voltage can be lowered to the frequency setting of 13MHz and is therefore most energy efficiently executed (compare with Figure 8.5).

Workloads requiring mostly flash accesses (*FLASH*) are most energy efficient at lower frequencies as the lower FWSs reduce the total cycles used. It should be noted that the *DIV* math operation, which takes more cycles to be performed and is more CPU demanding, is executed most energy efficiently at a higher CPU frequency than *ADD* or *MUL*. This shows that energy consumption is dependent on many task properties.

Concluding and regarding the second question of the beginning of this chapter, comparing the figures in this section about the traced task properties and the most energy efficient frequencies, a relation could be shown by:

Higher Energy Efficiency at Lower Frequencies	**Higher Energy Efficiency at Higher Frequencies**
• a noticeable flash data access count and no RAM access (see Figure 10.2) • a high *LSU* count (around 30 %) (see Figure 10.1) • a reduction in cycles needed at lower CPU frequencies (see Figure 10.3)	• no memory access or only RAM access (see Figure 10.2) • higher *CPI* count (see div in Figure 10.1) • no reduction in cycles (see Figure 10.3)

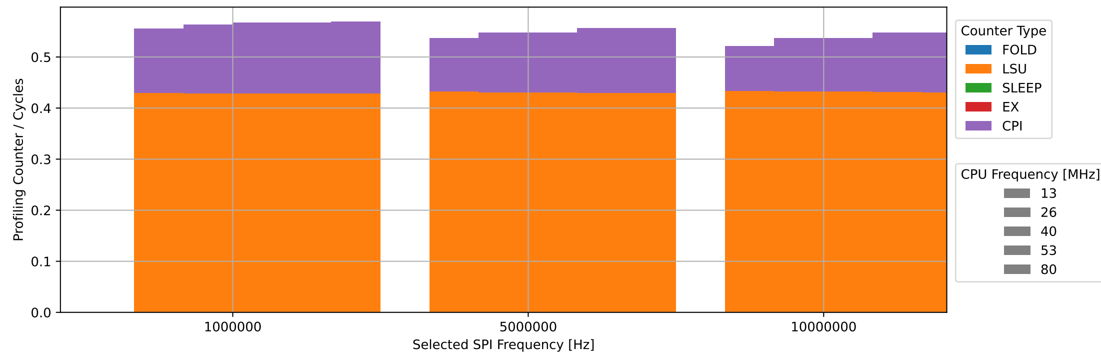
Table 10.1: Potential indicators for more energy efficient frequency settings.

10.2 I/O Workload

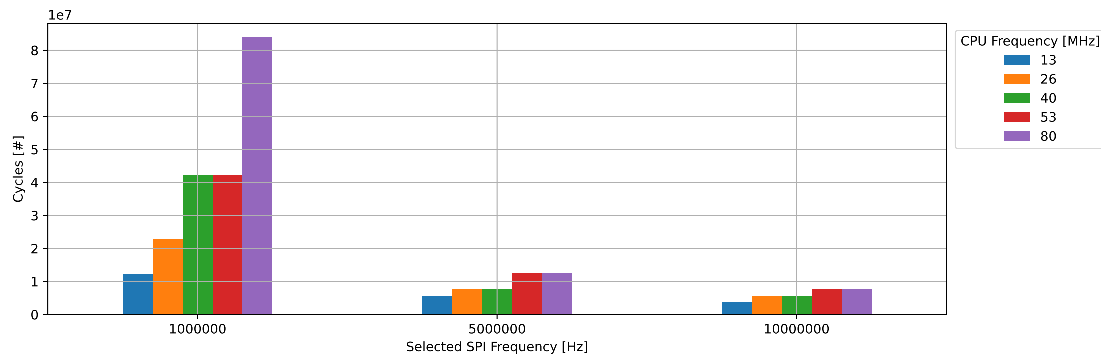
In this section a workload that mainly uses the serial communication interface SPI is examined. The usage of SPI is common on embedded devices and enables, e.g., to use external sensors, to persist data to external memory or send data wirelessly over radio interfaces. These are only a few examples of tasks that are limited by I/O access, which should execute more efficiently at a lower core frequency [10, sec. 4.1.2].

SPI Communication Intensive Workload

The SPI benchmark consists of transferring data via the low-level SPI driver, which uses the hardware SPI peripheral of the MCU. The benchmark is performed with different SPI speeds and with the same fixed number of transaction iterations. Each transaction consists of acquiring the bus, transferring 1024 Bytes of data and releasing the bus again.



(a) Normalized profiling counter results.



(b) Cycles needed for execution.

Figure 10.5: Trace results for the SPI intensive workload executed at different SPI frequencies and CPU frequencies. The task properties are measured with enabled Flash Wait State Adaption and *Fast Flash* DVS Policy.

Part (a) of Figure 10.5 shows that across all SPI frequencies at a CPU frequency of 80 MHz, the normalized *LSU* counter stays fairly high at around 42 percent. Furthermore, the profiling counter results are not affected by the different selected SPI frequencies.

Looking at the workload implementation, the *LSU* result of more than 40% is probably caused by the data that has to be loaded from the input buffer or has to be stored to an output buffer to transfer data over the SPI driver. Even though all workload configurations are performed with the same number of iterations, the different SPI frequencies affect the number of cycles that are needed to transfer the same number of data. Regarding the task behavior across different CPU frequencies, part (b) of Figure 10.5 shows a reduction of cycles with lower CPU frequencies for all traced SPI frequencies.

To investigate the reason for the reduced cycles at lower CPU frequencies, a test is performed that reveals the actual configured SPI frequencies that are hidden behind the selected ones at different CPU frequencies.

Selected SPI frequency **CPU Frequency**	**1 MHz**	**5MHz**	**10MHz**
80 MHz	0.625MHz	5MHz	10MHz
53 MHz	0.833MHz	3.33MHz	6.66MHz
40 MHz	0.625MHz	5MHz	10MHz
26 MHz	0.833MHz	3.33MHz	6.66MHz
13 MHz	0.833MHz	3.33MHz	6.66MHz

Table 10.2: Actual SPI frequency dependent on the CPU and selected SPI frequency.

Table 10.2 shows that a cycle reduction is not caused by an increase of the actual SPI frequency. Further, performing the SPI benchmark with disabled FWSA proves that the cycle reduction is not caused by a reduction of FWS.

By investigating the implementation of the SPI driver, it is shown that each byte of the input buffer is written to a peripheral data register. After each byte the CPU needs to wait for the completion of the byte transmission by polling a peripheral status register. Therefore, the number of total cycles required to transfer a fixed number of data bytes is highest when the difference between CPU and SPI frequency is highest.

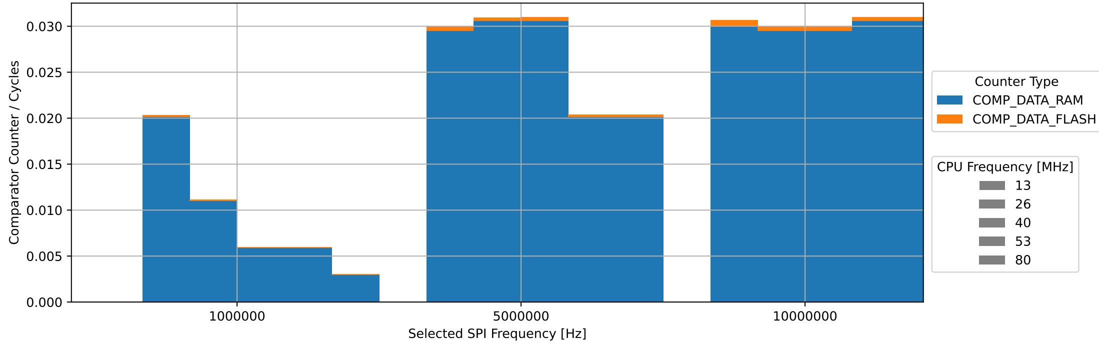


Figure 10.6: RAM/flash data access per cycle measured with the SPI workload. The results are grouped by SPI frequency and CPU frequency. The task properties are traced with enabled FWS and *Fast Flash* DVS Policy.

Figure 10.6 shows almost no flash usage. This indicates that the high *LSU* property is caused by many load/store operations on RAM. The normalized comparator counter results do not get higher than the value 0.03, which is also seen by Figure 10.2. This is caused by the bandwidth bottleneck of the TPIU mentioned in Section 7.2.2.

With the 1MHz SPI configuration, the increasing normalized RAM access counter with lower CPU frequencies is caused by the reduction of cycles, shown in part (b) of Figure 10.5, and a consistent non-normalized RAM access counter. With the 10MHz SPI configuration, the consistent normalized RAM access counter at all CPU frequencies is caused by the reduction of cycles and a reduction of the non-normalized RAM counter values with lower CPU frequencies. For the 5MHz SPI configuration, the normalized RAM counter dip at the CPU frequencies of 53 and 80 MHz in comparison to 26 and 40 MHz are caused by the reduction of cycles for 26 and 40 MHz and the same amount of non-normalized RAM counter values for both CPU frequency groups.

SPI Workload Energy Results

Figure 10.7 is based on the same methodology as Figure 10.4 and shows the most energy efficient CPU frequency and DVS Policy setting dependent on the configured SPI frequency.

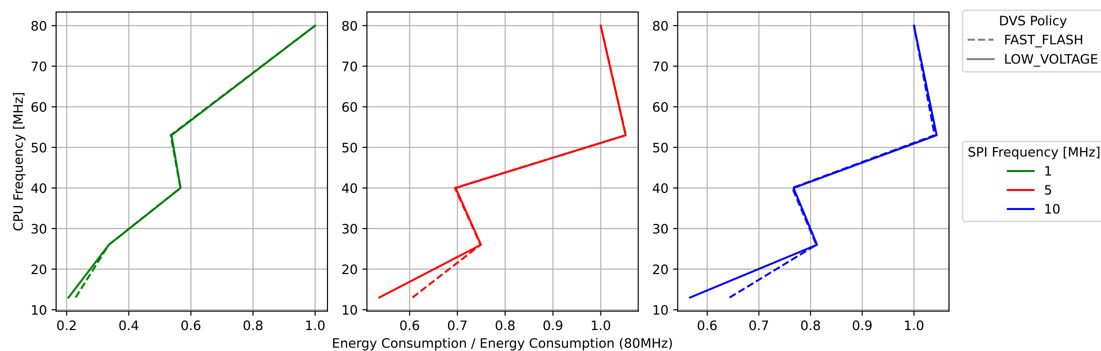


Figure 10.7: Energy consumption at different CPU frequencies in proportion to the energy consumption at 80 MHz measured with the SPI workload. The proportion results are grouped by the DVS Policy and the selected SPI frequency. The energy consumption is measured with enabled FWSA and DVS.

As already suggested by the saved cycles of Figure 10.5, the cycle reduction also reduces the total energy consumption. When using the *Low Voltage* DVS Policy, even more energy can be saved at the lowest CPU frequency.

SPI Task Differentiation

In Section 10.1 a task is shown that performs many RAM access and is most energy efficiently executed at the highest frequency. With the SPI tracing and energy results, a high normalized *LSU* counter of more than 42% is seen that is almost completely caused by RAM usage, but the SPI task is executed most energy efficiently at a lowermost CPU frequency. Therefore, using a high *LSU* counter as an indicator to a more energy efficient CPU frequency setting is questionable.

Fortunately, with the comparators it is also possible to match memory addresses that point to specific peripheral registers.

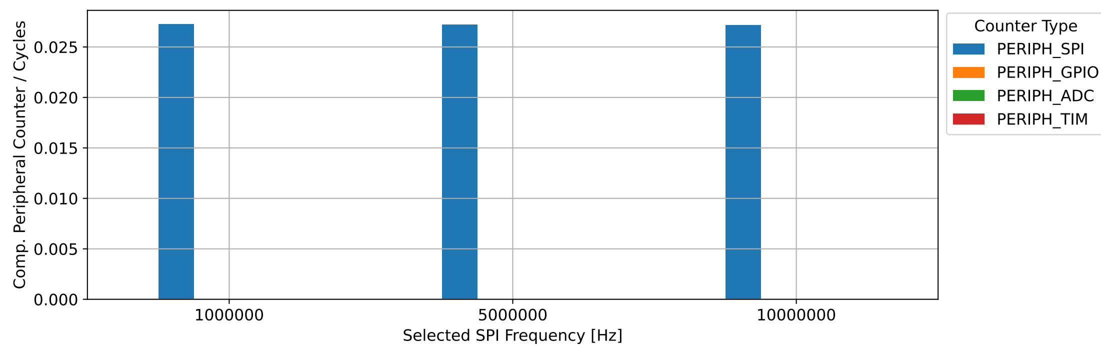


Figure 10.8: Peripheral data accesses per cycle of the SPI intensive workload. The data accesses are measured with DAM, as seen in Section 6.3.2.

Figure 10.8 shows the trace results with comparators that observe the address range of the SPI peripheral registers (see Table 6.1). The task characterization model can detect tasks with SPI usage. Regarding the height of the peripheral access values, future experiments need to be performed with tasks that do a mixture of operations. Thereby, it can be investigated whether the counter height can be used to detect different SPI frequencies or even as an indicator for selecting a lower MEECF setting. The presented task tracing is a proof-of-concept.

Concluding this section, task properties that potentially indicate a lower MEECF setting are:

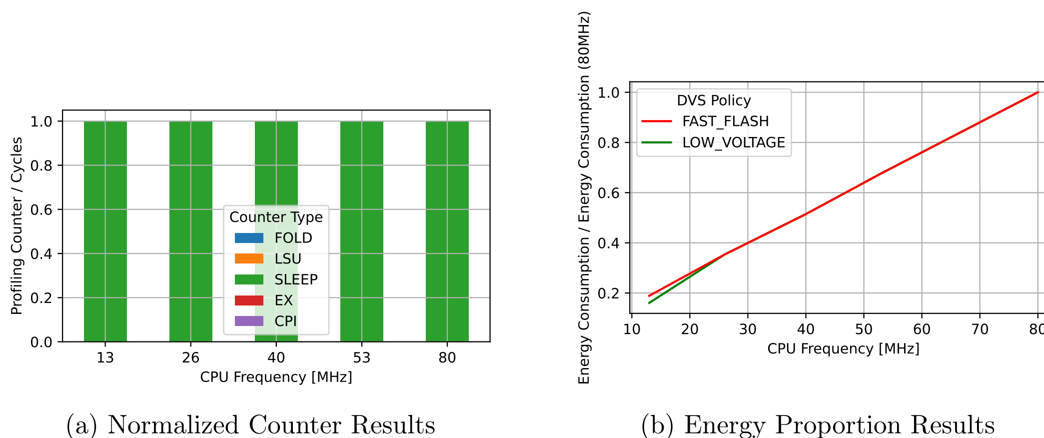
- a reduction in cycles at lower CPU frequencies (see Figure 10.5)
- a high SPI peripheral access

10.3 Inactivity

Some applications may need to wait for certain events to happen, in the meantime the system might not need to perform any calculations and can therefore idle for a certain amount of time. Waiting for an event to finish can be done via sleeping until interrupted.

SLEEP intensive Workload

Figure 10.9 is a result of a benchmark that sleeps for 1 second. Part (a) shows the normalized profiling counter results. Part (b) shows the relation between the energy and CPU frequency, which reveals the most energy efficient frequency.



(a) Normalized Counter Results

(b) Energy Proportion Results

Figure 10.9: SLEEP intensive workload properties measured at different CPU frequencies and different DVS Policy.

Part (a) shows that the task characterization model detects whether the CPU is sleeping, as the sleep cycles show a share of 100% to all cycles. Therefore, the most energy efficient frequency is the lowest, as a higher CPU activity is not contributing to the tasks progress. Workloads that sleep most of the time probably save more energy if the system is put into a low power mode after the CPU intensive workload is finished. Future research with tasks that only sleep part of the task execution need to investigate at which sleep counter height a higher energy efficient CPU frequency is preferable.

Concluding and regarding the second question of 10, the figures in this section show that a high normalized *SLEEP* counter is an indicator of a lower MEECF setting.

10.4 Application-focused Tasks with BEEBS

The prior sections focus on synthetic benchmarks that explore corner cases like only provoking peripherals or using mainly one memory type at a time. We now want to

evaluate how the task characterization model behaves under more application-focused loads. Therefore, the benchmark suite BEEBS is selected (see Section 8.3).

First, the ground truth task properties are investigated that are not collected via the implemented task characterization model. This expands the knowledge base for developers working with the benchmark suite in the future and justifies the selection of the benchmark.

Note: The following traced task properties for the BEEBS tasks are most often viewed at a CPU frequency of 80 MHz. Devices with static clock configuration typically select the highest CPU frequency as it provides the best performance. Performing tracings at lower frequencies is left for future work.

10.4.1 Model-Independent Task Properties

To start off this section, Figure 10.10 shows the time needed to perform an iteration dependent on the selected task. This shows that the benchmark suite includes a wide range of task complexity. The magnitude in task complexity is further increased considering that the Figure shows logarithmic scale on the y-axis.

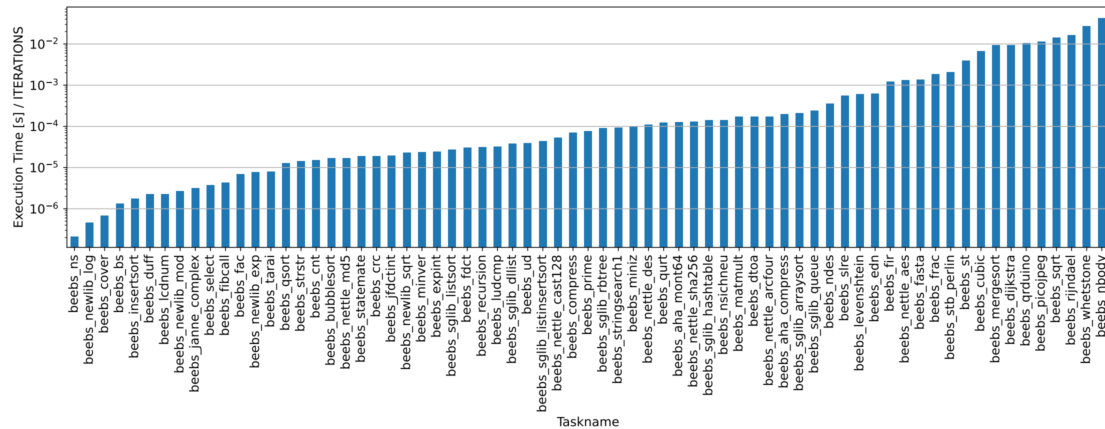


Figure 10.10: Execution time per iteration at a CPU frequency of 80 MHz. The y-axis is shown in logarithmic scale and the tasks are sorted by the highest value.

Next to the duration in time, the range of power consumption is investigated. Figure 10.11 shows that power usage at a CPU frequency of 80 MHz ranges from 42.7 to 56.6

mW and at the lowest frequency from 6.6 to 11.4 mW. It has to be noted that the power consumption results are calculated from current averages of a complete trace.

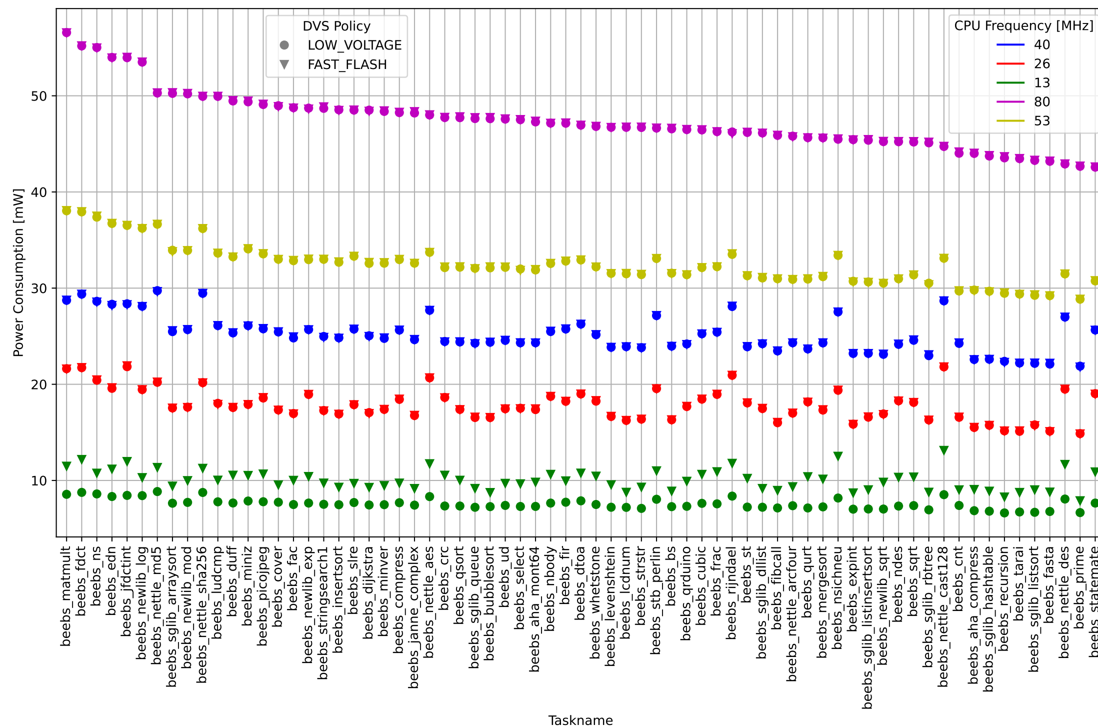


Figure 10.11: Average power consumption at different CPU frequencies. Grouped by CPU frequency and the DVS Policy. The power consumption was measured with enabled Flash Wait State Adaption. The tasks are sorted by the highest power consumption at 80MHz.

As the tasks are sorted by the highest power consumption, it is visible that some tasks lower their power consumption differently at lower frequencies than the neighboring ones. (see tasks *beebs_nettle_md5*, *beebs_nettle_sha256*, *beebs_stb_perlin*, etc.)

Continuing with the benchmark suite properties, the ground truth about the most energy efficient frequency/voltage setting per task is shown in Figure 10.12. The energy proportion is calculated in the same way as already described prior to Figure 10.4.

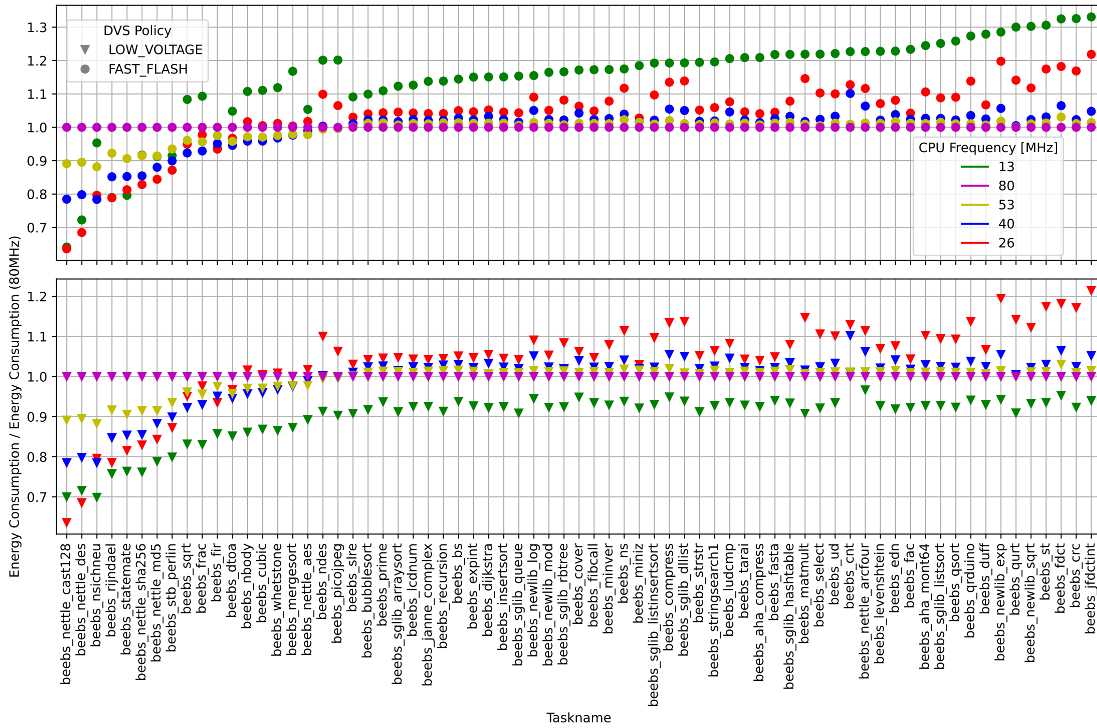


Figure 10.12: Energy consumption at different CPU frequencies and different DVS Policy in proportion to energy consumption at 80 MHz. The energy was measured with enabled Flash Wait State Adaption and DVS. The tasks are first sorted by the biggest energy consumption saving and secondly by the biggest energy consumption increase.

Regarding the DVS Policy, Figure 10.12 shows with enabled *Low Voltage* policy it is most energy efficient in most cases to select the lowest CPU frequency. If a *Fast Flash* is preferred, a grouping between tasks that save energy with lower CPU frequencies (from left to task *beebs_ picojpeg*) and tasks that do not save energy with lower frequencies (from task *beebs_ slre* to the right) is seen.

To better understand why the left group of tasks is more energy efficient at lower CPU frequencies than the right group of tasks, most of the following plots have a task ordering that is similar to Figure 10.12 (*task order of energy saving potential*). The same task ordering makes it on one hand easier to differentiate between the group of tasks that save energy at lower CPU frequencies at tasks that do not. Furthermore, task behaviors for a specific task are easier to compare between different plots if the tasks keep their position on the x-axis.

With the knowledge of the energy saving potential of Figure 10.12, it is appropriate to also mention the overhead power consumption results for the tracing utility from Section 9.3 applied to the BEEBS tasks. To give a comprehensive statement, only the overhead power consumption for the load intensive workloads of Section 9.3 at a CPU frequency of 80 MHz are used and applied to a power consumption mean between all BEEBS tasks of Figure 10.11. The mean power consumption between all BEEBS tasks can be seen in Figure 10.29, which is 47.423mW at the highest here measured CPU frequency. Each profiling counter tracing increases the average power consumption by 6.32 % ($\frac{3mW}{47.423mW}$) to 8.01 % ($\frac{3.8mW}{47.423mW}$). The DAM increases the power consumption by 6.5% ($\frac{3.1mW}{47.423mW}$) to 16.44 % ($\frac{7.8mW}{47.423mW}$). In a real setting multiple different counters might have to be traced after another, but a tracing is ideally not performed always.

The last task characterization model independent metric is the measure of time. Figure 10.13 shows the proportion of time to the highest CPU frequency (80 MHz).

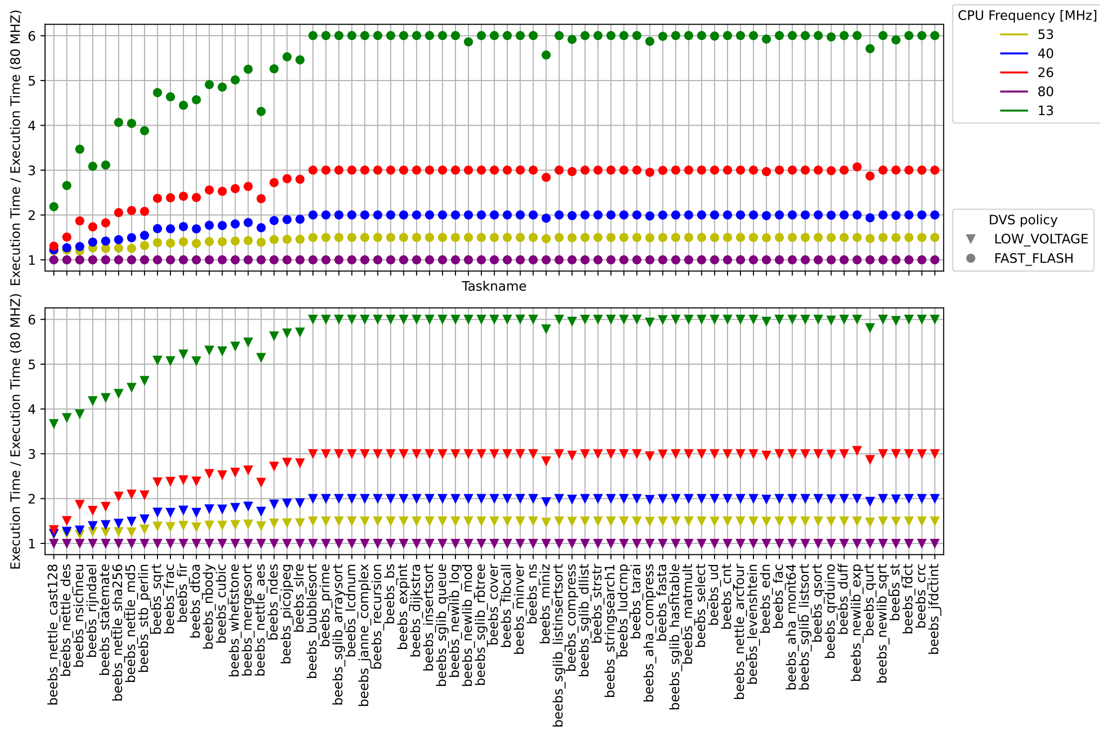


Figure 10.13: Execution time increase of different CPU frequency to 80 MHz. The execution time has been measured with enabled Flash Wait State Adaption and different DVS Policys (upper/lower figure). The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

It is shown that the leftmost group of tasks does not increase the task duration proportional to the CPU frequency reduction like most tasks of the rightmost group.

The illustration of time grouped by the DVS Policy further highlights the risk of always selecting the *Low Voltage* policy. Even though the voltage reduction ensures the highest energy efficiency at the lowest CPU frequency for all tasks (see Figure 10.12), the execution time for the rightmost group of tasks is still increased by the factor of 6.

Further, Figure 10.13 and 10.12 show that frequency scaling is a double-edged sword. On one hand, a frequency lowering can increase the energy efficiency and thereby increase the execution time at a lower factor than the frequency decrease. On the other hand, a misconfigured CPU frequency can result in an energy increase of up to 30% and increase the execution time by a factor of 6 (as seen by most tasks of the rightmost group).

Coming back to the observation of Figure 10.11 about specific tasks that have higher power consumption than their neighbor tasks at lower CPU frequencies. As seen by Figure 10.12 and 10.13, these are the same tasks that save energy when lowering the CPU frequency and do not increase their execution time at the same factor as the frequency reduction. This probably has something to do with the switching activity α (mentioned in Equation 2.2), which increases at lower CPU frequencies because of a lower bottleneck.

To conclude, this subsection justifies the usage of BEEBS as a base to evaluate the traced task properties with. BEEBS offers tasks that save energy and execution time at lower CPU frequencies and tasks that are most energy efficiently performed at the highest CPU frequency.

10.4.2 Tracing Inaccuracies

In the following sections the tracing results for the BEEBS tasks will be evaluated as an indicator of when CPU frequency should be reduced to increase the energy efficiency. To estimate the soundness of the tracing results, a notice about the fluctuation of the tracings is given. Figure 10.14 shows the standard deviation of counter readings in proportion to the mean of the counter readings per task among the 10 tracing repetitions. The proportion is used because the standard deviation is dependent on the absolute counter readings and would create a false impression about tasks that have a higher execution time.

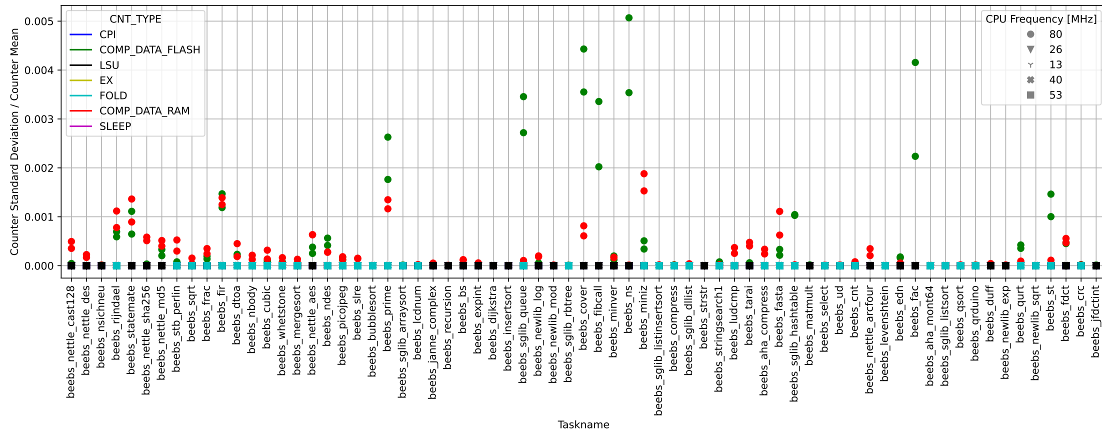


Figure 10.14: Standard deviation of measured counters in proportion to mean of the measured counters that is calculated with ten repetitions. The proportion is grouped by the counter type and the CPU frequency. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

Regarding the semantics of the standard deviation proportion, it is visible that only measurements taken at a CPU frequency of 80 MHz and with the comparator trace method show a fluctuation in the counter readings. This is probably caused by the nature of the different address offsets in the DTAOPs and the SWO is a bottleneck to packet conveying. Therefore, the following results traced due to DTAOP packets will also be illustrated with their standard deviation values, as seen in Figure 10.18 by the black caps. But the standard deviation proportion is quite small with many values being under 0.2%.

10.4.3 Tracing the Cycle Amount

The prior sections (like Section 10.1) show that the cycle saving potential is a good measure to find the most energy efficient frequency. Therefore, the evaluation of the tracings results is started by looking at the count of cycles at different CPU frequencies.

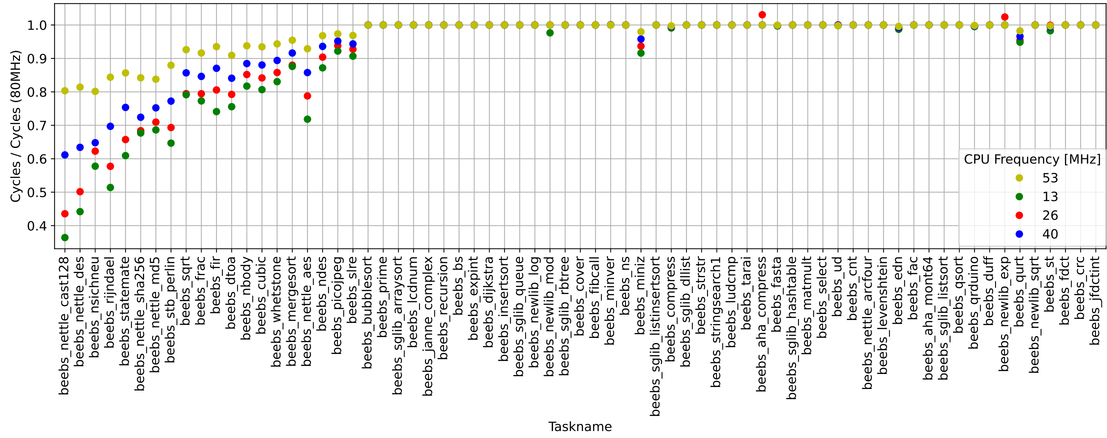


Figure 10.15: Cycle count at different CPU frequencies in proportion to the cycle count at 80 MHz. The measurements were performed with enabled Flash Wait State Adaption and the *FAST FLASH* DVS Policy. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

Figure 10.15 indeed shows that there is a correlation between the count of cycles that certain tasks can save at lower frequencies and the actual energy savings, as seen in Figure 10.12. Table 10.3 shows the linear correlation of the saved cycles to the saved energy grouped by the CPU frequencies.

Selected Group	**Linear Correlation (Pearson)**
FREQ:13MHz	0.901
FREQ:26MHz	0.915
FREQ:40MHz	0.970
FREQ:53MHz	0.989

Table 10.3: Linear correlation of all BEEBS tasks between the saved cycle proportion of Figure 10.15 and the energy proportion of Figure 10.12 and grouped by the CPU frequency.

The *pearson* correlation can be explained as how close points are to a fitted line and its value ranges from -1 to 1, with 0 meaning no correlation at all. A positive correlation value means that the two set of values are evolving into the same direction, meaning if values of one set get bigger the values of the other set also show an increase. A negative correlation means that when values of one set get smaller values of the other set get bigger.

Regarding the correlation outcome shown in Table 10.3, the calculated correlation values get smaller with the decrease in grouped CPU frequency. This is probably on one hand caused by tasks like *beebs_miniz* that save cycles, but not energy with lower CPU frequencies. On the other hand, tasks from *beebs_sqrt* to *beebs_picojpeg* do save energy with lower CPU frequencies, but they do not save the most energy with the lowest frequency. Nonetheless, it is shown that the saved cycles strongly correlate with the more energy efficient frequency settings with enabled *Fast Flash* DVS Policy. This means that the saved cycles at different CPU frequencies can be used as the ground truth for energy reduction potential.

Is a lower CPU frequency only more energy efficient if cycles are saved due to lower FWS?

As seen in Section 10.2, cycles can also be saved with disabled Flash Wait State Adaption at lower CPU frequencies due to asynchronous communication. To answer this section's question, another measurement was performed with the suite benchmarks and disabled Flash Wait State Adaption, as seen in figures 10.16 and 10.17.

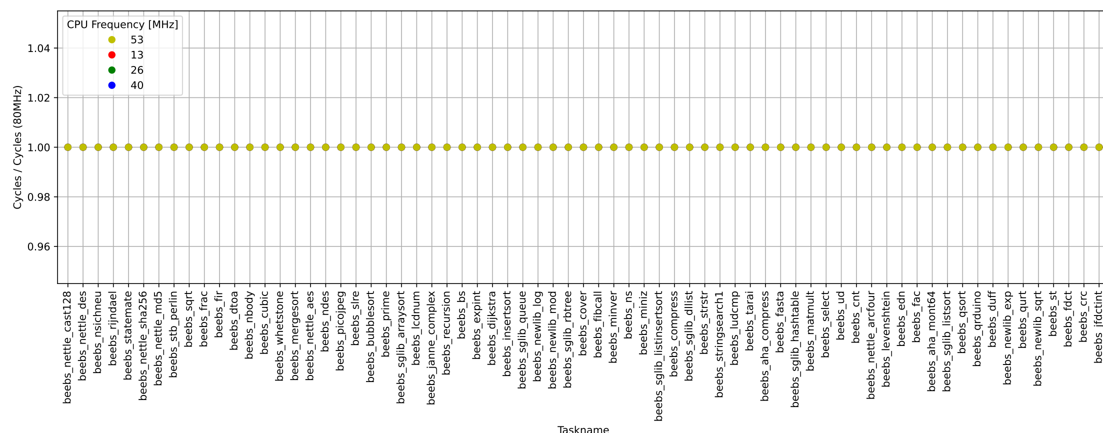


Figure 10.16: Cycle count at different CPU frequencies in proportion cycle count at 80 MHz. The cycles were measured with disabled Flash Wait State Adaption and configured *FAST FLASH* DVS Policy. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

Figure 10.16 shows than no cycles are saved with disabled Flash Wait State Adaption when lowering the CPU frequency. The BEEBS tasks are solely computation tasks

without time dependencies (timeouts, waits, etc.) and "other factors such as I/O and peripherals [are] excluded for portability [20, sec. 2]". In more complex applications with timers and I/O there would still be cycle potential with disabled FWSA. The evaluation of more complex systems was not the goal of this thesis but should be investigated in future work.

All synthetic tasks with configured *Fast Flash* DVS Policy are only more energy efficient at lower frequencies if cycles are saved. Further, it is interesting whether this is also the case for the BEEBS tasks.

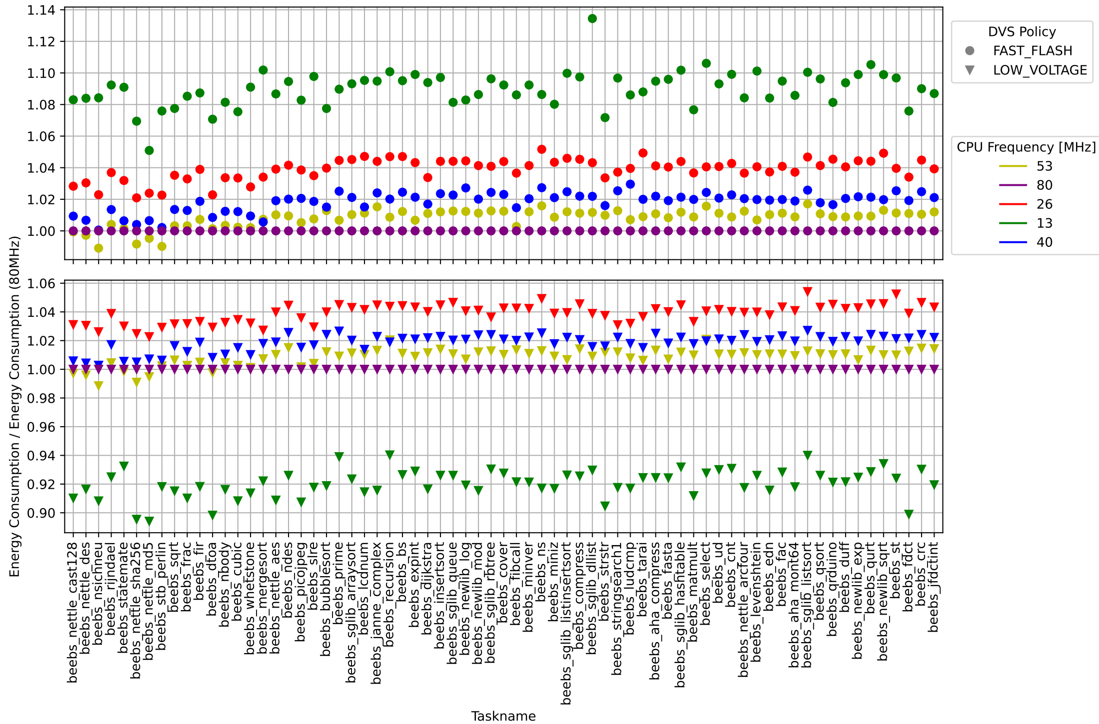


Figure 10.17: Energy consumption at different CPU frequencies in proportion to energy consumption at 80 MHz. The measurements were performed with disabled Flash Wait State Adaption, different DVS Policys and enabled DVS. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

Part (a) of Figure 10.17 indicates that very few tasks also save energy without lowering the FWS (see tasks *beebs_nettle_sha256*), even if this energy reduction is small at around 1%. Part (b) shows that the voltage reduction once again has a big effect on the total energy consumption with energy savings up to 10%, even without saving cycles. But

the missing FWS reduction ensures that the task duration increases proportional to the CPU frequency reduction factor.

10.4.4 Tracing with Comparators

This subsection evaluates the task properties traced with the DWT comparators, which can be used to separately observe the data access to RAM and flash memory.

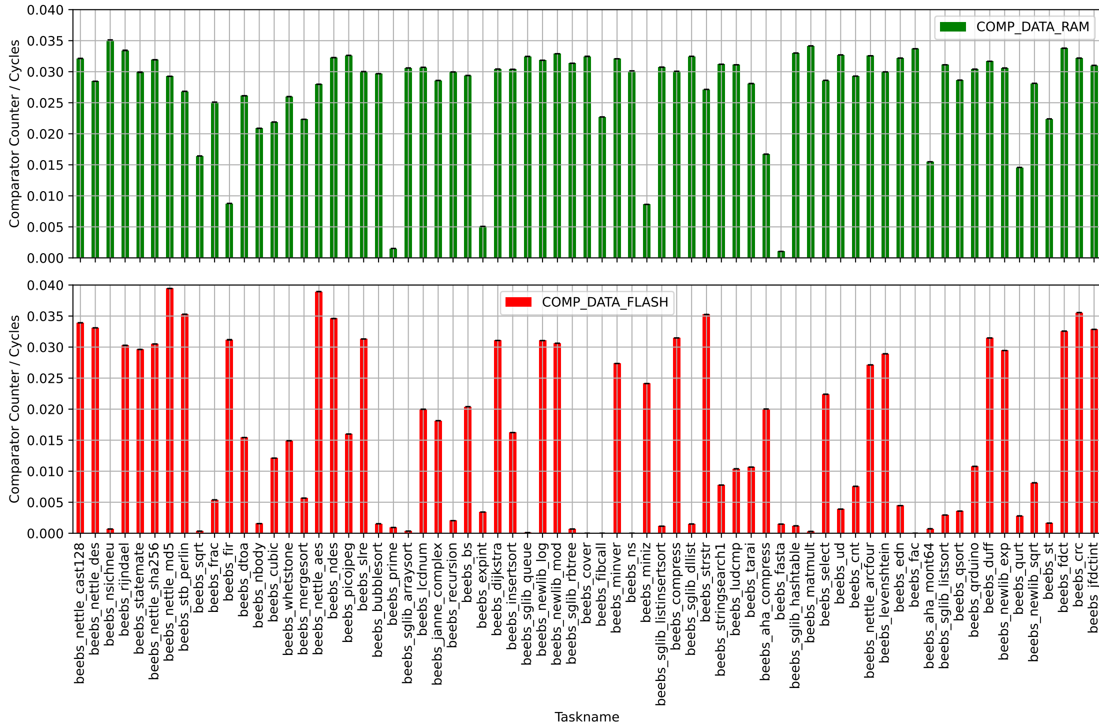


Figure 10.18: Memory accesses per cycle measured at a CPU frequency of 80 MHz. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

Before going into detail about certain Data Address Matching (DAM) results and their meaning regarding task type, looking at Figure 10.18 shows that both memory access types neither correlate with the cycle-savings shown in Figure 10.15. This indicates that just tracing the access to memory is not an indicator to save cycle or energy at lower CPU frequencies. Furthermore, it is visible that more tasks have a high RAM comparator counter of 0.03 than there are tasks that have a high flash comparator counter. This

indicates that the RAM is used more frequent than the flash and that the RAM counter is more affected by the bandwidth bottleneck of the SWO connection.

Inspecting Source Code of Selected Tasks

Before investigating the low correlation between the memory access and the cycle-saving potential of tasks, the source code written in C of selected tasks is observed. Thereby, tasks with similar counter type results are compared.

RAM As the tasks with a high comparator counter are limited by the packet conveying bottleneck of the TPIU, it makes more sense to look at the fewer tasks with a low RAM counter. Thereby, the tasks *beebs_prime* and *beebs_fasta* and also *beebs_fir* and *beebs_miniz* will be examined further regarding their RAM usage.

The tasks *beebs_fasta* and *beebs_prime* both have a low RAM comparator counter and a very low ROM footprint. The *prime* task uses only a single volatile variable at the end of one iteration, which is loaded to RAM avoiding the compiler optimization of the benchmark. There are no arrays that need to be loaded to RAM, therefore most operations can probably be held in the CPU registers. On the other hand, the *fasta* task definitely works on small data structures that are held in RAM.

The tasks *beebs_fir* and *beebs_miniz* have similar RAM counter readings at around 0.075. These are higher than the prior tasks, but half as small as the majority of tasks. The *fir* task performs a filter function that loads a huge data set from a *const* array (saved in flash) as an input, but actively calculates on an array in RAM. The *miniz* task performs a compression and decompression function. The input data is a *const* literal that is saved in flash, but the compression and decompression is performed on arrays that are allocated in RAM.

The comparison between the *prime* and *miniz* tasks show that the RAM counter height between very low and medium values is visible in source code.

Flash To evaluate the flash access matching ability, the tasks *beebs_fac* and *beebs_crc* are compared as they differ heavily in flash data access.

The *fac* task (low flash counter) has a very low footprint, has no data structures and performs a faculty calculation implemented via recursion. The recursion is probably the

reason why the RAM counter is very high at a value of 0.03. Only one literal is used, and no other variables have to be reloaded from flash.

The *crc* task (high flash counter) has a very low footprint and performs a cycle redundancy check. The algorithm loop uses a *const unsigned char* array that is stored on flash.

This very basic comparison of a low and a high flash counter task supports the working of the flash data address matching. Though a statement regarding the fine-grained difference in height can not be made.

Does a low RAM and low flash counter point to a higher MEECF?

The comparison of the RAM counter results from the sections 10.1 and 10.2 show that a high RAM access does not correlate with any CPU frequency setting. But the comparator counter results for no memory use show a higher MEECF.

The only tasks that have a very low RAM and flash usage are the tasks *beebs_prime* and *beebs_fasta*, which are already evaluated. As in Figure 10.12 they actually do perform most energy efficiently at the highest CPU frequency.

Do the tracing results differ across CPU frequencies?

As of completeness, in Figure 10.18 only the tracing results at a CPU frequency of 80 MHz are shown, but in Figure 10.15 cycle results across different CPU frequency are compared. Therefore, the question arises whether there is also a difference in tracing results for the comparator usage.

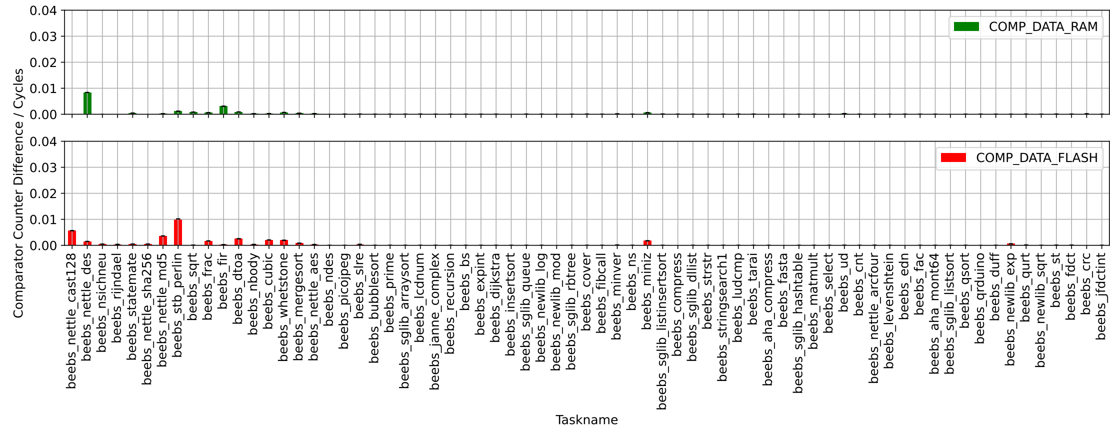


Figure 10.19: Difference of normalized comparator counter results (13MHz - 80MHz) with enabled *FAST FLASH* DVS Policy and enabled FWSA. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

Figure 10.19 shows the difference between tracing results of the CPU frequencies 13MHz and 80 MHz. It is noticeable that a difference is almost only present with the group of tasks that also show cycle savings, as seen in Figure 10.15. This difference is only visible because of the cycle reduction and therefore this additional information is redundant. The cycle count does not need to be traced by the timer counter feedback mechanism of Section 5.2, but can be read directly from the DWT register, which has a size of 32 Bit. Therefore, the information of saved cycles is easier to obtain and comes with a lower energy overhead.

Why does more flash access not indicate savable cycles?

Since the cycle savings of Figure 10.15 are a result of reducing the FWS at lower CPU frequencies, it could be argued that cycles are savable if the task shows a high flash data access.

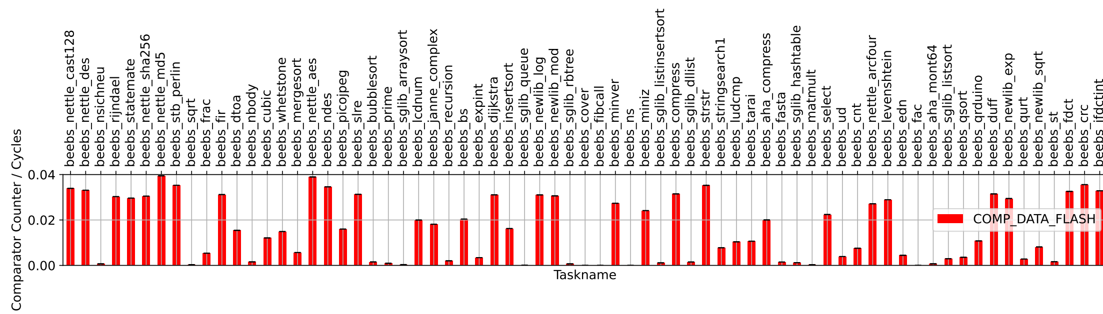
It is apparant, by comparing the flash data access results of Figure 10.18 and the saved cycles at different CPU frequencies of Figure 10.15, that not only those tasks that save cycles due to Flash Wait State Adaption have more flash access. Regarding the bandwidth bottleneck of the TPIU with DAM (see Section 7.2.2), it could be argued that the important difference in counter height is hidden. The important difference might be visible

with normalized values above 0.04. Nonetheless, some cycle-saving and non-cycle-saving tasks that also have a high flash access will be examined further.

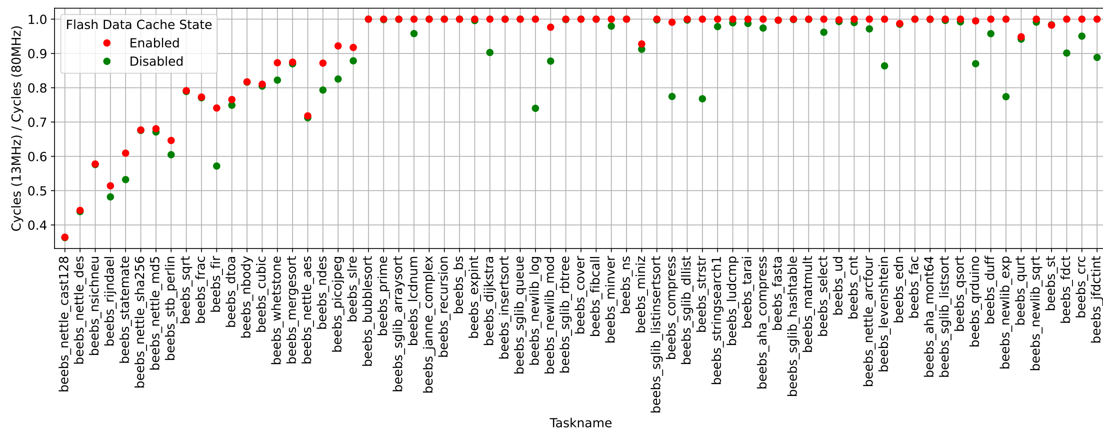
Tasks that do have a high flash count but no cycle savings are for example the *beebs_crc*, which performs a cycle redundancy check on a string saved in flash, the task *beebs_strstr*, which performs a substring search on a string that is saved in flash, or the task *beebs_newlib_log* that calculates a logarithm with float values that are saved in flash. These tasks have in common that they do work with literals or structures saved in flash, but do not require much flash space. The structures might be stored temporarily in flash cache for future access without the need of actual loads from flash and without the need to wait the minimum number of FWSs.

The embedded flash memory on the STM32L476RGT6 MCU is actually equipped with the Adaptive Real-Time Memory Accelerator (ART Accelerator) that comes with an instruction prefetch feature, a 1KByte instruction cache and a 256 Byte data cache [5, sec. 3.3.4 ART Accelerator]. With that in mind, tasks that show a high flash data access and save cycles due to Flash Wait State Adaption need to have a flash access behavior that the ART Accelerator cache is not able to accelerate.

Trying to verify that hypothesis, *beebs_nettle_cast128* performs symmetric-key block cipher encryption with access to 8 blocks of data stored in flash that each consist of 256 data points á 4 Bytes. The encryption algorithm also seems to access many values distributed across the used *const* arrays. This flash access behavior might also be important as caches work with the concept of spatial locality. Furthermore, *beebs_rijndael* for example also performs a symmetric-key block cipher encryption with at least 2 blocks of data saved in flash that each consist of 256 data points á 4 Bytes. To further prove the hypothesis about the flash data cache, a further measurement for the benchmark tasks was performed with disabled data flash cache. The results are shown in part (b) of Figure 10.20.



(a) Normalized flash access, as in Figure 10.18.



(b) Proportion of saved cycles at a CPU frequency of 13 MHz at different flash data cache states.

Figure 10.20: Comparison between the count of saved cycles at 13MHz with different flash cache states and the normalized flash access at an CPU frequency of 80 MHz. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

Part (b) of Figure 10.20 shows that many tasks right from task *beebs_picojpeg* are saving more cycles with disabled data flash cache (green dots) than with both enabled instruction and data flash cache (red dots). Part (a) of Figure 10.20 shows that the additional saved cycles due to disabling the data flash cache are often only present with a medium/higher flash comparator counter. The bar height alone does not define the amount of cycle saving improvement, as seen by the three rightmost tasks (*beebs_crc* etc.). This might be caused by the hidden difference above 0.03 or by the flash access behavior.

Tasks that were already saving cycles prior to the data flash cache disabling partially show a further cycle saving improvement. Some of them also show no improvement, as

seen by the three leftmost tasks for example. The lack of improvement could be caused by the flash access behavior that contradicts the cache concept of data access locality.

To sum up, it is shown that DAM reacts to flash data access. Whether the flash data actually has to be loaded from flash or whether the data access can be fulfilled by the cache is not differentiable by the flash data access counter.

Why do some tasks have a high count of saved cycles and show no flash usage?

Tasks that have low flash access compared to other tasks and save a big proportion of cycles at lower CPU frequencies are the tasks *beebs_ nbody*, *beebs_ sqrt* and *beebs_ nsichneu*, as seen in Figure 10.18. With disabled data flash cache (see Figure 10.20), these tasks also do not save additional cycles. On the other hand, Figure 10.16 shows no saved cycles at lower CPU frequencies with disabled Flash Wait State Adaption, which proves that the cycle savings must originate from flash usage.

As already described, the STM32L476RGT6 is equipped with the ART Accelerator, which also implements an instruction cache. Thereby, the hypothesis is formed that those tasks have such a high number of sequential instructions that the instruction cache is not able to reuse with the cache implementation [5, sec. 3.3.4 ART Accelerator].

Even though these instructions are loaded from flash, they are not tracked by the DAM, but probably by the IAM. As already described in Section 4.1.2, this matching type is not accessible with the STM32L476RGT6 MCU, because events generated by the comparators for IAM are sent to the ETM. Unfortunately, the ETM only outputs the trace data over the trace port of the TPIU, but the parallel trace port is not routed to a pin of the MCU package.

To support the hypothesis, a high number of sequential loaded instructions could take place if the task has a big footprint. To expose the task footprint size of the BEEBS tasks, a small static analysis was performed with the source code being compiled to assembler code. The compilation has been achieved with the *arm-none-eabi-gcc* toolchain, which is also used by the RIOT compile process for the Nucleo-L476RG board [71].

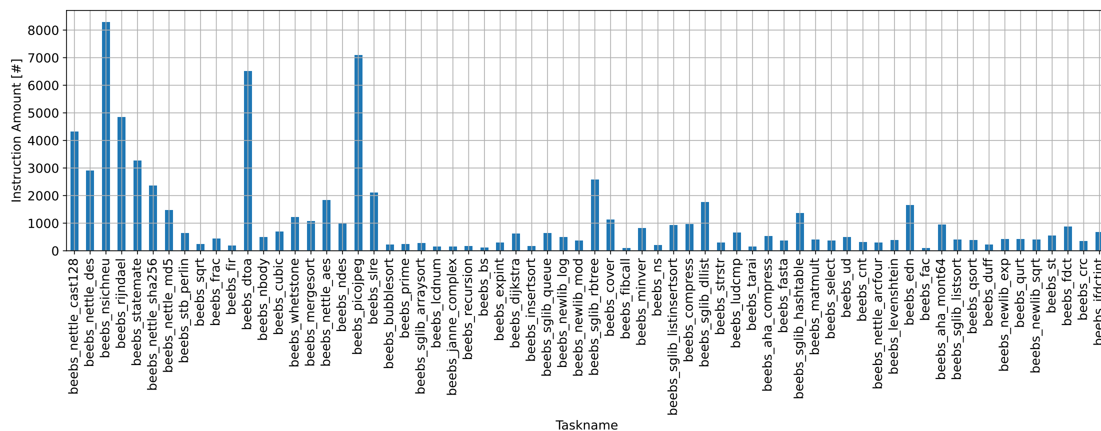


Figure 10.21: Amount of instructions per task assembler file. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

Figure 10.21 shows the result of simply counting all instructions in the assembler file per BEEBS task. It is appears, without only focusing on the named tasks, that the group of tasks that do save the most cycles due to Flash Wait State Adaption also seem to have the highest average number of instructions per task. But it has to be noted that the program flow of a task has an essential impact on how the instruction are loaded and how it can be accelerated by the flash cache.

Focusing on the named tasks, while *beebs_nsichneu* supports this hypothesis, the tasks *beebs_nbody* and *beebs_sqrt* on the other hand have a low count of instructions per assembler file.

By inspecting the C source code of both exceptional tasks, it is shown that they do have a low code footprint and both use floating point arithmetic in the form of float and double types. Therefore, the assumption is created that the usage of floating point arithmetic also has a huge impact on the savable cycles and further on the MEECF.

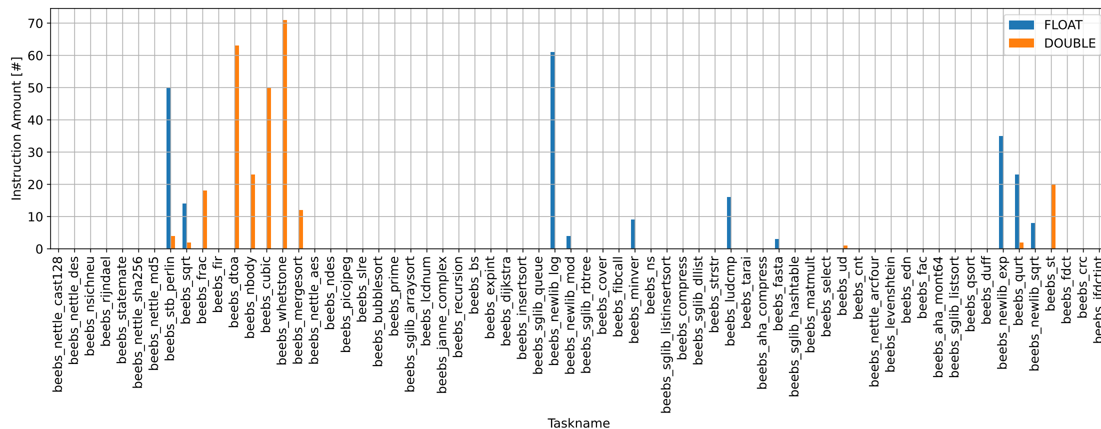


Figure 10.22: Amount of float and double operation instructions per task assembler file. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

Figure 10.22 is created on the same methodology as Figure 10.21, but counts the float and double operations instead. With static code analysis, it appears that *beebs_ nbody*, *beebs_ sqrt* and further tasks make use of floating point arithmetic. Many tasks using double floating point arithmetic reside in the task group that reduces the cycle count at lower CPU frequencies. Pallister et al. [20] also define floating point arithmetic as having a different energy consumption to integer arithmetic.

To be certain about the assumption that floating point arithmetic might save cycles and energy at lower CPU frequencies, a further measurement has been performed with the benchmark of Section 10.1. But this time with a float and double variable instead of an *uint32_ t* variable. It is shown that this benchmark with no memory access or RAM access also does not reduce the total cycles at lower CPU frequencies. Further, it is also not shown that the usage of float or double variables result in a lower MEECF setting, as seen in Figure 10.23.

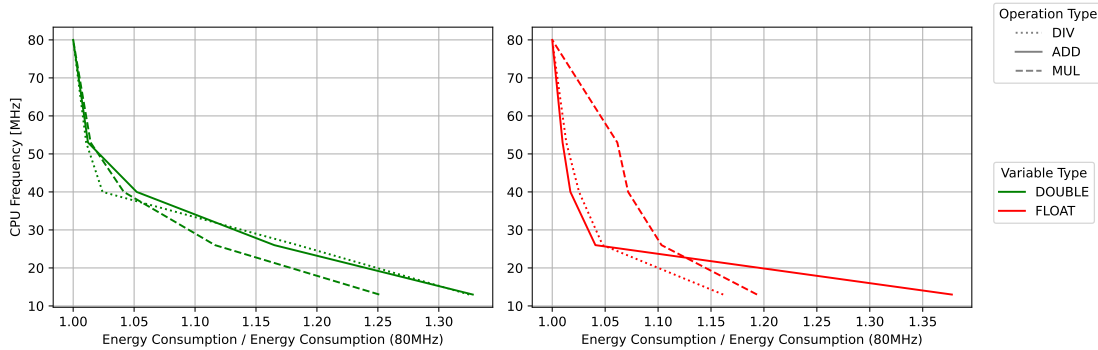


Figure 10.23: Energy consumption at different CPU frequencies in proportion to the energy consumption at 80MHz measured with the *REG* task of Section 10.1, but with float and double variables. The plot is grouped by the math operation and used variable types. The data was measured with enabled FWSA and *Fast Flash* DVS Policy.

Pre-Concluding Tracing with Comparators

Concluding this subsection, counting the flash data access to indicate tasks with cycle savings at lower CPU frequency is inaccurate due to the flash data cache. This is because the DAM also occurs if the flash access is fulfilled by the data flash cache. In addition, the address matching results are distorted because of the bandwidth bottleneck of the SWO output of the TPIU. The inaccuracy due to the flash cache could be solved with access to a flash cache miss counter, as in reference [38]. Unfortunately, nothing similar to this has been found on the STM32L476RGT6 MCU. The bandwidth bottleneck can be improved with the usage of the parallel trace port of the TPIU on other MCUs, as it provides a higher bandwidth.

Furthermore, the missing flash IAM when loading instructions from flash leaves potential cycle-saving tasks unrecognized. The inability to trace instruction flash access can be solved by using a MCU that allows the access to ETM tracing. Even though counting the flash instruction fetches will also encounter the limit of the flash instruction cache, it is shown that the cache can not always accelerate the flash access.

Lastly, by analyzing the assembler files at compile time, it is shown that many tasks of the leftmost cycle-saving task group make use of floating point arithmetic. A further measurement clarified that the float or double usage on its own does not save cycles and is not most energy efficiently performed at lower CPU frequencies.

Counter	High counter points to MEECF that is ...	Potential as indicator	Indicator limitation
COMP_DATA_RAM	None	None for the stand-alone counter	Bandwidth bottleneck due to the SWO
COMP_DATA_FLASH	Lower	Indicates cycle savings due to lowering FWS with high flash access	Flash access and flash cache access indistinguishable, bandwidth bottleneck with SWO output

Table 10.4: Findings for selecting a higher/lower MEECF with DAM and the BEEBS tasks.

10.4.5 Tracing with Profiling Counters

This subsection evaluates the DWT profiling counters, which enable to count cycles of different CPU activities. (see Table 4.1)

An experiment with *SLEEP* and *EX* tracing shows that the tasks of the BEEBS do not perform any sleeping or any exception processing due to interrupts or context switches. Profiling counter results that apply to the BEEBS tasks are visible in Figure 10.24.

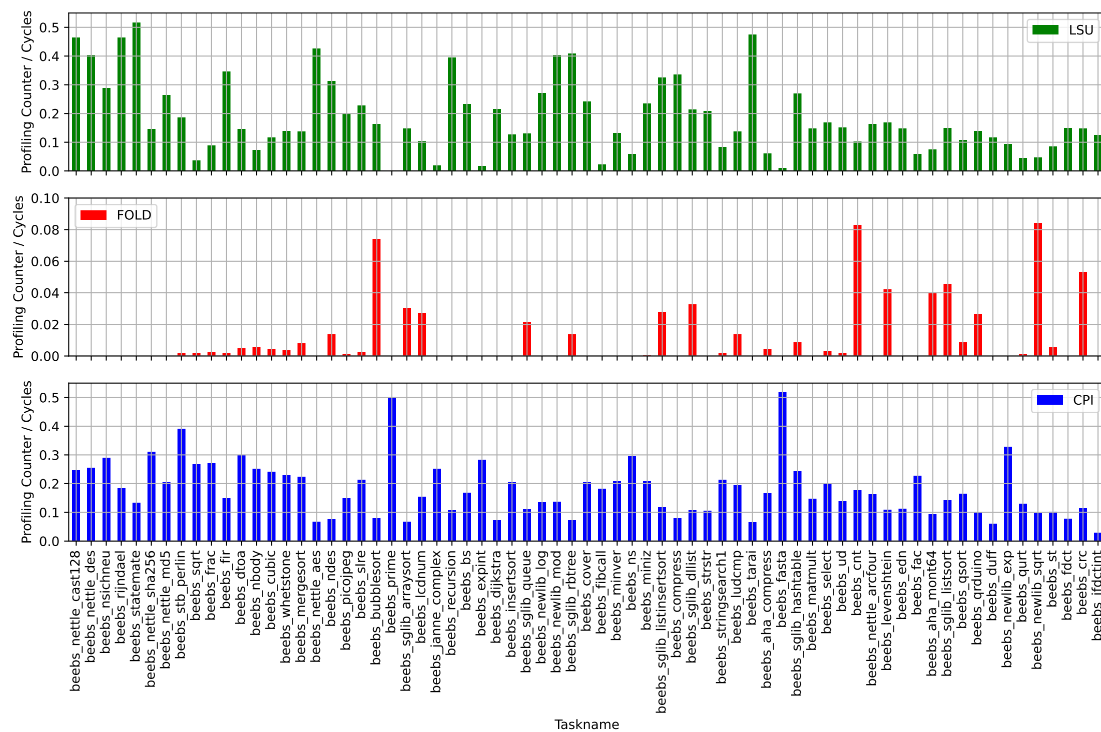


Figure 10.24: LSU, FOLD and CPI profiling counter in proportion to cycles. Measured at a CPU frequency of 80 MHz. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

Before going into detail about certain profiling counter results, the correlation to savable cycles (see Figure 10.15) per counter type is investigated.

Counter Type	**Saved Cycles Frequency Group**	**Linear Correlation (Pearson)**
FOLD @ 80MHZ	13MHz	0.141
LSU @ 80MHZ	40 MHz	0.223
CPI @ 80MHZ	53MHz	-0.345

Table 10.5: Highest linear correlation between profiling counter results of Figure 10.24 and the saved cycles of Figure 10.15.

Table 10.5 shows that the correlations between saved cycles at selected CPU frequencies and profiling counter tracing results are low. The higher negative linear correlation of -0.345 between the *CPI* counter and the saved cycles is probably caused by the higher

average normalized *CPI* values of the left group of tasks. The higher normalized *CPI* values might be caused by the usage of floating point arithmetic.

Inspecting Source Code of Selected Tasks

To get a better understanding of what certain profiling counter readings mean in terms of task semantics, in the following the C source code of a set of selected tasks is examined.

Fold: The *FOLD* counter counts on instructions that take 0 cycles. Further, the Technical Reference Manual of the Cortex M4 hints that "An IT instruction can be folded onto a preceding 16 Bit Thumb instruction, enabling execution in zero cycles [61, sec. 3.3.1]". An *IT* instruction is listed as an *if-then-else* state change. Therefore, with a high proportion of the *FOLD* counter to all cycles, many *IT* instructions have been saved. Regarding the task type, this could stand for a task that has many if-branches in loops with a small code footprint. Many if-branches could also be part of an algorithm that has a long decision path and a higher code footprint size.

The BEEBS tasks with the highest normalized *FOLD* counter of around 8% are *beebs_newlib_sqrt* (low data flash, high ram, MEECF at 80MHz), *beebs_cnt* (low data flash, high ram, MEECF at 80MHz) and *beebs_bubblesort* (low data flash, high ram, MEECF at 80MHz). To be not tricked by the scale of the normalized *FOLD* counter y-axis, in comparison to *LSU* or *CPI* the highest *FOLD* counter is still small.

The named three tasks have in common that they generally have a small code footprint (see Figure 10.21). They perform simple variable assignments and calculations that are executed dependent on short *if-structures*. The *if-structures* are further encapsulated by for or while loops.

To analyze a task that has a very low *FOLD* counter, but other similar properties, the task *beebs_stringsearch1* can be viewed at. It has a low flash access, a high ram access, a low *LSU* counter and a low to medium *CPI* counter. Investigating the source code, the task *beebs_stringsearch1* has a higher code footprint size and also many calculations inside *if-structures*, which are inside loops. A difference is visible regarding the simplicity of the if-conditions. While the tasks with a higher *FOLD* counter almost only check whether two given conditions are equal, bigger or smaller, *stringsearch1* often also performs increments or more complex calculations on the conditions inside the if-statements.

CPI: In Section 10.1 a higher *CPI* counter is observable with operations that need many cycles to be executed (*ADD* vs *DIV*). The question is whether a similar image can be drawn when analyzing this task property with more use-oriented BEEBS tasks.

Task that has a high *CPI* (50%), low *FOLD*, no *LSU*, very low *COMP_ DATA_ RAM* and low *COMP_ DATA_ FLASH* counter are *beebs_ prime* and *beebs_ fasta*. *Beebs_ prime* has a very small code base and performs a prime number check that involves many *MODULO* operations, which often use divide instructions. *Beebs_ fasta* also has a very small code base and performs many *ADD*, *MUL* and *DIV* operations.

A processing task that shows the opposite *CPI* counter height has not been traced. This is probably because many tasks do often perform many kinds of math operations. Furthermore, the *CPI* counter does not only count the mentioned expensive math operations, but all additional cycles required to perform instructions. This also includes branch instructions (*B <label>*) or *MOV PC* instructions (see reference [61, sec. 3.3.1]).

Regarding Figure 10.22, there might also be a correlation between a higher *CPI* counter and the usage of float/double operations. This originates from the observation that the tasks *beebs_ dtoa* to *beebs_ mergesort* and also *beebs_ newlib_ exp* seem to have a higher *CPI* counter of more than 20%, and also use many float/double assembler instructions. The instruction set of the Floating Point Unit (FPU) on the Cortex-M4 [61, sec. 7.2.3] shows that some instructions require more cycles than the integer ones. For example, *divide* takes 14 cycles, which is minimally 2 cycles longer than an integer divide with 2 to 12 cycles.

LSU: In Section 10.1 it is shown that the *LSU* counter correlates with the access to RAM and flash. This finding is strengthened with the detailed view of the *beebs_ prime* task. The task has a very low *LSU* counter, a very small footprint and further does not use any data structures that might be loaded from RAM or flash.

A BEEBS task that shows the opposite *LSU* counter height is the task *beebs_ tarai*. It has a very high *LSU* counter, a very small code footprint and does not use any data structures but a few literals. The high *LSU* counter is therefore caused by loading literals, but also by the fact that the task function is recursively calling itself three times with different inputs. The recursion demands a lot of RAM usage.

Why the FOLD, LSU or CPI counter alone can not be used to point to a lower MEECF?

As already seen by the correlation Table 10.5, none of the profiling counters by itself is an indicator of MEECF settings for all tasks without a huge inaccuracy. The question is, why is this the case and in which situations is there still potential?

FOLD: Whether *beebs_ newlib_ sqrt*, *beebs_ cnt* or *beebs_ bubblesort* perform most energy efficiently at the highest CPU frequency as a consequence of the high *Fold* counter, is questionable. The above-mentioned difference between similar tasks that only differ in the *FOLD* counter does not seem to be a sign on whether tasks can be executed more energy efficiently at any frequency.

CPI: A high *CPI* counter is found with tasks that calculate the majority of the execution time with operations that scale well with CPU frequency. Thereby, a correlation between high *CPI* and high CPU frequency could be drawn. This is true for two tasks of the rightmost group but fails with the task *beebs_ stp_ perlin* of the leftmost group. Here the differentiation might be possible by the combination with the *LSU* counter.

LSU: The *LSU* counter increments are connected to both RAM access, that scales with CPU frequency, and flash access, that does not always scale with frequency. Therefore, only the knowledge of the *LSU* counter can also not be correlated to any MEECF.

The full potential of selecting a MEECF by counting flash data access is prevented by the flash data cache. But the *LSU* counter might help in this situation. The *LSU* counter only counts on additional cycles to perform load/store operations. Thereby, it should increase less with flash data accesses that can be served by the flash data cache, than accesses that the cache can not fulfill. This should be true, as the cache is hardware-wise implemented as RAM [5, sec. 3.2], which takes fewer cycles to be accessed than flash. This *LSU* behavior has been shown in Figure 10.1, the same number of trace iterations with the same operations resulted in a higher *LSU* proportion for flash access than for RAM access. This statement is true under the assumption that the flash cache was bypassed. An experiment was further performed which traced the task of Figure 10.1 with enabled and disabled flash data cache, resulting in the same normalized *LSU* counter height. The flash data cache was also not used with enabled flash data cache, as the task implementation used a *volatile* variable.

With this knowledge about the *LSU* profiling counter, the removal of load/store cycles caused by RAM access could improve the *LSU* usage as a pointer to MEECFs settings (see Section 10.4.6).

Counter	High counter points to MEECF that is ...	Potential as indicator	Indicator limitation
CPI	Higher	Correlates to CPU-intensive operations	Many multi-cycle instructions visible with tasks that save energy at lower CPU frequencies
LSU	Lower	Higher with flash access, lower with flash cache acceleration	Counts RAM as well as flash accesses, but only flash accesses indicate optimization potential
FOLD	Higher	Higher MEECF due to tasks measured	Source code observation does not imply any MEECF

Table 10.6: Profiling counter findings with BEEBS tasks for selecting a higher/lower MEECF.

10.4.6 Counter Combinations to Improve the MEECF Selection

The profiling counter tracing results on their own do not have a high correlation to the savable cycles when lowering the CPU frequency and can not be used for selecting a MEECF setting without a high inaccuracy. Combinations might improve this correlation for all tasks or improve the ability to indicate a MEECF setting.

LSU without RAM

As *LSU* counts both RAM and flash access, the idea is to separate the RAM cycles from the *LSU* count to generate a task property that has a high correlation to savable cycles. Equation 10.1 shows the Normalized Combination Counter (NCC) formula used to calculate the data of Figure 10.25. The Figure is created with a Cycle(s) Per Access (CPA) of one. This means every RAM DTAOP stands for one additional cycle needed

to perform the access. In theory, the higher the *NCC* the more likely it is that cycles can be saved when lowering the CPU frequency.

$$NCC_{LSU-RAM} = \frac{\frac{LSU_{RAW}}{5/256} - \frac{RAM_{RAW}}{8.25} \cdot CPA}{CYCLES} \tag{10.1}$$

Comparing Figure 10.25 with the *LSU* counter plot of Figure 10.24 shows that this metric does not result in any better differentiation between tasks that save cycles and ones that do not.

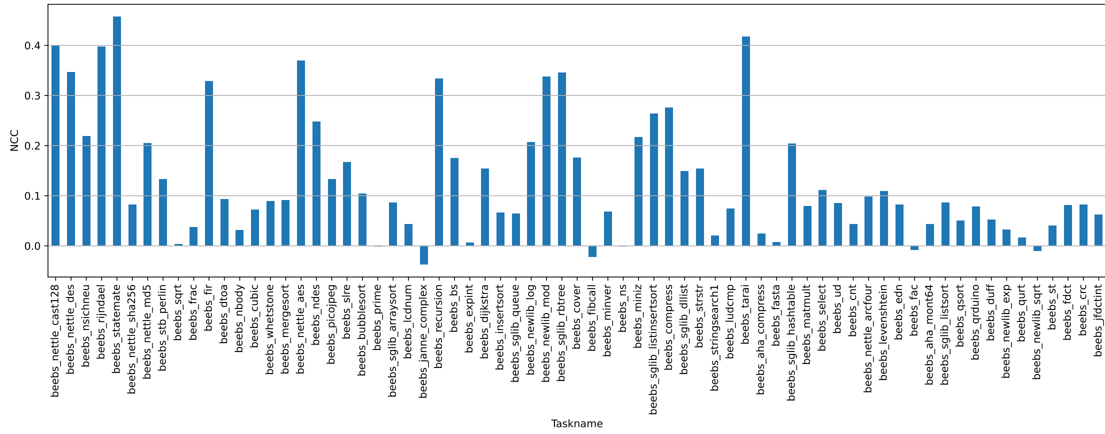


Figure 10.25: NCC of Equation 10.1 at a CPU Frequency of 80 MHz. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

Even a threshold selection that only looks at the *NCC* counter below a value of 0.1 seems to be arbitrary, as there are also tasks of the leftmost group of tasks that show a very low *LSU-RAM NCC* (< 0.05).

The many high counters (> 0.2) of the rightmost task group are caused by the inability of the RAM counter to differentiate more intensive RAM usage. Visible by Figure 10.18 is that many tasks have the same high normalized counter value of 0.03, which is caused by the bandwidth bottleneck of the SWO output. Thereby, this approach does not fail due to the concept but on the bottleneck, which can be improved by future work with the selection of a MCU that provides a trace output with a higher bandwidth.

High CPI and low LSU Looking at Figure 10.24, it could be argued that the combination of a normalized *CPI* counter value higher than 30% and a normalized *LSU* counter

being lower than 5% might indicate a higher MEECF. This only applies for two tasks of 69 traced (*beebs_prime* and *beebs_fasta*). A lower *CPI* height is not recommended being used for this combination, as it would also point to tasks of the leftmost group of tasks (see task *beebs_sqrt*). To conclude, this technique is not effective as it only selects a higher MEECF settings for two tasks.

Low CPI and low LSU If the majority of task instructions perform with a lower count of cycles indicated by a lower *CPI* and *LSU* count, it could be argued that these tasks are most energy efficiently performed at a higher CPU frequency.

Regarding a lower *LSU*, if the minority of instructions are multi-cycle load/store operations, lowering FWS should not save many cycles for task completion. A lower *CPI* means that the minority of instructions need multiple cycles to complete. These instructions should still scale well with CPU frequency, as seen in Section 10.1 by the add operation. In theory, tasks that spend fewer cycles for load/store instructions and other multi-cycle operations should execute more energy efficiently at a higher CPU frequency.

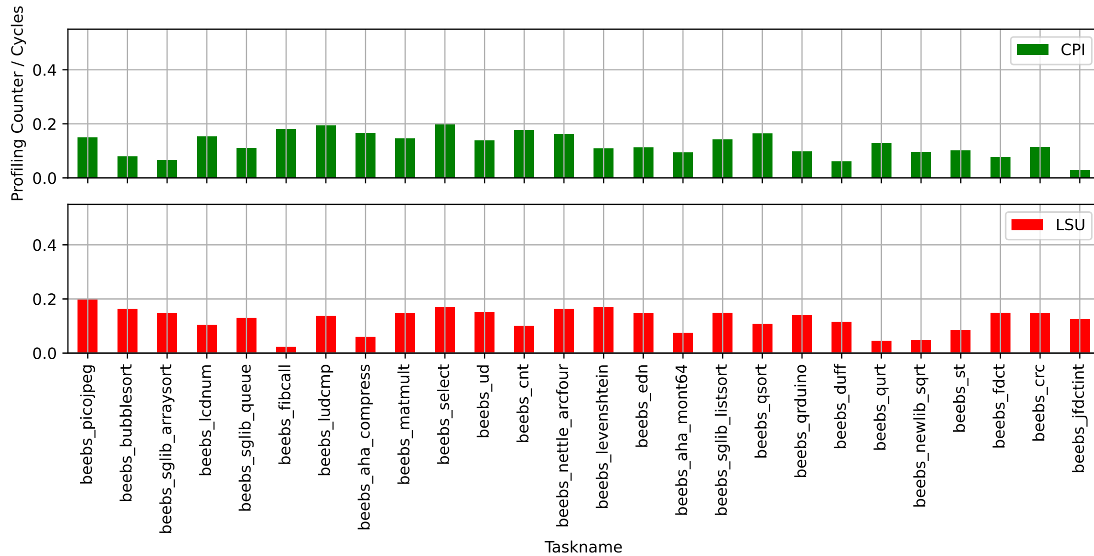


Figure 10.26: LSU and CPI counters in proportion to cycles for tasks with lower counters than 0.2. Measured at a CPU frequency of 80 MHz and The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

Figure 10.26 shows a threshold selection of tasks that both have a normalized *CPI* and *LSU* counter that is lower than 20%. It is possible to mainly select tasks that perform most energy efficiently at the highest CPU frequency (compare with Figure 10.12).

The only exceptional task is *beebs_picojpeg*, however in this particular case the energy reduction at a lower frequency is not significant.

Regarding the detection rate for a MEECF, this method uncovers 27% of all BEEBS tasks to scale well with CPU frequency.

Cycles Per Instruction A combination that is also presented by the ARM corporation [55] is shown in Section 4.1.1.

Figure 10.27 shows the proportion of traced cycles to the calculated instruction count to illustrate values that are independent on the actual trace length. A higher value means that more cycles are needed to perform an instruction. Regarding the profiling counters, a high *cycles per instruction* value is visible if both *CPI* and *LSU* make up a high proportion of all task cycles.

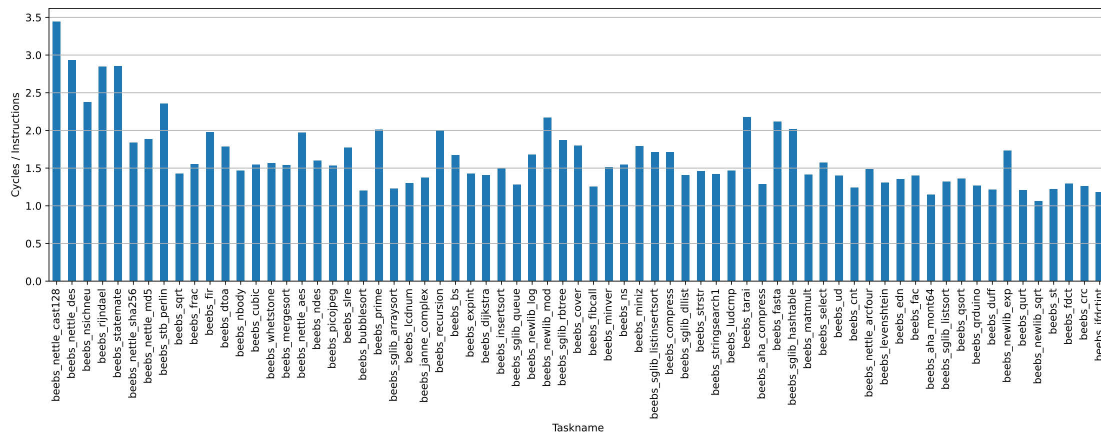


Figure 10.27: Cycles in proportion to the calculated instruction count (see Equation 4.1) at a CPU frequency of 80 MHz. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

It is clearly visible that a subgroup of the group of tasks that save cycles due to lowering the FWS at lower CPU frequencies (leftmost tasks) have the highest *cycles per instruction* values (compare with Figure 10.15). While it seems like most of the tasks that do not save cycles at lower CPU frequencies have a value that is lower than 1,5 *cycles per instructions*, some tasks also show a high value to up to 2,17 *cycles per instruction*.

Table 10.7 shows the linear correlation of the task property *cycles per instruction* of all BEEBS tasks in comparison to the saved cycles of Figure 10.15 at selected frequencies.

Cycles Per Instructions Frequency Group	**Saved Cycles Frequency Group**	**Task Grouping**	**linear Correlation (Pearson)**
80MHZ	13MHz	ALL	0.1166
80MHZ	13MHz	Task that save cycles at lower frequencies	0.4120
80MHZ	13MHz	Task that do not save cycles at lower frequencies	0.0928

Table 10.7: Highest correlations between the *cycles per instructions* task property at a CPU frequency of 80 MHz and the saved cycles of Figure 10.15, grouped by the task subgroups of Figure 10.12.

The correlation table shows that the *Pearson* correlation value of all tasks is unexpectedly small. This is probably caused by the high fluctuation of values in the group of tasks that do not reduce the total cycles at lower FWS.

By comparing the *cycles per instruction* values with the MEECF settings of Figure 10.12, it could be argued that the CPU frequency of 26 MHz can be configured if the calculated *cycles per instruction* property minimally reaches a value of 2,35. The highest CPU frequency should be selected below a value of 2,35. This threshold technique saves energy for 6 out of the 19 tasks that save energy at lower CPU frequencies. Due to the highest CPU frequency selection below the threshold, 13 tasks do not save energy. This technique selects the most energy efficient or a more energy efficient frequency setting for 78.7% ($\frac{69-13}{69}$) of all processing tasks. Thereby, the profiling counter values used to calculate the *cycles per instruction* have a high accuracy, as seen in Figure 10.14.

10.4.7 Cycle Tracings at Different CPU Frequencies

The prior sections focus on the task tracings at the highest CPU frequency, as this is usually the standard clock configuration of boards used by RIOT. Measuring an application at two different CPU frequencies enables to directly sense the saved cycles, to estimate

the savable cycles at CPU frequencies that have not been measured and to calculate the energy consumption dependent on the calculated cycles. As tracings at different CPU frequencies are invasive in terms of performance reduction, the constraint of measuring the cycles at only two different CPU frequencies is created to reduce the performance impact.

Savable Cycles The savable cycles can be estimated per task by first calculating how many cycles can be saved per FWS (SC_{FWS}) dependent on the two cycle measurements (80 MHz and a second measurement). Secondly, by applying SC_{FWS} to Equation 10.3 the number of cycles at a CPU frequency that has not been measured (CYC_{nm}) can be calculated.

$$SC_{FWS} = (CYC_{80MHz} - CYC_{other})/(FWS_{80MHz} - FWS_{other}) \tag{10.2}$$

$$CYC_{nm} = CYC_{80MHz} - (FWS_{80MHz} - FWS_{nm}) \cdot SC_{FWS} \tag{10.3}$$

Figure 10.28 shows the proportion of the calculated cycles per task (CYC_{nm}) in proportion to the measured number of cycles at 80 MHz. It is visible that the cycle estimation produces similar results to the measured cycle proportion of Figure 10.15. Nevertheless, the proportion of estimated cycles at 13 MHz differs from the measured cycle proportion of Figure 10.15. This is because the cycle estimations per task of Figure 10.28 are based on cycle measurements at the CPU frequencies of 80 and 53 MHz. The inaccuracy of this estimation is controllable by using a second cycle measurement at a lower CPU frequency, but this increases the performance impact of the system due to the lower CPU frequency.

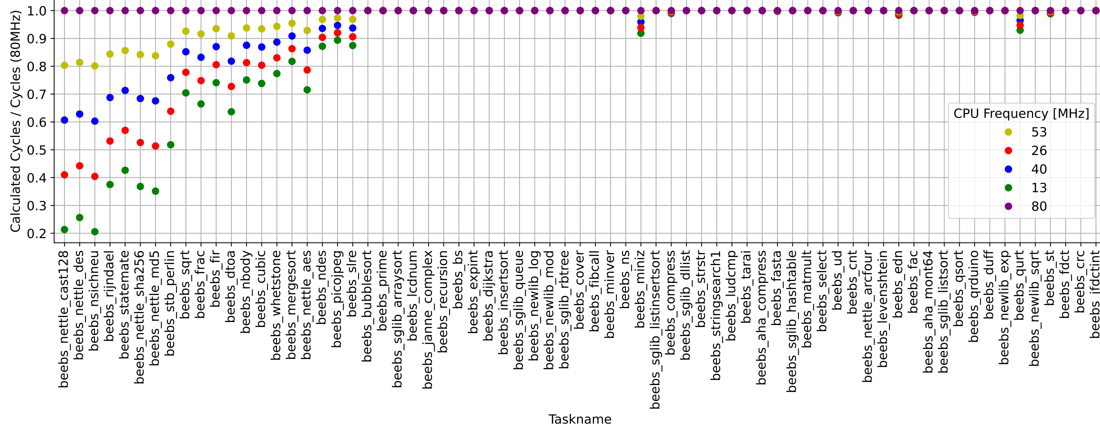


Figure 10.28: Calculated number of cycles per CPU frequency in proportion to the number of cycles at 80 MHz. The calculation is based on Equation 10.3 and uses cycle measurements at 80 and 53 MHz. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

From the perspective that cycle savings at lower CPU frequencies imply energy savings (see Section 10.4.3), measuring cycles at two different frequencies detects 25 of 69 tasks as being more energy efficiently performed at a lower CPU frequency. Under the assumption that the CPU frequency can be lowered to 40MHz if a task shows a cycle reduction, 17 out of 69 tasks save energy, whereas for 9 tasks the total energy is increased. Thereby, this technique selects the most energy efficient or a more energy efficient frequency setting for 87% of all processing tasks.

Estimated Energy Proportion With the estimated cycles at all frequencies the energy consumption per frequency can be calculated and used to detect a MEECF setting. The detection is possible with the following Estimated Energy Proportion (EEP), which is based on the Equation 2.6.

$$EEP = \frac{E_{total80mhz}}{E_{totalOther}} = \frac{\alpha_{80mhz} \cdot C \cdot V_{80mhz}^2 \cdot CYC_{80mhz} + P_{leak80mhz}(V) \cdot \frac{CYC_{80mhz}}{f_{80mhz}}}{\alpha_{other} \cdot C \cdot V_{other}^2 \cdot CYC_{other} + P_{leakOther}(V) \cdot \frac{CYC_{other}}{f_{other}}} \tag{10.4}$$

When tracing cycles at two different CPU frequencies and under the assumption that the switching activity α is not changing, all factors of Equation 10.4 can be measured at runtime, apart from the static power consumption $P_{stat}(V)$ and $\alpha \cdot C$.

Extrapolated Static Power Consumption The static power consumption is actually not dependent on the task behavior, but on the system configuration (see Section 2.1.3). This means, for a given system configuration it can be calculated offline. Therefore, the power consumption results at different CPU frequencies are extrapolated to a CPU frequency of 0 MHz. This extrapolation of the static power consumption is shown in Figure 10.29. The average power consumption per CPU frequency is derived from the power consumption of all tasks of Figure 10.11 with the DVS Policy *Fast Flash*. As the power consumption with the DVS Policy *Low Voltage* is only measured at a single CPU frequency, an extrapolation for the static power consumption is not possible and will not be considered.

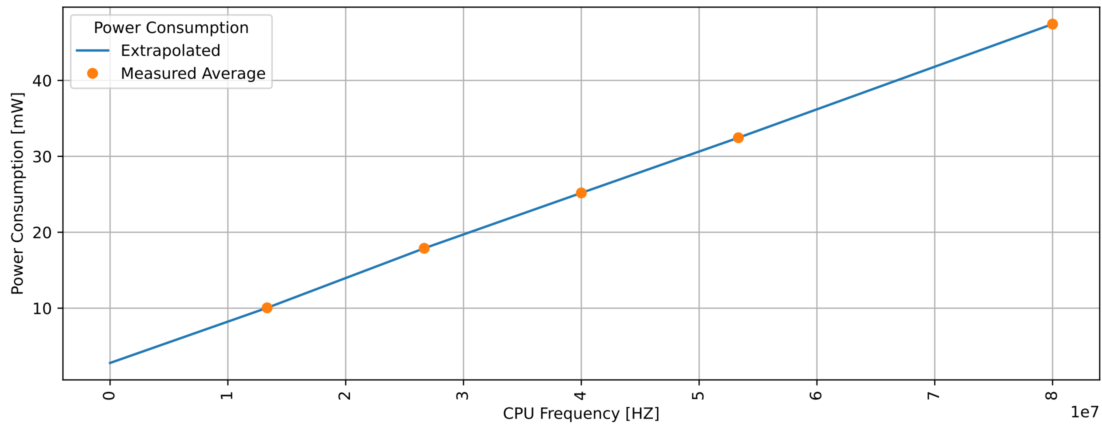


Figure 10.29: Average static power consumption per CPU frequency of all tasks of Figure 10.11 and extrapolated power consumption to a CPU frequency of 0 MHz.

The extrapolated static power consumption $P_{leakCalc}$ of the Nucleo-L476RG board results in 2.78207mW at a CPU voltage of 1.2V.

Average *alphaC* **($\alpha \cdot C$)** While the total load capacitance of a circuitry C is hardware dependent, the switching activity α is task dependent (see Section 2.1.4). The factor α is therefore expected to change for different tasks and further slightly at different CPU frequencies with tasks that save cycles at lower frequencies. To be able to estimate the energy consumption proportion of Equation 10.4, an average *alphaC* for all tasks and CPU frequencies is assumed. Therefore, *alphaC* first has to be calculated offline for all

tasks at different CPU frequencies with Equation 10.5, which is derived from Equation 2.6.

$$\alpha \cdot C = \frac{E_{totalMeasured}(f) - P_{leakCalc} \cdot \frac{CYC(f)}{f}}{V^2 \cdot CYC(f)} \tag{10.5}$$

Figure 10.30 shows the task and frequency dependent *alphaC* values for all BEEBS tasks. Based on this data an average *alphaC* value between all tasks and all CPU frequencies is calculated and results in a factor of $3.876473 \cdot 10^{-10}$.

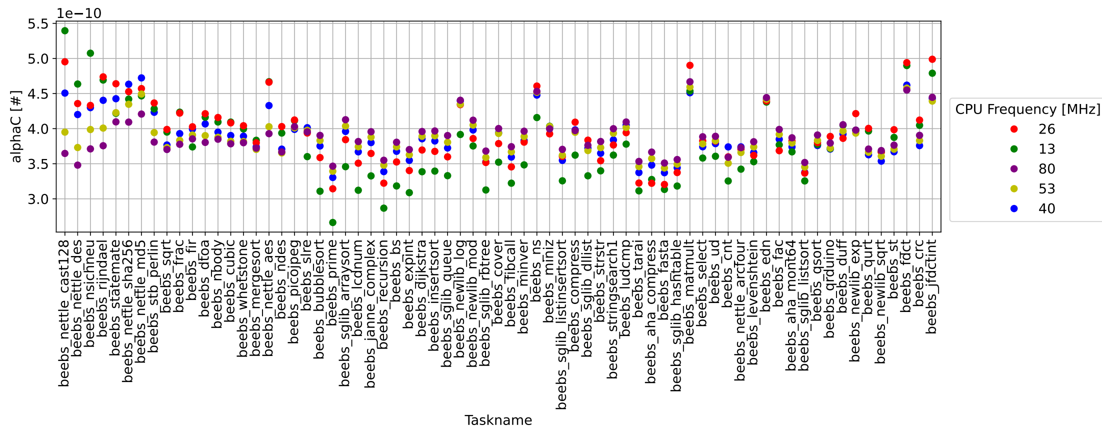


Figure 10.30: Calculated $\alpha \cdot C$ factor at different CPU frequencies with Equation 10.5. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

Results of Estimated Energy Proportion With the extrapolated static power consumption of the STM32L476RGT6 MCU and average *alphaC* factor, Equation 10.4 can be used to detect a MEECF setting per task. Figure 10.31 shows the calculated *EEP* per task at different CPU frequencies. Thereby, the cycles only have been measured for the CPU frequencies of 80 and 13 MHz.

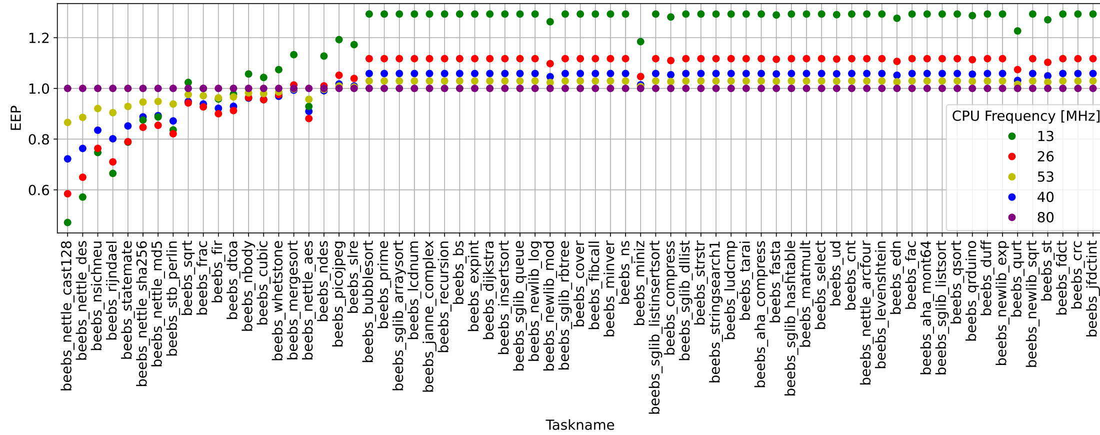


Figure 10.31: Estimated Energy Consumption as shown in Equation 10.4 with cycle measurements of 80 and 13 MHz, grouped by CPU frequency. The tasks are sorted by the *task order of energy saving potential* as seen in Figure 10.12.

Comparing Figure 10.31 with the estimated cycles per task of Figure 10.28 shows that tasks like *beebs_ miniz* are correctly selected as being most energy efficiently performed at the highest CPU frequency, despite their savable cycles at lower CPU frequencies. This behavior is probably caused by the static energy which is increased at lower CPU frequencies.

Comparing Figure 10.31 with the measured energy proportion per task of Figure 10.12 shows that all tasks that perform most energy efficiently at the highest CPU frequency are correctly identified by the energy estimation. Many tasks that save energy at lower CPU frequencies are correctly identified as being most energy efficient at the lowest or second-lowest CPU frequency. Regarding the amount of energy savings, the estimated energy savings differ from the measured energy savings by up to 10%, as seen by task *beebs_ nettle_ cast128*.

Assuming the estimated most energy efficient CPU frequency of Figure 10.31 will be selected for all processing tasks. Thereby, this technique selects the most energy efficient or a more energy efficient frequency setting for 95.6% (66 of 69 tasks) of all processing tasks. The tasks *beebs_ nbody*, *beebs_ cubic* and *beebs_ nettle_ aes* only slightly increase energy consumption by selecting the CPU frequencies of 26 MHz.

10.5 Tracing Overhead for Selected Task Properties in Practice

Assuming we have a deployed system and want to trace certain application parts with the task characterization model to assess task properties and increase the energy efficiency. In this section the overhead in terms of number of tracings will be highlighted and optimization possibilities are discussed. The task properties of *cycles per instruction* and *cycles saved* are examined as they can be used to select MEECF settings. Further, the *flash access* traced with comparators is examined as it has the potential of being valuable for the selection of a MEECF setting.

10.5.1 Cycles Per Instruction

Assuming the *cycles per instruction* property is selected as it does not need to be traced at different CPU frequencies. To obtain this task property with the calculation shown in Section 4.1.1 and considering the limitation of the implemented task characterization model (see Table 6.2) a total of 5 tracings have to be performed for each application part.

As shown in Figure 10.24, the *CPI* and *LSU* counter have the biggest impact on the instruction calculation. It could be argued that only these both counters are required, therefore the necessary number of tracings could be reduced to 2.

The profiling counters are designed to be able to generate overflows simultaneously and the packet encoding of an ECP can hold multiple profiling counter flags in one packet (see Figure 4.1). The feedback mechanism does not deserialize the trace packets and can not distinguish the packet flags of an ECP. Therefore, the separate tracings of the profiling counters are necessary to trace the individual profiling counter information (see Table 6.2). As the *CPI* and *LSU* counter are added for the *cycles per instruction* metric, the hypothesis is formed that a simultaneous overflow generation might be possible for this special case. This would reduce the necessary number of tracings to a single one. Analyzing this optimization has not been the scope of this thesis, but future work has to investigate whether this optimization offers the necessary information content.

10.5.2 Cycles Saved

Assuming it has been decided to use the cycle saving metric of Section 10.4.7 alone, as this resulted in the most confident tracing results. Even though tracing the 32 Bit cycle counter does not require the timer counter feed mechanism, the cycles saving metric is invasive as it requires to trace the application parts at different CPU frequencies. When using the two highest CPU frequencies this impact is considered to be small, but it requires to trace the same part of a program twice.

To reduce the negative impact of tracing cycles at lower CPU frequencies in situations when the embedded device performs important application load, it could be argued to perform an initial tracing of multiple different application parts and frequencies. This is shown by Rottleuthner et al. [10] with their PU-assessment.

10.5.3 Flash Access

The benefit of counting the flash access to detect tasks that save cycles at lower CPU frequency is limited (see Section 10.4.4). Many DTAOP packets are lost due to the high packet generation of the comparators in combination with the bandwidth bottleneck of the SWO connection. Furthermore, the comparators track both the flash access that results in an actual load from flash memory and the flash access that can be served by the flash data cache. Nevertheless, the ability to detect cycle saving tasks via flash access counting can be improved. MCUs that support the parallel trace port of the TPIU can solve the bandwidth bottleneck and additionally enable the access to instruction access tracing with an ETM. A mechanism that counts the flash cache misses or gives insight on the flash access pattern can improve the differentiation of actual flash loads and flash cache acceleration. Furthermore, some MCUs might not have a flash cache.

Still, as shown in Section 6.3.1, to fully trace the flash access a total of 8 traces per flash access type (data, instruction) are needed for the 1 MByte of flash size on the STM32L476RGT6 MCU. However, the amount of memory space that has to be observed can be adjusted to the application firmware footprint size. The firmware size is known statically and can therefore potentially lower the 8 required traces. Furthermore, on MCUs with similar flash size, the number of comparators and the observable range per comparator is possibly bigger.

11 Conclusion and Outlook

11.1 Achievements

This thesis proves that debug components of the Cortex-M4 CPU can be instrumented to obtain task specific properties at runtime without the need to use dedicated hardware debug components. The obtained task properties are useful for indicating whether tasks save energy by lowering the CPU frequency and whether tasks already execute at the most energy efficient CPU frequency.

The instrumented debug and trace components of the Cortex-M4 processor are available on three more Cortex-M processors [3]. Consequently, the trace components are potentially usable on 49.23% of all RIOT boards. The designed feedback mechanism of counting trace packets with a timer counter produces a very low hardware overhead of only two resistors and three cables. Furthermore, timers with a counter width of 32 Bit that can be driven by one external clock source are available on most RIOT boards [58].

Selecting the processing tasks of the BEEBS is a reasonable choice to evaluate the traced task properties with. The benchmark suite offers 69 different tasks, of which 19 tasks reduce energy consumption at lower CPU frequencies and 50 tasks perform most energy efficient at the highest CPU frequency.

The task characterization model detects whether tasks have a high flash access, a low RAM access or utilize peripherals like the SPI unit without the need to inspect or manipulate the source code. By tracing the profiling counters, it is possible to detect whether the CPU is sleeping, is performing exception processing or is executing load/store instructions. Furthermore, it is possible to detect how many instructions have been performed in a certain amount of cycles.

Using this kind of information about tasks together with DVFS energy can be saved by lowering the CPU frequency and voltage. When a task sleeps most of its execution cycles or a high SPI peripheral access is traced, the lowest CPU frequency shows the lowest energy consumption. Other traced task properties do not show this simple correlation.

It has been discovered that some tasks save cycles at lower CPU frequencies due to lowering the FWS. Cycle saving tasks are highly correlated with tasks that also save energy when lowering the CPU frequency while increasing the execution time less than the factor of the CPU frequency reduction.

A shown threshold technique that uses the calculated cycles per instruction property selects the most energy efficient or a more energy efficient frequency setting for 78.7% of all processing tasks. Thereby, it is traceable without changing the CPU frequency. Tracing the cycles of a task at minimally two different CPU frequencies allows to select the most energy efficient or a more energy efficient frequency setting for 87% of all processing tasks. The required tracing at multiple CPU frequencies comes with the inconvenience of invasively changing the system performance. With additional information of the offline measured device static power consumption and *alphaC* factor, the traced cycles can be used to estimate energy consumption. This selects a more energy efficient setting for 95.6% of all processing tasks.

The base for the evaluation are the BEEBS tasks, where one task uses up to 35% less energy with 30% longer execution time by reducing the CPU frequency. On the other hand, if the frequency setting is misconfigured the consequence is an increase in execution time at a factor of 6 and an increased energy consumption of up to 30%. These energy values do not incorporate the energy overhead due to changing the frequency/voltage setting or the energy overhead of tracing task properties with the implemented task characterization model. Tracing the profiling counters, which are used to calculate the cycles per instruction property, increases the power consumption of BEEBS tasks by 6.32% to 8.01% dependent on the task activity and configured profiling counter. But it has to be noted that tracing does not always need to be active.

11.2 Problems

Some tasks save cycles due to FWS reduction at lower CPU frequencies. This cycle reduction implies energy savings. Therefore, a correlation with the flash access has been

examined. The flash data access can be traced by comparators that generate packets when matching an accessed flash memory address. Unfortunately, many generated packets are lost due to the high packet generation of the comparators in combination with the bandwidth bottleneck of the TPIU output connection. Furthermore, the comparators track both the flash access that results in an actual load from flash memory and the flash access that can be served by the flash data cache. This lack of access differentiation results in tasks that show a high flash access but do not save cycles at lower CPU frequencies. The bandwidth bottleneck is a problem of the slow SWO connection which is the only available connection on the MCU package of the STM32L476RGT6. MCUs that support the parallel trace port with a higher bandwidth should be able to reduce this bottleneck. The lack of differentiation for flash access is only a problem for MCUs that implement a flash cache. Additional information on cache misses or the flash access pattern can improve this differentiation for MCUs with flash caches.

Some tasks also save many cycles at lower CPU frequencies when the instruction flash cache can not accelerate the flash instruction load. Again, due to the available SWO connection on the used STM32L476RGT6, this type of memory access has not been able to trace. MCUs that support the parallel trace port and the optional ETM module should also be able to trace instruction flash loads.

The *LSU* profiling counter increases if additional cycles are needed for load/store instructions. Therefore, it tracks flash data access that takes multiple cycles to perform. The approach to use the *LSU* profiling counter to indicate cycle savings at lower CPU frequencies fails. The counter also counts cycles for RAM access, but RAM access does not correlate to cycle reduction with lower FWS. The removal of RAM cycles from the measured *LSU* counter with the RAM comparator counter was not possible with high accuracy. The low accuracy is again caused by the bandwidth bottleneck of the SWO. The accuracy will be improved with a higher bandwidth trace port.

11.3 Future Work

Many problems deal with the high generation of packets when using comparators in combination with the bandwidth bottleneck of the available SWO connection on the used STM32L476RGT6 MCU. Another board with a MCU that enables tracing over the higher-bandwidth parallel trace port should be examined. Consequently, tracing the flash data access for selecting tasks that save cycles at lower CPU frequencies and the approach

to remove the RAM cycles from the *LSU* counter should be re-evaluated. Therefore, it would make sense to investigate a MCU that also implements the ETM module. This also enables to trace instruction access with the comparators and potentially provides trace filtering features. A reasonable starting point might be to use the SLSTK3402A board, which provides access to the parallel trace port and the ETM module [54].

While this thesis is focused on tracing tasks at a granularity of 1s, further research should examine the impact on the measurements with tracings at a finer granularity. Furthermore, the benchmarked tasks have been measured separately on a single thread. Future work has to examine at which accuracy task properties can be attributed to tasks or threads in dynamic scenarios with multiple threads, multiple tasks and different task lengths.

Lastly, there may even be more potential in the data that is shown, which has not been discovered yet. A future approach could be to evaluate the data with a machine learning approach or other techniques that search for additional correlation between traced data, saved cycles and measured energy consumption.

Bibliography

[1] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective.* Prentice-Hall, Inc., Jan 1996.

[2] J. Butts and G. Sohi, "A static power model for architects," in *33rd annual Symp. on Microarchitecture (MICRO'00).* Monterey, CA, USA: IEEE, 2000, pp. 191–201.

[3] J. Yiu, "ARM Cortex-M for Beginners - An overview of the ARM Cortex-M processor family and comparison," ARM, Tech. Rep., March 2017. [Online]. Available: https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2057-00-00-00-01-28-35/Cortex_2D00_M-for-Beginners-_2D00_-2017_5F00_EN_5F00_v2.pdf

[4] *ARMv7-M Architecture Reference Manual*, ARM, February 2021, rev. E.e. [Online]. Available: https://developer.arm.com/documentation/ddi0403/latest

[5] *RM0351 Reference manual - STM32L47xxx, STM32L48xxx, STM32L49xxx and STM32L4Axxx advanced Arm®-based 32-bit MCUs*, ST, June 2021, RM0351 Rev 9. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0351-stm32l47xxx-stm32l48xxx-stm32l49xxx-and-stm32l4axxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

[6] *User manual (UM1724) - STM32 Nucleo-64 boards (MB1136)*, ST, August 2020, Rev 14. [Online]. Available: https://www.st.com/resource/en/data_brief/nucleo-l476rg.pdf

[7] *Model DMM7510 7-1/2 Digit Graphical Sampling Multimeter User's Manual*, Keithley, September 2019, Rev. C. [Online]. Available: https://www.tek.com/en/support/product-support?series=DMM7510%20&type=manual

[8] *ARMv8-M Architecture Reference Manual*, ARM, September 2010, Version ID28092022. [Online]. Available: https://developer.arm.com/documentation/ddi0553/latest

[9] M. Rottleuthner, T. C. Schmidt, and M. Wählisch, "Sense Your Power: The ECO Approach to Energy Awareness for IoT Devices," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 3, pp. 24:1–24:25, March 2021. [Online]. Available: https://doi.org/10.1145/3441643

[10] M. Rottleuthner, T. C. Schmidt, and M. Wählisch, "Dynamic Clock Reconfiguration for the Constrained IoT and its Application to Energy-efficient Networking," in *International Conference on Embedded Wireless Systems and Networks (EWSN'22)*. New York, USA: ACM, October 2022.

[11] S. Mittal, "A Survey of Techniques For Improving Energy Efficiency in Embedded Computing Systems," *International Journal of Computer Aided Engineering and Technology*, January 2013.

[12] S. Ahmed, Q. ul Ain, J. H. Siddiqui, L. Mottola, and M. H. Alizai, "Intermittent Computing with Dynamic Voltage and Frequency Scaling," in *Proc. of the 2020 International Conference on Embedded Wireless Systems and Networks*, ser. EWSN '20. USA: Junction Publishing, February 2020, pp. 97–107.

[13] T. Pering, T. Burd, and R. Brodersen, "Voltage Scheduling in the IpARM Microprocessor System," in *Symp. on Low Power Electronics and Design (ISLPED'00)*. ACM, February 2000, pp. 96 – 101.

[14] D. Grunwald, P. Levis, K. I. Farkas, C. B. M. III, and M. Neufeld, "Policies for Dynamic Clock Scheduling," in *4th Symp. on Operating Systems Design and Implementation (OSDI'00)*. San Diego, CA, USA: USENIX Association, 2000.

[15] D. Molka, R. Schöne, D. Hackenberg, and W. E. Nagel, "Detecting Memory-Boundedness with Hardware Performance Counters," in *8th ACM/SPEC International Conference on Performance Engineering (ICPE'17)*. New York, NY, USA: ACM, 2017, pp. 27–38.

[16] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, December 2018. [Online]. Available: http://dx.doi.org/10.1109/JIOT.2018.2815038

[17] "A ScaleClock Implementation for RIOT," retrieved 2022-10-21. [Online]. Available: https://github.com/inetrg/RIOT/tree/ScaleClock

[18] *Data brief - NUCLEO-XXXXCX NUCLEO-XXXXRX NUCLEO-XXXXRX-P*, ST, December 2021, DB2196 - Rev 15 . [Online]. Available: https://www.st.com/resource/en/data_brief/nucleo-l476rg.pdf

[19] *UG257: EFM32 Pearl Gecko PG12 Starter Kit User's Guide*, Silicon Labs, January 2017, Rev. 1.0. [Online]. Available: https://www.silabs.com/documents/public/user-guides/ug257-stk3402-usersguide.pdf

[20] J. Pallister, S. Hollis, and J. Bennett, "BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms," Open Archive: arXiv.org, Technical Report arXiv:1308.5174v2, September 2013. [Online]. Available: https://arxiv.org/abs/1308.5174

[21] L. Niu and G. Quan, "Reducing Both Dynamic and Leakage Energy Consumption for Hard Real-Time Systems," in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)*. New York, NY, USA: ACM, 2004, pp. 140–148.

[22] C.-H. Hsu, U. Kremer, and M. Hsiao, "Compiler-Directed Dynamic Voltage/Frequency Scheduling for Energy Reduction in Microprocessors," in *International Symposium on Low Power Electronics and Design (ISLPED'01)*. IEEE, 2001, pp. 275–278.

[23] C.-H. Hsu and W. chun Feng, "Effective Dynamic Voltage Scaling through CPU-Boundedness Detection," in *International Workshop on Power-Aware Computer Systems (PACS'04)*. Springer, 2004, pp. 135–149.

[24] S. Eyerman and L. Eeckhout, "Fine-Grained DVFS Using on-Chip Regulators," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 1, February 2011.

[25] A. Varma, B. Ganesh, M. Sen, S. R. Choudhury, L. Srinivasan, and B. Jacob, "A Control-Theoretic Approach to Dynamic Voltage Scheduling," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'03)*. New York, NY, USA: ACM, 2003, pp. 255–266.

[26] G. Dhiman, K. K. Pusukuri, and T. Rosing, "Analysis of dynamic voltage scaling for system level energy management," *USENIX HotPower*, vol. 8, 2008. [Online]. Available: https://www.usenix.org/legacy/event/hotpower08/tech/full_papers/dhiman/dhiman_html/

[27] A. Shahid, S. Arif, M. Qadri, and S. Munawar, *Power Optimization Using Clock Gating and Power Gating: A Review.* IGI Global, 07 2016, pp. 1–20.

[28] E. Calore, A. Gabbana, S. F. Schifano, and R. Tripiccione, "Evaluation of DVFS techniques on modern HPC processors and accelerators for energy-aware applications," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 12, p. e4143, Mar 2017.

[29] I. Manousakis, M. Marazakis, and A. Bilas, "FDIO: A Feedback Driven Controller for Minimizing Energy in I/O-Intensive Applications," in *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'13).* San Jose, CA: USENIX Association, 2013.

[30] P. Pillai and K. G. Shin, "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems," in *18th Symp. on Operating Systems Principles (SOSP'01).* New York, NY, USA: ACM, 2001, pp. 89–102.

[31] A. Yeganeh-Khaksar, M. Ansari, S. Safari, S. Yari-Karin, and A. Ejlali, "Ring-DVFS: Reliability-Aware Reinforcement Learning-Based DVFS for Real-Time Embedded Systems," *IEEE Embedded Systems Letters*, vol. 13, no. 3, pp. 146–149, 2021.

[32] R. Medina and L. Cucu-Grosjean, "Work-in-Progress: Probabilistic System-Wide DVFS for Real-Time Embedded Systems," in *IEEE Symp. on Real-Time Systems Symposium (RTSS'19).* IEEE, 2019, pp. 508–511.

[33] T. Tatematsu, H. Takase, G. Zeng, H. Tomiyama, and H. Takada, "Checkpoint Extraction Using Execution Traces for Intra-task DVFS in Embedded Systems," in *6th IEEE International Symposium on Electronic Design, Test and Application (DELTA'11).* IEEE, 2011, pp. 19–24.

[34] J. Pouwelse, K. Langendoen, and H. Sips, "Dynamic Voltage Scaling on a Low-Power Microprocessor," in *7th annual International Conference on Mobile Computing and Networking (MobiCom'01).* New York, NY, USA: ACM, 2001, pp. 251–259.

[35] R. Antonio, R. Costa, A. Ison, W. Lim, R. Pajado, D. Roque, R. Yutuc, C. Densing, M. T. de Leon, M. Rosales, and L. Alarcon, "Implementation of Dynamic Voltage Frequency Scaling on a Processor for Wireless Sensing Applications," in *TENCON 2017 - 2017 IEEE Region 10 Conference.* Piscataway, NJ, USA: IEEE, November 2017, pp. 2955–2960.

[36] P. Mantovani, E. G. Cota, K. Tien, C. Pilato, G. D. Guglielmo, K. Shepard, and L. P. Carlon, "An FPGA-based Infrastructure for Fine-Grained DVFS Analysis in High-Performance Embedded Systems," in *53nd ACM/EDAC/IEEE Design Automation Conference (DAC'16)*. IEEE, 2016, pp. 1–6.

[37] K. K. Rangan, G.-Y. Wei, and D. Brooks, "Thread Motion: Fine-Grained Power Management for Multi-Core Systems," in *36th International Symp. on Computer Architecture (ISCA'09)*. New York, NY, USA: ACM, 2009, pp. 302–313.

[38] C. Poellabauer, L. Singleton, and K. Schwan, "Feedback-Based Dynamic Voltage and Frequency Scaling for Memory-Bound Real-Time Applications," in *11th IEEE Symp. on Real Time and Embedded Technology and Applications*. IEEE, 2005, pp. 234–243.

[39] S. Saewong and R. Rajkumar, "Practical Voltage-Scaling for Fixed-Priority RT-Systems," in *9th IEEE Symp. on Real-Time and Embedded Technology and Applications*. IEEE, 2003, pp. 106–114.

[40] J. Khan, S. Bilavarn, and C. Belleudy, "Energy Analysis of a DVFS based power strategy on ARM platforms," in *IEEE Faible Tension Faible Consommation (FTFC'12)*. IEEE, 2012, pp. 1–4.

[41] S. Liu, Q. Qiu, and Q. Wu, "Energy Aware Dynamic Voltage and Frequency Selection for Real-Time Systems with Energy Harvesting," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '08. New York, NY, USA: ACM, March 2008, pp. 236–241.

[42] T. Wu, Y. Liu, D. Zhang, J. Li, X. S. Hu, C. J. Xue, and H. Yang, "DVFS-Based Long-Term Task Scheduling for Dual-Channel Solar-Powered Sensor Nodes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 11, pp. 2981–2994, 2017.

[43] C. Zhuo, S. Luo, H. Gan, J. Hu, and Z. Shi, "Noise-Aware DVFS for Efficient Transitions on Battery-Powered IoT Devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 7, pp. 1498–1510, 2020.

[44] L. Songfan, L. Lu, M. Hussain, Y. Ye, and H. Zhu, "Sentinel: Breaking the Bottleneck of Energy Utilization Efficiency in RF-Powered Devices," *IEEE Internet of Things Journal*, vol. 6, pp. 705–717, 02 2019.

[45] E. Rotem, A. Mendelson, A. Naveh, and M. Moffie, “Analysis of The Enhanced Intel® Speedstep® Technology of the Pentium® M Processor,” in *First Workshop on Temperature-Aware Computer Systems (TACS'04)*. University of Virginia, 2004.

[46] C.-H. Hsu and U. Kremer, “The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction,” in *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'03)*. New York, NY, USA: ACM, 2003, pp. 38–48.

[47] K. Govil, E. Chan, and H. Wasserman, “Comparing Algorithm for Dynamic Speed-Setting of a Low-Power CPU,” in *1st annual International Conference on Mobile Computing and Networking (MobiCom'95)*. New York, NY, USA: ACM, 1995, pp. 13–25.

[48] M. Weiser, B. Welch, A. Demers, and S. Shenker, “Scheduling for Reduced CPU energy,” in *Mobile Computing*, T. Imielinski and H. F. Korth, Eds. Springer, 1996, pp. 449–471.

[49] T. Pering, T. Burd, and R. Brodersen, “The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms,” in *International Symposium on Low Power Electronics and Design (ISLPED'98)*. IEEE, 1998, pp. 76–81.

[50] R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt, “Predicting Performance Impact of DVFS for Realistic Memory Systems,” in *45th annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE, 2012, pp. 155–165.

[51] K. Choi, R. Soma, and M. Pedram, “Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 1, pp. 18–28, 2005.

[52] A. Weissel and F. Bellosa, “Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management,” in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'02)*. New York, NY, USA: ACM, 2002, pp. 238–246.

[53] “KEIL MDK - Microcontroller Development Kit,” retrieved 2022-10-17. [Online]. Available: https://www.keil.com/

[54] *EFM32PG12 Wireless Gecko Family Reference Manual*, Silicon Labs, February 2022, Rev. 1.0. [Online]. Available: https://www.silabs.com/documents/public/reference-manuals/efm32pg12-rm.pdf

[55] "Using DWT and other methods to count executed instructions on Cortex-M." retrieved 2022-10-17. [Online]. Available: https://developer.arm.com/documentation/ka001499/1-0

[56] *Embedded Trace Macrocell Architecture Specification - ETMv1.0 to ETMv3.5*, ARM, September 2011, Rev. Q. [Online]. Available: https://developer.arm.com/documentation/ihi0014/q/

[57] *CoreSight ETM-M4 - Technical Reference Manual*, ARM, June 2010, Rev. r0p1. [Online]. Available: https://developer.arm.com/documentation/ddi0440/c/functional-description/interfaces

[58] N. Gandraß, M. Rottleuthner, and T. C. Schmidt, "Work-in-Progress: Large-scale Timer Hardware Analysis for a Flexible Low-level Timer-API Design," in *Proceedings of EMSOFT 2021*. New York, NY, USA: ACM, October 2021, pp. 35–36. [Online]. Available: https://doi.org/10.1145/3477244.3477617

[59] *EFM32PG12 Family Data Sheet*, Silicon Labs, June 2022, Rev. 1.3. [Online]. Available: https://www.silabs.com/documents/public/data-sheets/efm32pg12-datasheet.pdf

[60] *STM32L476xx - Datasheet - Ultra-low-power Arm®Cortex®-M4 32-bit MCU+FPU, 100DMIPS, up to 1MB Flash, 128 KB SRAM, USB OTG FS, LCD, ext. SMPS*, ST, June 2019, DS10198 Rev 8. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32l476rg.html

[61] *Cortex-M4 Technical Reference Manual*, ARM, March 2010, rev. r0p1. [Online]. Available: https://developer.arm.com/documentation/ddi0439/b/

[62] "Appendix of Bachelor Thesis on Topic: Design and Evaluation of a Task Characterization Model for Performance Control of Embedded Devices at Runtime," retrieved 2022-11-6. [Online]. Available: https://github.com/aco401/task_characterization_model_for_performance_control

[63] *MSO1000Z/DS1000Z Series Digital Oscilloscope Programming Guide*, Rigol Technologies, Inc., December 2015, Rev. PGA19108-1110. [Online].

Available: https://www.batronix.com/files/Rigol/Oszilloskope/_DS&MSO1000Z/MSO_DS1000Z_ProgrammingGuide_EN.pdf

[64] *MSO1000Z/DS1000Z Series Digital Oscilloscope User's Guide*, Rigol Technologies, Inc., December 2015, Rev. UGA19110-1110. [Online]. Available: https://www.batronix.com/pdf/Rigol/UserGuide/MSO1000Z_DS1000Z_UserGuide_EN.pdf

[65] "SPEC's Benchmarks," retrieved 2022-10-17. [Online]. Available: https://www.spec.org/benchmarks.html#cpu

[66] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *4th annual IEEE international workshop on workload characterization. (WWC'01)*. IEEE, 2001, pp. 3–14.

[67] D. W. Chang, C. D. Jenkins, P. C. Garcia, S. Z. Gilani, P. Aguilera, A. Nagarajan, M. J. Anderson, M. A. Kenny, S. M. Bauer, M. J. Schulte, and K. Compton, "ERCBench: An Open-Source Benchmark Suite for Embedded and Reconfigurable Computing," in *International Conference on Field Programmable Logic and Applications (FPL'10)*. Milan, Italy: IEEE, 2010, pp. 408–413.

[68] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *30th annual Symp. on Microarchitecture (MICRO'97)*. IEEE, 1997, pp. 330–335.

[69] "MediaBench Consortium," retrieved 2022-10-17. [Online]. Available: https://cs.slu.edu/~fritts/mediabench/

[70] "MiBench2," retrieved 2022-10-17. [Online]. Available: https://github.com/impedimentToProgress/MiBench2

[71] "STMl476rg RIOT Docs," retrieved 2022-11-4. [Online]. Available: https://doc.riot-os.org/group__boards__nucleo-l476rg.html

[72] "RIOT - The friendly Operating System for the Internet of Things," retrieved 2022-11-5. [Online]. Available: https://www.riot-os.org/

Glossary

Flash Wait State Adaption A feature that adapts the amount of FWS when scaling the CPU frequency or voltage. The adaption is limited to constrains and should be performed correctly to preserve the working of the flash memory. Lowering the CPU frequency usually lowers the FWS.

RIOT "The friendly Operating System for the Internet of Things. Riot is a free, open source operating system developed by a grassroots community gathering companies, academia and hobbyists, distributed all around the world. [72]".

Task Characterization The process to trace a set of task properties for a given task with a task characterization model.

Task Characterization Model A component that dispenses task properties for a given task by tracing the task. It uses resources like for example cortex-M trace features to trace task properties. The resources are made accessible with a timer counter feedback mechanism. Software defines the start- and endpoint of a trace. The aim of the model is to capture task properties that indicate the performance utilization of the traced task.

Trace Method The type of trace feature that is used to measure a task property. The two types are profiling counter tracing and tracing via address matching with comparators.

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

__________________	__________________	____________________________________
Ort	Datum	Unterschrift im Original