FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Eva Dengler

Clock-Tree–Aware Resource-Consumption Models for Embedded SoC Platforms

Masterarbeit im Fach Informatik

30. September 2022

Please cite as:
Eva Dengler, "Clock-Tree–Aware Resource-Consumption Models for Embedded SoC Platforms", Master's Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, September 2022.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

Clock-Tree–Aware Resource-Consumption Models for Embedded SoC Platforms

Masterarbeit im Fach Informatik

vorgelegt von

Eva Dengler

geb. am 28. Juli 1998
in Ingolstadt

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Dr.-Ing. Peter Wägemann** **Simon Schuster, M. Sc.**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat**
Beginn der Arbeit:	**01. April 2022**
Abgabe der Arbeit:	**30. September 2022**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Eva Dengler)
Erlangen, 30. September 2022

ABSTRACT

As of today, more and more applications with both timing and energy constraints arise in embedded platforms. A famous example of such a platform is an artificial cardiac pacemaker. It has a safety-critical timing constraint, but also requires appropriate energy management to ensure the battery lasts until the next charging date in any case. To be able to give guarantees regarding these requirements, Worst-Case Execution Time (WCET) and Worst-Case Energy Consumption (WCEC) analyses are necessary to determine an upper bound of time and energy consumption. To make the recharging procedures as infrequent as possible, achieving the minimum energy consumption at a given timing constraint is the goal.

There are two main options to save energy in energy-constrained real-time systems: Either a component, e.g., the CPU or a peripheral device, is turned off entirely if not needed, or it operates at a lower speed. As a consequence, the system may need more time to complete its current task but uses less power. While a different component setting can save power, one has to consider that there are *penalties* regarding runtime and power consumption for the platform reconfiguration, which can outweigh the benefit of the lower consumption for both time behaviour and energy consumption. This applies to the main processing unit as well as peripheral devices on an embedded microcontroller. In addition, the activated and used peripherals can have a massive impact on these parameters, as well as the frequency these are running at. All these configuration options are based on the *clock tree* of the embedded chip, which is used to convert the input clocks to the required output signals for the CPU or peripheral devices.

Two objectives arise from this problem description and are dealt with in this work: First, this thesis develops a mathematical model to determine the energy-optimal solution of a sequential set of tasks for a given hardware platform, which still gives guarantees regarding the execution time of the program. It considers the theoretical WCET and the available *clock-tree configurations* for required frequency and device configurations for each task, the corresponding power consumption, a periodic deadline for the program, and the penalties for changing between different clock-tree configurations to determine the best possible solution for the set of tasks. The second objective is to check whether this model can be used to determine the minimal power consumption of real hardware for a given set of tasks. Therefore, a hardware model is created for a hardware platform, the ESP32-C3 (a RISC-V single-core microprocessor), and integrated into the open-source analysis tool PLATIN. This model is validated with a benchmark suite and compared to external time measurements. Finally, power consumption measurements were performed for the ESP32-C3, and the energy-optimal solutions for two example tasks are determined with the mathematical model. In comparison to a pessimistic approach that does not selectively reconfigure clock trees the optimisations achieved significant energy savings.

KURZFASSUNG

In der heutigen Zeit gibt es für eingebettete Systeme immer mehr Anwendungen, bei denen sowohl Zeit- als auch Energiebeschränkungen eingehalten werden müssen. Ein Beispiel hierfür sind künstliche Herzschrittmacher. Zum einen müssen zeitliche Garantien gegeben werden, zum anderen erfordert es aber auch ein angemessenes Energiemanagement, um sicherzustellen, dass die Batterie auf jeden Fall bis zum nächsten Aufladetermin ausreicht. Hierfür werden Worst-Case Execution Time (WCET)- und Worst-Case Energy Consumption (WCEC)-Analysen verwendet, um Obergrenzen für Zeit und Energieverbrauch zu bestimmen. Zusätzlich möchte man die Aufladevorgänge so selten wie möglich durchführen. Daher ist es das Ziel, den minimalen Energieverbrauch eines Systems bei einer gegebenen Zeitbeschränkung zu erreichen.

Um Energie in energiebeschränkten Echtzeitsystemen einzusparen, gibt es zwei Möglichkeiten: Entweder werden Komponenten komplett abgeschaltet, oder sie arbeiten mit einer geringeren Geschwindigkeit. Infolge dieser Sparmaßnahmen benötigt das System möglicherweise mehr Zeit, um Aufgaben zu erledigen, verbraucht dadurch aber weniger Strom. Hierbei ist zu bedenken, dass durch eine Neukonfiguration zusätzliche *Nachteile* in Bezug auf die Laufzeit und den Stromverbrauch entstehen, die den Vorteil des geringeren Verbrauchs zunichte machen können. Darüber hinaus haben Peripheriegeräte einen massiven Einfluss auf das Zeit- und Energieverhalten, ebenso wie die Frequenz, mit der diese betrieben werden. Somit hängt der Energieverbrauch des Chips von dessen *Clock Tree* ab, der zur Umwandlung der Eingangstakte in die erforderlichen Ausgangssignale für die CPU oder die Peripheriegeräte verwendet wird.

Hieraus ergeben sich zwei Ziele für diese Arbeit: Zunächst wird ein mathematisches Modell entwickelt, um die energieoptimale Lösung für die sequentielle Abarbeitung eines Sets von Aufgaben zu bestimmen, während die Laufzeitgarantien weiterhin gegeben bleiben. Dazu werden die theoretische WCET, verfügbare *Clock-Tree-Konfigurationen* für jeden Programmabschnitt, die entsprechende Leistungsaufnahme, eine periodische Deadline sowie die *Kosten* bei Clock-Tree-Neukonfigurationen betrachtet, um die bestmögliche Lösung zu finden. Das zweite Ziel besteht darin, dieses Modell auf Tauglichkeit für reale Hardware zu testen. Hierzu wird ein Hardwaremodell für den RISC-V Single-Core Mikroprozessor ESP32-C3 erstellt und in das Open-Source Analysewerkzeug PLATIN integriert. Nachdem das Modell mit einer Benchmark-Suite validiert wurde, folgen Stromverbrauchsmessungen, um im Anschluss die energieoptimalen Lösungen für zwei Beispielszenarien zu bestimmen. Dies führte im Vergleich zu einem pessimistischen Ansatz ohne Geräterekonfigurationen zu signifikanten Energieeinsparungen.

CONTENTS

Abstract v

Kurzfassung vii

1 Introduction 1

2 Fundamentals 3
2.1 Real-Time Systems . . . 3
2.2 WCET Analysis . . . 5
2.2.1 Static Runtime Analysis . . . 5
2.2.2 The Implicit Path-Enumeration Technique . . . 5
2.2.3 Integer Linear Programs for the Implicit Path-Enumeration Technique . . . 6
2.2.4 Constraints of the Integer Linear Program . . . 7
2.2.5 Hardware Model . . . 7
2.3 Hardware Platform . . . 8
2.3.1 Clock Tree . . . 8
2.3.2 Power Modes . . . 9

3 The Problem of Unknown Optimal Power Configurations 11

4 Approach 15
4.1 Description of the Linear Programming Problem . . . 16
4.2 Modelling Multiple Sleep Options . . . 19

5 Evaluation of the Solver Performance 23
5.1 Theoretical Problem Complexity . . . 23
5.2 Solver Efficiency . . . 25
5.2.1 Integer Linear Program versus Quadratic Program . . . 26
5.2.2 Problem-Size–Scaling Behaviour of the Solver . . . 26
5.3 Viability of the Problem Description . . . 27

6 Implementation for the ESP32-C3 31
6.1 The ESP32-C3 . . . 31
6.1.1 Hardware Overview . . . 31
6.1.1.1 Chip Details . . . 31
6.1.1.2 Memory . . . 32
6.1.2 Espressif IoT Development Framework . . . 32

6.1.3 Clock Tree . . . 33
6.1.4 Power Modes . . . 33
6.2 PLATIN . . . 34
6.2.1 Toolchain Setup for the ESP32-C3 . . . 34
6.2.2 Timing Behaviour of the ESP32-C3 . . . 34
6.2.3 Platin Evaluation . . . 36

7 Evaluation for the ESP32-C3 **39**
7.1 Measurements . . . 39
7.1.1 Measurement Setup . . . 39
7.1.2 Active Modes . . . 41
7.1.3 Peripherals . . . 42
7.1.4 Light Sleep and Deep Sleep . . . 46
7.1.5 Reconfiguration . . . 46
7.1.5.1 Switching between Frequencies . . . 46
7.1.5.2 Configuring Devices . . . 46
7.1.5.3 Light Sleep . . . 49
7.1.5.4 Deep Sleep . . . 49
7.2 Real Values for the Quadratic Program . . . 51
7.2.1 Single Task . . . 51
7.2.2 Inter-Integrated Circuit Communication Test . . . 54
7.3 Conclusion for the ESP32-C3 . . . 56

8 Related Work **59**
8.1 Multi-Objective Optimisation . . . 59
8.2 Clock-Tree Configurations . . . 60
8.3 Power Management . . . 60
8.4 Energy-Constrained Real-Time Systems . . . 61

9 Conclusion **63**

Lists **65**
List of Acronyms . . . 65
List of Figures . . . 67
List of Tables . . . 69
List of Listings . . . 71
Bibliography . . . 73

1 INTRODUCTION

Nowadays, many computers use powerful but complex instruction sets, allowing for more optimised single instructions and therefore improving the overall performance. While this is handy for the actual runtime, it makes the time and power consumption of the Central Processing Unit (CPU) hard to predict, as the power usage of an instruction depends heavily on the input parameters, as seen in Pallister et al. [26]. The same holds for the time usage. When having a look at embedded real-time platforms, this leads to an overestimate for both Worst-Case Execution Time (WCET) and Worst-Case Energy Consumption (WCEC), as one has to use the most pessimistic outcome of an instruction to stay sound. Nevertheless, for such systems, where time and energy are scarce resources, an accurate model is essential - first, in terms of functionality, second, in terms of safety. If a system does not meet a deadline, it does not provide the result of the task at a specific time. If it uses more energy than expected, and the battery in the system runs empty, a task cannot be finished. On the other hand, if the system does not make use of its available time or energy, that resources are wasted and the real potential of the system is underestimated.

The time and the energy consumption depend heavily on the underlying hardware platform. If the microprocessor cannot be modelled realistically, the analysis may be too pessimistic for being useful for developing real-world products. In addition, not all available CPU resources are used, as tasks can be bound by input/output operations or the CPU be a powerful one, which results in idling. On some platforms, it is possible to reconfigure the clock tree. The *clock tree* is a network of input sources, scalers, multiplexers, and gates to produce signals for the CPU or the peripherals on the chip. It can be modified to deliver a lower frequency for the CPU to run at, lowering the power consumption. The clock tree has a massive impact on runtime, energy consumption and the frequency of activated and used peripherals. Turning off devices can save energy, but it introduces *penalties* as additional time and energy to change the *clock-tree configuration*. Furthermore, not all peripherals can be used at all frequencies or support switching between two frequencies while the device is in use. All of this has to be considered when looking at a task, e.g., it could be advantageous not to turn off a device when it is used again later as turning it off and on again is expensive. However, the power consumption would significantly drop while it is deactivated.

To determine a power-optimal solution for a sequential series of tasks while still being able to give guarantees regarding the execution time of the program, this thesis presents a mathematical optimisation problem, which a mathematical problem solver can solve. It considers the theoretical WCET of sections of each task, the available frequency and device configuration settings for each phase, their corresponding power consumption, a periodic deadline, and the costs to change between different device configuration states to determine the best possible solution.

To be able to evaluate the mathematical model, a suitable hardware platform has to be found. It must be appropriate for energy-constrained real-time systems; as such, a valid and accurate

hardware model is required. Such a hardware model is created for a hardware platform candidate, the ESP32-C3, and integrated into the open-source analysis tool PLATIN [28]. The model results are validated with a benchmark suite by comparing the WCET determined by PLATIN with time and cycle-counter measurements. Together with power-consumption measurements for the ESP32-C3 and the hardware model in PLATIN, the mathematical model is tested whether a lower power consumption can be achieved by adding clock-tree configuration changes.

In the following, the design and implementation are described in detail, as well as the evaluation. Therefore, first of all, a brief description of real-time systems, an overview of WCET analysis and an introduction to the hardware configurability of embedded microcontrollers is given in Chapter 2. Chapter 3 follows with a comprehensive description of the optimisation problem described above. A solution for this problem is presented in Chapter 4, whose efficiency is tested and evaluated in Chapter 5. After that, the focus changes from a theoretical point of view to a practical one in Chapter 6. There, the implementation for the ESP32-C3 and its integration into PLATIN are detailed. Chapter 7 continues this work to be able to evaluate of the model from Chapter 4 on the ESP32-C3. To conclude this thesis, Chapter 8 discusses related work before a summary is given in Chapter 9.

2 FUNDAMENTALS

Before this thesis details the problem description and the solutions for it, a set of fundamentals is needed. First, Section 2.1 gives a brief introduction to real-time systems. Section 2.2 follows, which explains the term WCET. Afterwards, Section 2.3 details the hardware requirements needed to perform WCET analysis and changes to time and power consumption.

2.1 Real-Time Systems

A real-time system has different demands compared to a typical desktop user system. In addition to being reactive to user inputs, calculations are tied to real-time requirements. The units of work, also known as tasks, to be completed by the system have to be functionally correct and meet timing constraints. They have an instant of time at which they become available for execution (which also can be the start of the system) and have to be finished by a given deadline. That can either be a point in time by which the execution is required to be completed (known as *absolute* deadline) or it has a response time, also known as *relative* deadline, from which the absolute deadline is calculated as the start point plus the relative deadline [18, Chapter 2.2].

Contrary to what might be expected from the term real-time, real-time systems do not necessarily have to be fast but meet their given deadlines. If the system cannot meet such a deadline, there are different ways the system deals with that:

- In *soft* real-time systems, missed deadlines can be tolerated. However, a late calculation result is undesirable, as the results lose relevance over time. Examples are DVD players or multimedia systems.
- *Firm* real-time systems can also tolerate missed deadlines, but the result of the calculation gets useless after the deadline and is discarded. Communication systems can be found in this category.
- Missing a deadline in a *hard* real-time system is a fatal fault because a late result may have disastrous consequences. Therefore, missing it can not be tolerated. This is the case for heart pacemakers, flight control systems or airbag control.

The applications of real-time systems can be categorized into four types [18, Chapter 1.5]:

- Purely cyclic: Every task executes periodically, so the behaviour of the system is deterministic.
- Mostly cyclic: Most tasks execute periodically, but the system must also respond to external events such as interrupts.

- Asynchronous and somewhat predictable: Most tasks are not periodic, but the variations in time of each task have either bounded ranges or known statistics.
- Asynchronous and unpredictable: Most tasks are not periodic and cannot be predicted.

In this work, periodic tasks in a hard real-time–system environment will be analysed. These aspects will be detailed in the following.

The *execution time* of a task is the amount of time required to complete execution on a specific hardware when it executes alone on a system and has all required (hardware) resources available. That time can vary for a task, as it depends on the underlying hardware and on the given input. First, the influence of the input is analysed. Take the sorting algorithm BubbleSort for a list of n integers as an example (Listing 2.1): If the input is already given sorted, it takes $\mathcal{O}(n)$ steps for the function to return a sorted list. However, $\mathcal{O}(n^2)$ instructions are needed in the worst case. This leads to two boundaries: The minimum and the maximum execution time. Most of the time, in real-time systems, primarily the maximum execution time is of interest, as this sets the boundary on whether the task can always be completed before its deadline or not. As the minimum, average, and maximum execution time can differ by far, using the maximum as execution time can lead to massive underutilisation of a system. This is justified by the fact that most hard real-time systems are safety-critical. For example, it can be tolerated that the crash-control system in a car idles most of the time, but it is unacceptable if the airbag does not work in the case of a crash. Therefore, in most hard real-time systems, the variations in task execution times are kept small to minimise the gap between minimum and maximum execution time: This causes an overestimation of the actual requirements, leading to unused resources. Additionally, the hard real-time portion of a system is often small, and thus the overestimation is not that problematic [18, Chapter 3.2.2].

In the airbag example, the task to check whether a crash occurred or not has to be executed regularly. That can be depicted with a periodic task model. Assume there are tasks T_i, and each task T_i has a period p_i. The periods p_i are the minimum lengths of all time intervals between release times of consecutive tasks of T_i. The hyperperiod H is the amount of time when the release times of all tasks overlap in the same pattern again. It can be determined by calculating the least common multiple of all p_i [18, Chapter 3.3].

```
1  void bubbleSort(uint8_t size, uint8_t array[]) {
2    uint8_t n = size;
3    bool swapped = false;
4    do {
5      swapped = false
6      for (uint8_t i = 0; i < n-1; ++i) {
7        if (array[i] > array[i+1]) {
8          swap(array[i], array[i+1]);
9          swapped = true;
10       }
11     }
12     --n;
13   } while (swapped);
14 }
```

Listing 2.1 – A BubbleSort implementation.

2.2 WCET Analysis

For now, focus on one periodic task and return to the problem of the maximum execution time. To meet the timing criteria of a hard real-time system, tasks need to meet their deadline. To ensure that a task is completed within its period, the WCET has to be determined beforehand. The most straightforward approach to that - execute all possible inputs - can become time-consuming. Take the BubbleSort example (Listing 2.1) from above. With `uint8_t` as data type there are

$$(2^8)^{(2^8-1)}$$

different input lists with a maximum length of $2^8 - 1$ and 2^8 possibilities for each value to consider. This number has 615 decimal digits, so it is clear that testing all inputs is no feasible solution except for small input ranges.

2.2.1 Static Runtime Analysis

Another option to determine an upper bound for program execution is to perform a static runtime analysis. Therefore, two preparation steps are necessary [17]:

1. Program-path analysis: A path analysis is run to explore path constraints and exclude infeasible paths if possible. This is done by building a Control-Flow Graph (CFG) for the program, which is explained in the next section.

2. Micro-architectural modelling: As any program's execution time heavily depends on the underlying hardware platform, it is required to perform a thorough hardware and architecture analysis to determine timings for each program block.

With all this information available, the WCET can be computed. First, the following section starts with an explanation of how a CFG is built and used in the analysis.

2.2.2 The Implicit Path-Enumeration Technique

A CFG represents the program flow of an analysed program. The assembler instructions are grouped into so-called Basic Blocks (BBs) [17]. Each BB has exactly one entry point and exactly one exit point. That means this code segment is always executed entirely or not at all (if no interrupt occurs). Also, there are no jumps into the middle of the block or from the middle to somewhere else. Each BB acts as a node in the CFG. BBs can also be extended to the notion of Atomic Basic Blocks (ABBs) [33] and Power Atomic Basic Blocks (PABBs) [34, 41]. ABBs are defined as atomic sections from a scheduling point of view and can span over multiple BBs. A PABB then generalises this idea to include power-consumption configurations by adding the restriction that the system's energy consumption shall not change within one PABB. This includes that no changes to the device configurations are made during a PABB. If the control flow passes between two BBs, an edge is inserted into the CFG. For each node, a variable is introduced (x_i), similar for each edge ($e_{i,j}$), which states how often this node or edge is used during a program flow. The WCET is then calculated as the maximum of the sum of the costs per basic block c_i times the execution frequency x_i (with the assumption that the edges do not have any costs):

$$\max \sum_{i=1}^{N} c_i x_i$$

The x_is have restrictions, which are given by the CFG - more on that in Section 2.2.4. One could look for the maximum by manually searching for the longest possible path through the graph, but this is, besides not being feasible, not very functional. At this point, the idea behind the Implicit Path-Enumeration Technique (IPET) [17, 29] comes in: In general, it is not necessary to identify the exact worst-case paths, but just the worst-case execution time. Therefore, the problem of determining the bounds of each x_i is converted into an Integer Linear Program (ILP), which then can be solved by a problem solver. The following section briefly introduces the usage of ILPs for IPET.

2.2.3 Integer Linear Programs for the Implicit Path-Enumeration Technique

An Integer Linear Program (ILP) consists of an objective (or goal) function with n frequency variables x_i to be either maximised or minimised, and a set of m constraints, which describe the constraints for variables used in the objective function. The canonical form of an ILP is:

$$
\begin{array}{rlrcl}
\text{maximise} & \displaystyle\sum_{i=1}^{n} & c_i x_i & & \\
\text{subject to} & \displaystyle\sum_{i=1}^{n} & a_{i1} x_i & \leq & b_1 \\
& \displaystyle\sum_{i=1}^{n} & a_{i2} x_i & \leq & b_2 \\
& & \cdots & & \\
& \displaystyle\sum_{i=1}^{n} & a_{im} x_i & \leq & b_m \\
& \forall i: & x_i & \geq & 0 \\
\text{and} & \forall i: & x_i & \in & \mathbb{Z}
\end{array}
$$

The c_i are the cost of each frequency variable, where in the context of WCET analysis the x_i corresponds to the execution frequency of a BB and the c_i to the time needed by this BB. The a_{ij} are the influence of the i-th frequency variable on the j-th constraint, which is bounded by b_j. The variables $x_i \in \mathbb{Z}$ ensure that only linear solutions are valid - meaning that a BB must be executed in its entirety. If one or more variables can also be real numbers instead of just integers, the problem is called a mixed-integer linear programming problem. The canonical form of an ILP as presented above is often needed for passing a problem description to an ILP solver, e.g., lpsolve [21] or Gurobi [14]. Other forms, e.g., when the conditions are malformed or the goal is to minimise a function, can be adapted easily to match the required canonical form. An optimal solution is any solution where the constraints are satisfied and the value of the objective function is maximised.

In general, ILPs are NP-complete, meaning that a solving algorithm has an exponential worst-case complexity. Li and Malik [17] showed that an actual blowup did not occur for their WCET analyses as the constraints matched a network flow problem, which can be solved in polynomial time. However, in principle, the full range of possible constraints can lead to a general ILP. The following section looks at the constraints needed to describe a task as an ILP.

2.2.4 Constraints of the Integer Linear Program

Mainly two constraints restrict the previously introduced x_i:

- Structural constraints: These are directly extracted from the CFG.
  For example, the number of ingoing edges must be the same as the execution frequency of the BB, and the execution frequency of the BB must equal the number of outgoing edges.
- Functionality constraints: These have to be provided by the user as additional information.
  For example, data-flow analysis or the programmer itself can add information on how many times a loop will be executed at maximum. This information can be used as follows: When examining loops, there are two different types of ingoing edges: Entry edges, where the program flow enters a BB the first time, and back edges, which are the connections from the end of the loop back to the beginning. If an upper bound of the loop-body executions is known, the number of edges can be limited: The sum of all back edges must be smaller or equal to the sum of entry edges times the number of loop executions, as each entry edge can trigger as many as n executions of the first outgoing edge.

How the exact modelling of a program into an ILP works would exceed the fundamentals required for this work. For further reading, the modeling procedure is detailed in Li and Malik [17] or Ballabriga and Cassé [3].

2.2.5 Hardware Model

The execution time of one BB heavily depends on the underlying hardware platform. The hardware model to determine these costs can be of variable complexity and, therefore, variably precise. A simple model analyses the execution time of each assembler instruction in the BB and sums up all (upper) bounds to determine the WCET of a BB. Additionally, models for pipelines and caches exist to improve the accuracy [15, 30, 44] and thus minimise the error of the WCET estimation. However, cache modelling can become quite hard, and several works have already addressed this topic [15, 39]. Depending on the level of detail of the hardware modelling, one can thus obtain a more and more accurate model of the program behaviour on the system. The downside of this is that the time required for the analysis also increases. Whether the higher analysis effort for developing the more precise outcome and the longer analysis time is worth the more accurate estimate depends on the use case of the model. When creating a hardware model, it should be taken care of an evaluation regarding the presence or absence of timing anomalies and, if there are any, model them accordingly. Timing anomalies [6, 22] are situations where the local WCET does not lead to the global WCET. One famous example is when a cache miss as local worst case can result in a shorter global WCET due to scheduling effects for out-of-order execution. If timing composability does not apply, WCET analysis becomes more difficult and time-consuming [31].

Altogether, the WCET of a program may be overestimated. However, the proposed techniques can determine the WCET in a feasible amount of time. Also, the analysis does not miss any outliers to the higher end, which would be a disaster for a hard real-time system.

2.3 Hardware Platform

The heart of controlling timing and energy consumption of a chip is its clock tree. Section 2.3.1 explains the term and how it works. Depending on the clock-tree configuration, a chip operates in different modes. With regards to power consumption, these are known as power modes, which are detailed in Section 2.3.2.

2.3.1 Clock Tree

A *clock tree* is a clock-distribution network on a microcontroller chip. Its task is to distribute and route the source clock signals to all internal components. A network of source, intermediate, and sink nodes is used. The complexity of the network depends on the chip. Intermediate nodes can be multiplexers to choose between multiple inputs for its single output, multipliers or dividers to scale the frequency, or clock gates to completely shut off single devices or a whole subtree from the network. Figure 2.1 shows an example: There are three source clocks, namely `PLL`, `RC` and `OSC`. These three source signals are routed to three output clocks: `CPU_CLK`, `WIFI` and `BLE`. In between is a clock tree, which is made of scalers, multiplexers and clock gates. With these, a broad range of output clocks can be achieved: For example, the `WIFI` signal can either be configured by taking the `PLL` signal or by choosing the `RC` signal. This is then divided by `FOSC_DIV`, selected by `MUX1` instead of the `OSC` signal and then divided again by `DIV`. Finally, `MUX3` chooses this signal before the `GATE` is set to be open such that `WIFI` receives the signal. If `WIFI` is not used, the `GATE` can be closed such that `WIFI` does not use additional energy and therefore reduces the energy consumption of the chip, which is power over time.

The clock tree needs to be configured accordingly to provide the frequencies required by the application and the peripherals. For example, the standard operation mode of the WiFi chip of an ESP32-C3 [9] only works with fast clocks, while slow clocks are required for low-power operation. So if one needs the WiFi chip to be active, the clock tree has to be configured such that a fast clock drives the WiFi chip. There are two options for configuring the clock tree: Configuration before and setup during startup, or (re-)configuration during runtime. The hardware usually offers interfaces to reconfigure the clock tree, although not all parts of a clock tree allow the system to be reconfigured during runtime. Often memory-mapped hardware registers contain

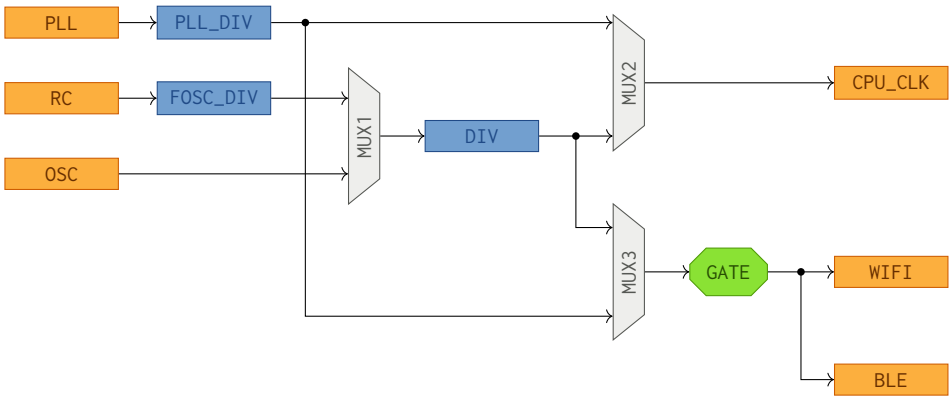


Figure 2.1 – A clock tree with three source clocks and three output clocks. In between, a clock tree, made of scalers, multiplexers and clock gates, configures the values of the output clocks.

the configuration, and writing them triggers a state change, e.g., setting a divider value or gating peripherals. Sometimes, writing a single value is all it needs to reconfigure a clock. However, other clocks introduce further *reconfiguration penalties*: Some take time until a stable clock is reached again, or changing one clock influences the whole system, e.g., changing the frequency for a subtree and all devices on it or turning off devices that are not supposed to be switched off [32]. There are also two options for how reconfiguration happens: either all configuration changes are known beforehand, which enables optimisations at compile time. The other option is to dynamically react to clock-tree reconfiguration requests, which makes a system more adaptable to application demands. But, as a consequence, every change has to be checked on compliance with the restrictions of other active requests and requirements.

2.3.2 Power Modes

Power modes of a system are predefined configurations of the clock tree. Examples are light sleep or deep sleep, where the CPU clock is turned off. A Power-Management Unit (PMU) controls the power supply to different parts of a chip. It decides which power domains stay powered on and which are powered off in each power mode. A power domain describes a section of a chip, e.g., CPU, clock sources, or peripherals. To switch from a high power mode to a mode where the CPU is powered down, the PMU is configured to react to a wakeup source before entering a sleep mode. This can be a GPIO pin or happen via UART, which then powers up the CPU again after it was in low-power mode. When choosing a power mode with lower energy consumption, often only a limited set of features of the chip are available. The other way round also holds: If not all features are needed, one does not need as much power as before due to the fact that a power mode with less energy consumption can be used.

3 THE PROBLEM OF UNKNOWN OPTIMAL POWER CONFIGURATIONS

Determining the WCET is a central part of research in the field of real-time systems [44]. The problem is mostly solved for single tasks or applications [13]. Besides that, strategies to provide timing behaviour guarantees for whole systems are also available [1, 2]. The aspect of energy consumption for individual applications or even the entire system is an ongoing research challenge, as each device and its activation or deactivation influence the total system behaviour [41]. When looking at such energy-constrained real-time systems, not only time but also energy-consumption guarantees have to be met. A popular example of such a system is an artificial cardiac pacemaker: The electrical impulses must regularly and reliably provide a signal to perform the cardiac contraction – the time component of the requirements of such a device. The energy output of the battery is just as important: There must always be enough energy to control the signalling system, otherwise, this leads to system failure, an unacceptable state. An additional goal is to minimise the overall energy consumption, which means to optimise the usage of the available resources according to the needs of (a set of) given tasks, such that the energy source lasts longer. In the case of the artificial cardiac pacemaker, its battery does not have to be controlled and changed as often.

To tackle this problem in a first step, only a single, periodic task with a previously known number of phases is examined in the context of an energy-constrained real-time system. Each phase has different requirements regarding the clock-tree configuration, which induces possible peripheral device usage. In between the phases, the system has the option to be reconfigured by changing the clock-tree configuration of the microcontroller to e.g., power on or turn off peripherals. This allows the system to adapt to the requirements of each phase if necessary. An example is given in Listing 3.1: It consists of three phases with different constraints on the clock-tree configuration. After the three phases, it idles until the next period starts. From one phase to the next, the *reconfiguration points* allow the system to adjust the clock tree to meet the demands of each phase. These mode changes introduce penalties regarding time and power consumption, which must be taken into account. For example, the time to wake up the ESP32-C3 from deep sleep takes 300 ms.

A safe way to ensure the power source does not run out of energy earlier than expected is to assume the maximum power consumption of the microcontroller at all times. This leads to higher predicted energy consumption and therefore shorter operation period of the energy source, which, in turn, means that e.g., a battery has to be replaced sooner, although it was not yet required. A configuration like this for the example from Listing 3.1 is shown in Figure 3.1a. The power demand is shown as a curve, and the shaded area underneath corresponds to the energy consumption, as

```
1   while(true):
2     reconf(idle, p1);
3     execute(p1);
4     reconf(p1, p2);
5     execute(p2);
6     reconf(p2, p3);
7     execute(p3);
8     reconf(p3, idle);
9     wait(idle, until=period_end);
```

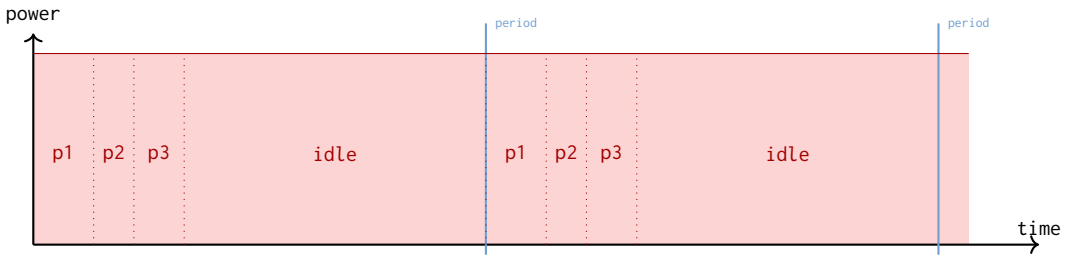
Listing 3.1 – Example task for the problem description: A periodic task with three sequenced phases `p1`, `p2`, `p3` and a sleep phase afterwards is given. In between each two of those, a clock-tree reconfiguration can occur and thus change the microcontroller timing behaviour and energy consumption.

it equals the power demand multiplied by time. The actual energy consumption can be far less, e.g., by using feedback-based optimisation methods: Chiang et al. [7] introduce a kernel-based dynamic clock management system. It reduces energy consumption by changing the clock of an embedded microcontroller based on ongoing computations and I/O requests. They claim that slower clocks with a lower power consumption are more suitable for fixed-time I/O operations, while pure computations shall be done with faster clocks as these use less energy per clock tick. An example of such a dynamic reconfiguration during runtime is shown in Figure 3.1b. In the first period, slower clocks lead to less energy consumption than in Figure 3.1a, but to a longer runtime of the phases themselves. This behaviour still meets the requirement of the timing constraint, as all tasks are finished before the next period starts. In the second period, the dynamic reconfiguration is acting more aggressively and, in some phases, uses even slower clocks. Again, the energy consumption drops, but the periodic deadline is no longer met. This is unacceptable and leads to system failure in hard real-time systems. That approach also has several other problems:

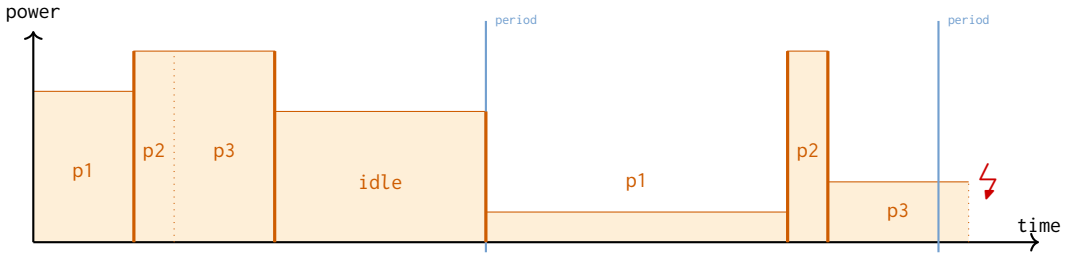
- To select a more suitable configuration, feedback on the current system - first - has to be collected and - second - has to be evaluated. Since both tasks inevitably require extra time and computing power and therefore consume additional energy, the result cannot be optimal.
  Solution: Use static analysis before program execution to determine the most suitable configurations without runtime overhead.

- The feedback loop only reacts to the currently running application. It thus neither can take a better configuration into account for the previous task nor prepare the system for the next task in advance. This is seen in the first period of Figure 3.1b, where a longer execution time of a phase and, therefore, lower power consumption is still acceptable.
  Solution: A static analysis of the complete task and its environment before runtime uses this information to construct configurations for the overall task set. As a result, the energy-optimal solution is obtained.

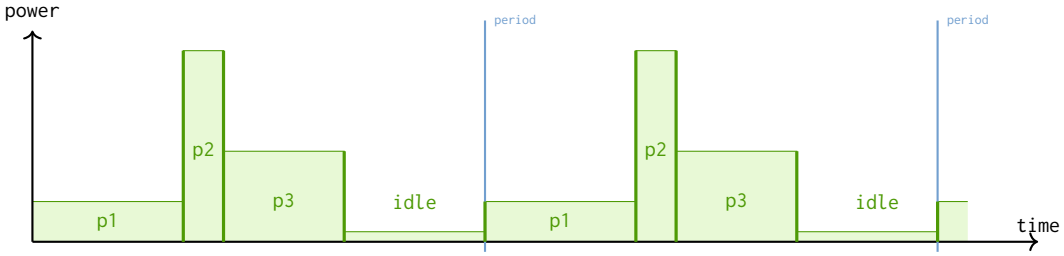
- The outcome of the feedback evaluation is unknown beforehand. This can lead to problematic behaviour like in the second period of Figure 3.1b: A deadline miss occurs. Even worse, no guarantees for the execution time or power consumption can be given, which is disastrous for hard real-time systems.
  Solution: By using WCET analysis, timing behaviour is determined before runtime and, as consequence, guarantees for real-time systems can be given.



(a) Option 1: Always-on approach. All devices on the microcontroller are active, therefore the maximum power consumption is assumed.



(b) Dynamic, feedback-based optimisation: The system has the option to reconfigure the microcontroller during operation to reduce power consumption.



(c) Static optimisation with timing guarantees: By collecting information about energy consumption, device configuration constraints and timing behaviour, an optimal solution is determined beforehand.

Figure 3.1 – Finding the optimal power configuration for the example from Listing 3.1. Three different strategies are displayed and described above.

All these problems are tackled with the approach presented in this thesis: By using a mathematical model (further description follows in Chapter 4) for a static analysis before execution, no runtime overhead is introduced. It uses information about the whole system with its clock-tree configurations for all tasks, which shall be executed periodically with a given hyperperiod or deadline. Therefore, the model also evaluates whether it is advantegous to leave a peripheral device turned on even if not needed in the following phase(s) instead of accepting the reconfiguration costs occuring with powering off and on the device for both energy-consumption minimisation and timing guarantees. WCET analysis is used to determine the expected cycle count of each phase, from which - together with the clock-tree configuration - both time and energy consumption behaviour are derived for each phase. This is used as input for the mathematical model, together with the power consumption for each clock-tree configuration and the penalties for reconfiguration. With all this information, guarantees for meeting the periodic deadline and minimal power consumption can be given. As a result, an optimal power-consumption curve, as shown in Figure 3.1c, can be achieved. The next chapter describes the mathematical model, which expresses all requirements.

4 APPROACH

This chapter introduces the chosen approach to tackle the problems described in Chapter 3. It introduces the overall theoretical problem description for determining time- and power-optimal frequency settings and clock-tree configurations and discusses possible design options.

As in the previous chapter, it is assumed that the required power consumption for all available clock-tree configurations and all penalties for changes between clock-tree configurations are known. This includes changing the frequency of the CPU, turning on peripherals, using them, and turning them off. A periodic task like the one shown in Listing 3.1 is given, which is split into phases. Just one task is running at a point in time. The order of the task's phases is fixed. Splits between two phases are called reconfiguration or decision points. At these points, the clock-tree configuration and, therefore, the power consumption of the system can, but does not need to, be changed - e.g., power on a device, change to a lower frequency for I/O-bound operations or choose a higher frequency for computation-intense operations. Therefore, there are no other points in time where the power consumption of the device changes. For each phase, the worst-case number of CPU cycles is determined with a WCET analysis. The maximum energy needed for a phase can be calculated by multiplying that number with the current frequency and the power consumption per time, which heavily depends on the used peripheral devices. The period length of the periodic task is given, which is also the hyperperiod of the system, as only one phase of the sequential tasks is running at each point in time. The goal is to determine the energy-optimal solution with regards to the choice of frequencies and clock-tree configurations per phase, which still meets the give time constraint - the hyperperiod. As stated by Chiang et al. [7], clock choice is essential to the system's energy consumption. Slower clocks drain less energy per time, but fast clocks are more energy-efficient for computation-intense tasks on microcontrollers, as the system draws more power but less per CPU cycle. That means the best trade-off between power consumption, computing speed and clock-tree reconfiguration penalties is to be determined.

The possible clock-tree configuration changes can be represented as an acyclic graph, as shown in Figure 4.1. Each phase has a set of possible configurations, representing each as a node in the graph. A node has a cost depending on the chosen clock-tree configuration, which corresponds to a set frequency and power consumption, and the overall time, which depends on the number of CPU cycles. Between each configuration of a phase and all configurations of the next phase, a transition by changing the clock tree (possibly in multiple steps) is possible and represented as an edge in the acyclic graph. If a change is not viable, the corresponding edge is not part of the graph. The edges have the costs of the power consumption of the corresponding transition. The `S` and `E` nodes resemble the start and end of a task during a period. The remaining time between `S` and `E` is spent in sleep mode; the edges from `S` or to `E` represent the costs of waking up or going to sleep. The

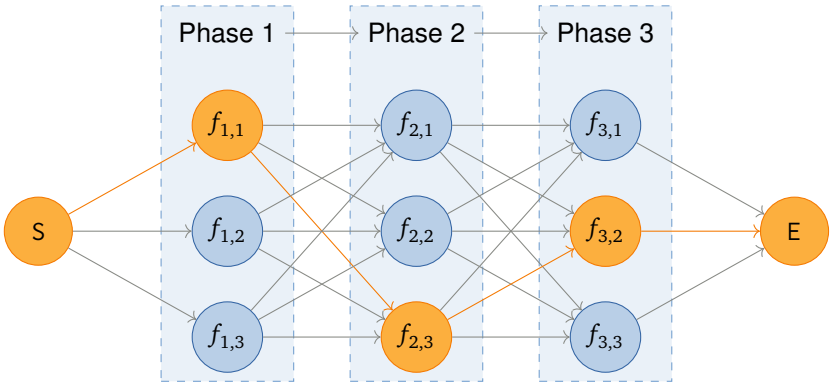


Figure 4.1 – Graph representation of the mathematical description: The possible changes between all clock-tree configurations for each phase can be represented as edges of a cyclic directed graph. Each edge resembles a valid change. Each edge and node has resource (time and energy) costs. In the end, exactly one path from S to E is active. In this example, there are three phases, where each one has three possible settings. All changes are possible, therefore, all edges are displayed in the graph.

time in sleep is calculated as the hyperperiod without all time needed in the previous phases and transitions, depending on the choice of clock-tree configurations.

The goal is to find a way through the graph from S to E where exactly one node is selected in each layer - as per phase, exactly one configuration can be selected. In addition, the most cost-effective solution is to be determined. Therefore, the graph describes a minimum cost, maximum flow problem with an additional constraint on the total time needed by all nodes and edges, which needs to be less than the given hyperperiod, and a flow of 1. Starting with a flow of 1 at S, a flow with minimum cost through the network by selecting the best combination of nodes and edges has to reach E. Min-Cost-Max-Flow problems can be modelled as ILPs. As this problem falls into this problem class, but with additional constraints, a suitable ILP model is presented in the next section.

4.1 Description of the Linear Programming Problem

Let

- $x \in \mathbb{N}$ be the amount of phases of the current problem,
- $c_i \in \mathbb{N}$ the amount of cycles in phase i,
- $f_i \in \mathbb{N}$ the number of available frequencies for phase i,
- $\forall j \in \{0, \ldots, f_i - 1\}$. $f_{i,j} \in \mathbb{Q}$ all available configurations for phase i,
- $\forall j \in \{0, \ldots, f_i - 1\}$. $p_{i,j} \in \mathbb{Q}$ the corresponding power consumption and
- $H \in \mathbb{Q}$ the hyperperiod time.

The time for one phase i with configuration $f_{i,j}$ is defined as function $t_{i,j}()$, which shall return a fixed number. For example, the function depends on the WCET and the frequency of the current configuration, where the frequency is given as how many CPU cycles can be executed per second.

Then, the time for this phase is calculated as the product of amount of CPU cycles, as determined by WCET analysis, times the CPU frequency of the current setting,

$$t_{i,j}() = c_i \cdot f_{i,j} \ .$$

Similarly, the energy consumption of a phase i with a configuration $f_{i,j}$ is defined as function $e_{i,j}()$. For example, the energy consumption is defined as product of time and the power consumption, which results in

$$e_{i,j}() = t_{i,j}() \cdot p_{i,j} \ .$$

For simplicity, $t_{i,j}()$ and $e_{i,j}()$ are noted as $t_{i,j}$ and $e_{i,j}$ in the following.

Changing the frequency or activating a device when switching from one phase to the next can induce additional time and/or power resources. To model these, each change introduces a penalty with regards to energy ($e_{i,j\to i+1,j'}$) and time ($t_{i,j\to i+1,j'}$).

For all nodes (available clock-tree configurations for each phase) and edges (the changes between nodes) there are some constraints:

- Per phase i exactly one node is selected. This node is called *active node* for phase i.
- There is exactly one active edge between two phases. An edge can only be active if both predecessor and successor nodes are used and, therefore, also active.
  This can also be formulated in another way: the ingoing edges for a node sum up to the amount of outgoing edges, which splits up into:
  - A node is only active if it has an active ingoing edge, and
  - can only have an active outgoing edge if it is active.
- The time used by all active nodes and edges should be smaller than a previously known, constant hyperperiod. All remaining time is spent in sleep mode.

To formulate these constraints mathematically, binary selection variables are introduced:

- $n_{i,j} \in \{0,1\}$ selects a configuration for a phase, together with its power consumption.
- $n_{i,j\to i+1,j'} \in \{0,1\}$ represents an active change from the j-th state of phase i to the j'-th state of phase $i+1$. For the connections to S and E there are the variables $n_{s\to 0,j} \in \{0,1\}$ and $n_{(x-1),j\to e} \in \{0,1\}$.

With these variables, the constraints from above can be expressed as follows:

- Each phase can only have exactly one active node:

$$\forall i \in \{0,\dots,x-1\} : \sum_{j=0}^{f_i-1} n_{i,j} = 1 \tag{4.1}$$

- The amount of active ingoing edges equals the amount of active outgoing edges, and edges can only be in- or outgoing if the node itself is active:

$$\forall i \in \{0,\dots,x-1\}.\ \forall j \in \{0,\dots,f_i-1\}. \quad \sum_{j'=0}^{f_{i-1}-1} n_{i-1,j'\to i,j} = n_{i,j} \tag{4.2}$$

$$\forall i \in \{0,\dots,x-1\}.\ \forall j \in \{0,\dots,f_i-1\}. \quad n_{i,j} = \sum_{j'=0}^{f_{i+1}-1} n_{i,j\to i+1,j'} \tag{4.3}$$

- All times needed for the phases itself plus the times to change from one phase to another plus the remaining sleep time t_{sleep} should add up to the predefined hyperperiod H.

$$\sum_{i=0}^{x-1}\sum_{j=0}^{f_i-1} t_{i,j} + \sum_{i=0}^{x-2}\sum_{j=0}^{f_i-1}\sum_{j'=0}^{f_{i+1}-1} n_{i,j\to i+1,j'}\, t_{i,j\to i+1,j'} + \sum_{i=0}^{f_1-1} t_{s\to 0,j} + \sum_{i=0}^{f_x-1} t_{(x-1),j\to e} + t_{sleep} = H \quad (4.4)$$

- If a change from phase i with setting j to phase $i+1$ with setting j' is not possible, the decision variable for this state is set to zero:

$$n_{i,j\to(i+1),j'} = 0 \quad (4.5)$$

The optimisation problem is then formalised as follows:

$$\begin{aligned}
\min \quad & \sum_{i=0}^{x-1}\sum_{j=0}^{f_i-1} n_{i,j}\; e_{i,j} && (4.6)\\
+ \quad & \sum_{i=0}^{x-2}\sum_{j=0}^{f_i-1}\sum_{j'=0}^{f_{i+1}-1} n_{i,j\to i+1,j'}\; e_{i,j\to i+1,j'} && (4.7)\\
+ \quad & \sum_{j=0}^{f_0-1} n_{s\to 0,j}\; e_{s\to 0,j} && (4.8)\\
+ \quad & \sum_{j=0}^{f_{x-1}-1} n_{(x-1),j\to e}\; e_{(x-1),j\to e} && (4.9)\\
+ \quad & t_{sleep} p_{sleep} && (4.10)
\end{aligned}$$

wrt.

$$\begin{aligned}
\forall i \in \{0,\dots,x-1\}: \quad & \sum_{j=0}^{f_i-1} n_{i,j} = 1 && (4.11)\\
\forall i \in \{0,\dots,x-1\}: \forall j \in \{0,\dots,f_i-1\}: \quad & \sum_{j'=0}^{f_{i-1}-1} n_{i-1,j'\to i,j} = n_{i,j} && (4.12)\\
\forall i \in \{0,\dots,x-1\}: \forall j \in \{0,\dots,f_i-1\}: \quad & n_{i,j} = \sum_{j'=0}^{f_{i+1}-1} n_{i,j\to i+1,j'} && (4.13)\\
& \sum_{i=0}^{x-1}\sum_{j=0}^{f_i-1} t_{i,j} + \sum_{i=0}^{x-2}\sum_{j=0}^{f_i-1}\sum_{j'=0}^{f_{i+1}-1} n_{i,j\to i+1,j'}\; t_{i,j\to i+1,j'} \\
& \quad + \sum_{j=0}^{f_0-1} t_{s\to 0,j} + \sum_{j=0}^{f_{x-1}-1} t_{(x-1),j\to e} + t_{sleep} = H && (4.14)
\end{aligned}$$

This is an ILP, as all values except for the n_* and t_{sleep} variables are known beforehand, or are calculated as functions from the known variables for frequency, cycle count, and power consumption.

4.2 Modelling Multiple Sleep Options

Until here, the time t_{sleep} spent sleeping from E to S is the remaining time which is not spent with either executing a phase or switching from one phase to another. With this precondition, this time spent in sleep mode is not fixed beforehand like the time needed when executing a phase with a specific frequency. However, a fixed value can be determined when the solver determined the selection variables for the ILP. To model more comprehensive problems, idling has more options to "go to sleep" than just one, as real chips have more modes, for example:

- **Deep Sleep:** In this state, nearly everything on a microcontroller is powered down. As a consequence, the power consumption of it drops low. For example, the ESP32-C3 needs 5 µA in deep sleep compared to 20 mA during normal operation without peripherals [11]. The disadvantage of this sleep mode is that the penalty for going there and waking up is higher than in the following sleep options. For the ESP32-C3, it takes 300 ms to wake up (see Section 7.1.5.4 for more details).
- **Light Sleep:** In this state, the CPU and other parts of the chip are powered down, but the system can be woken up faster than from deep sleep. The downside is that this state consumes more energy than deep sleep.
- **Idling:** Switching between different CPU frequencies is very fast, e.g., for the ESP32-C3 it takes up to 21 CPU cycles. But, as the CPU stays powered on, power consumption is usually higher than in sleep mode. This can be the only way if turning the CPU off, like in light or deep sleep, takes too much time. Also, every available clock-tree configuration can be an option for idling.

As just mentioned, the time needed to enter or leave a mode varies for every option. Therefore, t_{sleep} depends on what sleep mode is selected. To model this, the S and E phases are merged into one phase, which contains all possible sleep modes. With this, the resulting graph, shown in Figure 4.2, is not acyclic anymore, as the start state has to be the same as the end state - the chosen sleep state.

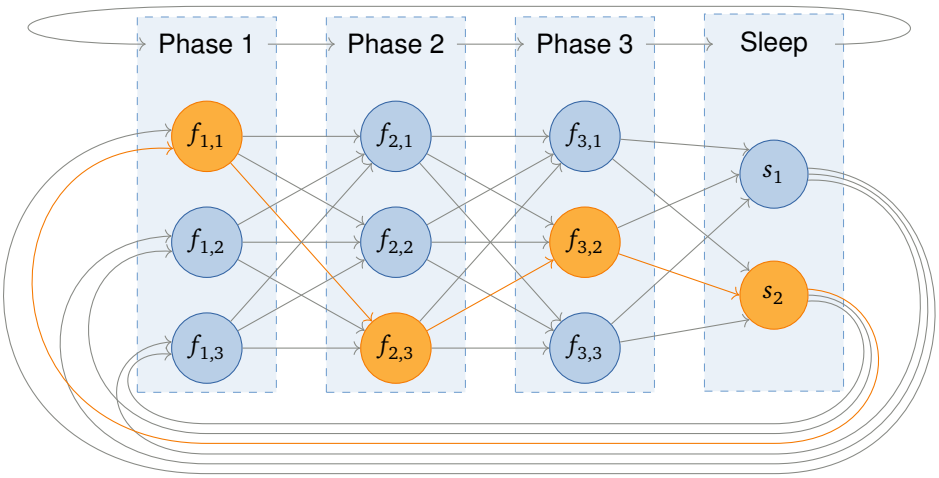


Figure 4.2 – Graph representation with multiple sleep modes: The S and E states are merged into a new phase, which contains all possible sleep modes. In this case, there are two possible sleep modes.

To describe this graph, new variables are introduced:

- The amount of sleep modes f_{sleep}. The sleep phase is added to the x phases, therefore, $f_{sleep} = f_x$.
- New binary selection variables $\forall i \in \{0, \dots, f_{sleep} - 1\}.\ n_{sleep,i}$ to select the sleep mode.
- The power consumptions $\forall i \in \{0, \dots, f_{sleep} - 1\}.\ p_{sleep,i}$ of all sleep modes.
- Energy and time costs for changing from the last phase to sleep modes, $\forall i \in \{0, \dots, f_{x-1} - 1\}. \forall j \in \{0, \dots, f_{sleep} - 1\}.\ e_{x-1,i \to sleep,j}, t_{x-1,i \to sleep,j}$, and for changing from sleep to the first phase, $\forall i \in \{0, \dots, f_{sleep} - 1\}. \forall j \in \{0, \dots, f_0 - 1\}.\ e_{sleep,i \to 0,j}, t_{sleep,i \to 0,j}$.
- The corresponding binary variables for the changes $\forall i \in \{0, \dots, f_{x-1} - 1\}. \forall j \in \{0, \dots, f_{sleep} - 1\}.\ n_{x-1,i \to sleep,j}$ and $\forall i \in \{0, \dots, f_{sleep} - 1\}. \forall j \in \{0, \dots, f_0 - 1\}.\ n_{sleep,i \to 0,j}$.
- For the remaining time in sleep depending on the selected sleep mode, the variables $t_{sleep,i}$. These are - as the binary variables for selecting the states per phase - not known beforehand.

This results in the following minimisation problem:

$$\min \quad \sum_{i=0}^{x-1} \sum_{j=0}^{f_i - 1} n_{i,j}\ e_{i,j} \tag{4.15}$$

$$+ \quad \sum_{j=0}^{f_{sleep} - 1} n_{sleep,j}\ t_{sleep,j}\ p_{sleep,j} \tag{4.16}$$

$$+ \quad \sum_{i=0}^{x-1} \sum_{j=0}^{f_i - 1} \sum_{j'=0}^{f_{(i+1) \bmod (x+1)} - 1} n_{i,j \to (i+1) \bmod (x+1),j'}\ e_{i,j \to (i+1) \bmod (x+1),j'} \tag{4.17}$$

wrt.

$$\forall i \in \{0, \dots, x\}: \quad \sum_{j=0}^{f_i - 1} n_{i,j} = 1 \tag{4.18}$$

$$\forall i \in \{0, \dots, x\}: \forall j \in \{0, \dots, f_i - 1\}: \quad \sum_{j'=0}^{f_{(i-1) \bmod x} - 1} n_{(i-1) \bmod x,j' \to i,j} = n_{i,j} \tag{4.19}$$

$$\forall i \in \{0, \dots, x\}: \forall j \in \{0, \dots, f_i - 1\}: \quad \sum_{j'=0}^{f_{(i+1) \bmod x} - 1} n_{i,j \to (i+1) \bmod x,j'} = n_{i,j} \tag{4.20}$$

$$\sum_{i=0}^{x-1} \sum_{j=0}^{f_i - 1} n_{i,j}\ t_{i,j} \ + \ \sum_{j=0}^{f_{sleep} - 1} n_{sleep,j}\ t_{sleep,j}$$

$$+ \quad \sum_{i=0}^{x} \sum_{j=0}^{f_i - 1} \sum_{j'=1}^{f_{(i+1) \bmod x} - 1} n_{i,j \to (i+1) \bmod x,j'}\ t_{i,j \to (i+1) \bmod x,j'} = H \tag{4.21}$$

This formulation is not linear anymore, as the objective function contains the terms

$$\sum_{j=0}^{f_{sleep}-1} n_{sleep,j}\ t_{sleep,j}\ p_{sleep,j} \tag{4.22}$$

and

$$\sum_{j=0}^{f_{sleep}-1} n_{sleep,j}\ t_{sleep,j}. \tag{4.23}$$

In both equations, $n_{sleep,j}$ and $t_{sleep,j}$ are variables that are unknown and shall be determined by the solver. But, as they are multiplied, the linear program turns into a Quadratic Program (QP) and is not a valid input to any linear-program solver. To circumvent this problem, another solver might be employed, which is also able to solve non-linear problems. One example of this is Gurobi [14]. If such a solver is not available, one can generate a separate optimisation problem for each sleep state, as introduced in Section 4.1. First, every one of these problems is solved, and then the minimum value from all runs and the corresponding solution is selected in an additional step.

This results in the following problem statement for an ILP solver:
Select the minimum value for the problem described by varying $Z \in \{0, \ldots, f_{sleep} - 1\}$:

$$\min \sum_{i=0}^{x-1}\sum_{j=0}^{f_i-1} n_{i,j}\ e_{i,j} \tag{4.24}$$

$$+ \quad t_{sleep,Z}\ p_{sleep,Z} \tag{4.25}$$

$$+ \sum_{i=0}^{x-1}\sum_{j=0}^{f_i-1}\sum_{j'=0}^{f_{(i+1) \bmod (x+1)}-1} n_{i,j\to(i+1) \bmod (x+1),j'}\ e_{i,j\to(i+1) \bmod (x+1),j'} \tag{4.26}$$

wrt.

$$\forall i \in \{0, \ldots, x-1\}: \quad \sum_{j=0}^{f_i-1} n_{i,j} = 1 \tag{4.27}$$

$$n_{sleep,Z} = 1 \tag{4.28}$$

$$\forall j \in \{0, \ldots, f_{sleep}-1\}\backslash\{Z\}: \quad n_{sleep,j} = 0 \tag{4.29}$$

$$\forall i \in \{0, \ldots, x\}: \forall j \in \{0, \ldots, f_i - 1\}: \quad \sum_{j'=0}^{f_{(i-1) \bmod x}-1} n_{(i-1) \bmod x,j'\to i,j} = n_{i,j} \tag{4.30}$$

$$\forall i \in \{0, \ldots, x\}: \forall j \in \{0, \ldots, f_i - 1\}: \quad \sum_{j'=0}^{f_{(i+1) \bmod x}-1} n_{i,j\to(i+1) \bmod x,j'} = n_{i,j} \tag{4.31}$$

$$\sum_{i=0}^{x-1}\sum_{j=0}^{f_i-1} n_{i,j}\ t_{i,j} \ + \ t_{sleep,Z} \ +$$

$$+ \sum_{i=0}^{x}\sum_{j=0}^{f_i-1}\sum_{j'=0}^{f_{(i+1) \bmod x}-1} n_{i,j\to(i+1) \bmod x,j'}\ t_{i,j\to(i+1) \bmod x,j'} = H \tag{4.32}$$

When the chosen mathematical solver supports non-linear problems, both problem descriptions shall determine the same solution. By extracting the selected binary variables, a power-consumption–optimal program with real-time guarantees can be developed. It depends on the performance of the mathematical solver whether solving f_{sleep} linear problems for each sleep state or one quadratic problem is faster. This topic is analysed in the next chapter.

5 EVALUATION OF THE SOLVER PERFORMANCE

The problem description from the previous chapter has multiple parameters. Three of these have a definite influence on runtime: the number of phases, the number of available clock-tree configurations per phase, and the number of idle or sleep modes. Other parameters are important for determining the solution of real inputs but have no influence on the problem complexity itself. Therefore, they are left out of the following problem-size analysis. These are the number of cycles per phase and all costs for the configurations per phase or the costs for changing from one configuration to another in terms of time and power consumption.

First, Section 5.1 takes a look at the complexity of the problem with regards to the input. After that, it is to be tested whether the mathematical model description from Chapter 4 is efficient enough for the intended use case. Therefore, input sets for the solver are generated automatically, and the solver performance is evaluated in Section 5.2. Finally, based on those two insights, a conclusion is made in Section 5.3.

5.1 Theoretical Problem Complexity

Let $x + 1$ be the amount of phases x plus the sleep phase, and n the maximum number of possible frequencies per phase, including the sleep phase. The parts influencing the model complexity are:

1. **The amount of binary and integer variables:** A solver must determine a solution for each unknown variable. A more extensive search space means more options and combinations it has to try to reach the optimal solution.

2. **The length and complexity of the objective function:** If the number of summands in the objective function increases, the solver has more calculation work when checking the objective term. Also, more complex terms (linear versus quadratic) mean more calculation operations.

3. **The length and complexity of the constraints.**

The actual complexity depends heavily on the chosen solver and its optimisation strategies. Even though a general statement is complex, the problem has several externally quantifiable parameters. One could expect influences of these parameters on the problem complexity due to the comparatively uniform structure of flow problems.

Both models from Section 4.2 are tested.

- First, the model with all sleep options in one single optimisation problem is discussed:

  1. **Amount of variables:**
     There are binary selection variables for each phase and frequency option, which results in $(x+1) \cdot n$ binary variables. Between each layer, the maximum of connections is given when every node from the current phase is connected to every other node from the next phase. A binary variable for each connection controls the selection of these connections, therefore $(x+1) \cdot n^2$ binary variables. The remaining time spent in a sleep mode is modelled with an integer variable per sleep mode, which are n linear variables.
     In total, there are $(n+1) \cdot ((x+1) \cdot n)$ binary and n linear variables.
  2. **Complexity of the objective function:**
     Equation 4.15 has $x \cdot n$ linear terms, Equation 4.16 has n quadratic terms, and Equation 4.17 is made of $(x+1) \cdot n^2$ linear terms.
  3. **Complexity of the constraints:**
     Equation 4.18 has $(x+1)$ linear terms with n summands each, Equation 4.19 and Equation 4.20 $(x+1) \cdot n$ linear terms, each with n summands per term. Equation 4.21 is made of $(x+1) \cdot n + x \cdot n^2$ linear summands, and $n + n^2$ quadratic ones.

  Summing up: The problem size of both linear and quadratic terms grows in a quadratic manner for the number of nodes per phase and linear for the number of phases. It is important to point out that the quadratic terms are either made from two binary selection variables or one real number and a binary selection variable.

- Next, just one sleep mode per input to the solver is assessed. The quadratic terms disappear and create an ILP, which corresponds to the following complexity:

  1. **Amount of variables:**
     The amount of variables stays the same, as just the objective function and the constraints are adapted to provide a valid ILP description.
  2. **Complexity of the objective function:**
     Equation 4.24 has $x \cdot n$ linear terms, Equation 4.25 has one linear term, and Equation 4.26 is made of $(x+1) \cdot n^2$ linear terms.
  3. **Complexity of the constraints:**
     Equation 4.27 has x linear terms with n summands each, Equation 4.28 and Equation 4.29 together are n linear terms. Equation 4.30 and Equation 4.31 are both $(x+1) \cdot n$ terms with n summands, and Equation 4.32 is made of $(x+1) \cdot n + 1 + (x+1) \cdot n^2$ linear summands.

  Here, the problem size also grows quadratically for the number of nodes per phase and linear in the number of phases, but only for linear terms instead of quadratic ones. However, the quadratic factor is reintroduced as the problem has to be executed n times.

To check the solver efficiency for larger problems, the influence of the number of phases and the influence of frequency options per phase are tested. Extrapolating from the notion of problem size as described above, a linear growth for the number of phases and a quadratic growth for the configuration options per phase are to be expected.

5.2 Solver Efficiency

The first step before evaluating a solver's performance is to identify suitable solvers for the given problem. Two popular options are lpsolve [21] or Gurobi [14]. While lpsolve is an open-source tool, Gurobi is a licensed mathematical solver which distributes free copies of it for use in research. Because Gurobi is closed-source software, it is neither possible to identify or reconstruct which solution approach the solver takes nor which parameters directly influence the problem complexity. As the name suggests, lpsolve can only solve linear problems, while Gurobi is also capable of more complex terms such as quadratic optimisation problems.

All tests were performed on a computer with an Intel(R) Core(TM) i5-4590 CPU and 16 GB of RAM, running Debian with a Linux kernel, version 5.10. lpsolve is installed in version `5.5.2.5-2`, Gurobi in version `9.5.1` with an academic license. Throughout the evaluation, Gurobi outperformed lpsolve, e.g., solving a program that took Gurobi milliseconds was aborted for lpsolve after 10 minutes. Therefore, the performance tests were run with Gurobi.

To have a look how the theoretical complexity inspected in Section 5.1 maps to real performance, time measurements for varying problem sizes are performed. This thesis assumes that a range from five to 80 phases and from five to 40 configurations per phase is a reasonable choice for hard real-time systems in respect of a real-time task with a limited task size and a microcontroller with a limited clock-tree size and, therefore, limited configuration options. A python [43] script generates input files for Gurobi with random values. The random number generator uses a fixed seed for reproducibility reasons. After generation, the solver is called with the input file, and its total time usage is measured.

First, Section 5.2.1 discusses the performance of both variants detailed in Chapter 4. Afterwards, Section 5.2.2 compares the performance of the solver when increasing the problem sizes with regards to the number of phases as well as the number of configuration options per phase for approaches.

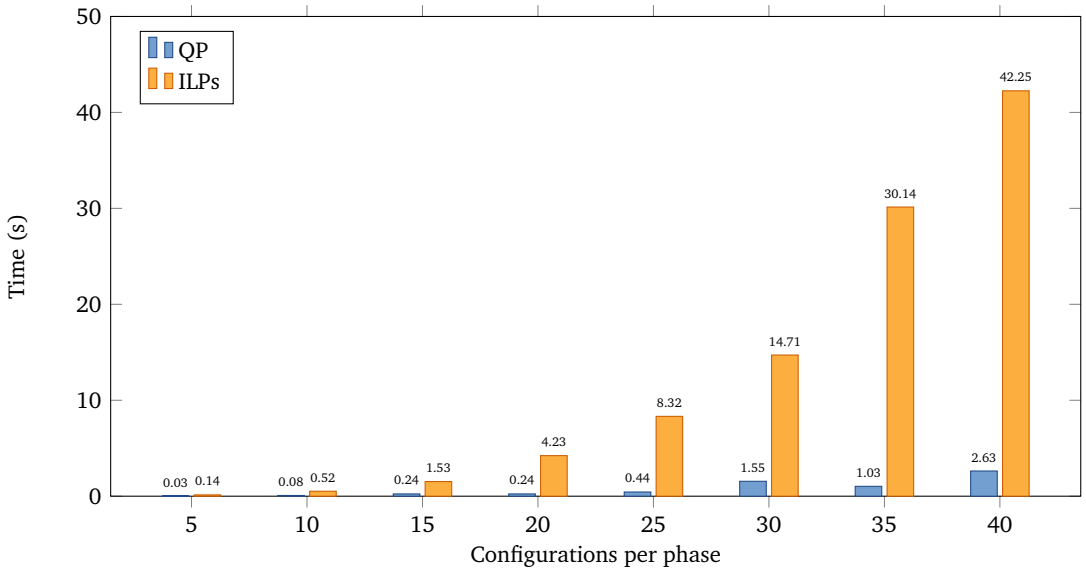


Figure 5.1 – Comparing the solver performance of the QP and the ILPs for a fixed number of 25 phases.

5.2.1 Integer Linear Program versus Quadratic Program

Chapter 4 provides two approaches for handling multiple sleep options: Either all sleep modes are handled in one optimisation problem, which then is a QP and requires a powerful solver who is able to handle such problems, or an ILP is generated for each sleep mode, which can be handled by a ILP solver but has to run multiple times, once for each sleep mode, instead of one time.

As the chosen solver, Gurobi, can handle both, this section compares both approaches. The solution for both approaches yields the same optimum for the same settings. The use of floating-point numbers of the solver leads to slight deviations of the optimum value, but these did not influence the optimal configuration choices. When comparing the time of the QP and the multiple ILPs, the first one outperforms the second one. Figure 5.1 displays this for an increasing number of configurations per phase for 25 phases: The time needed to solve the given problem(s) grows for both inputs, but for all values, the QP is faster. In Figure 5.2, all measured values are compared by dividing the time required to solve the QP by the time needed to solve all ILPs. If the ratio is smaller than 1, the QP is faster than the ILPs, otherwise, the ILPs outperform the QP. The second case never occurs in this test. The performance gets even worse in comparison to the single QP if there are more ILPs to solve, as the number of configurations per phase and therefore sleep phases increases, and the ILPs get more complex, as the number of phases increases as well as the number of ILPs to solve. In summary, although the time needed to solve several ILPs is not comparable to the time needed to solve a single QP in terms of usability, both methods return the same solution for the given problem.

5.2.2 Problem-Size–Scaling Behaviour of the Solver

Although the QP formulation is solved much faster than the approach with ILPs, both approaches are evaluated in this section separately. Figure 5.3 shows the total time needed to solve the problem for the ILP formulation for all tested input sizes. Figure 5.4 displays the same for the QP formulation.

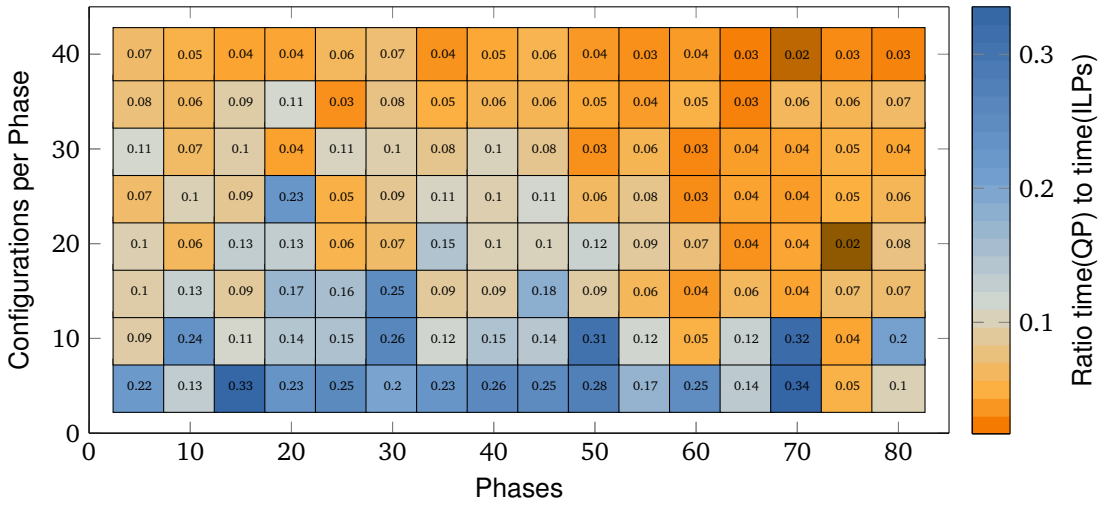


Figure 5.2 – Heatmap showing the ratio between the time taken to solve the QP and the time taken to solve all ILPs for all measured combinations of phases and configurations per phase. If the ratio is smaller than 1, the QP is faster than the ILPs, otherwise, the ILPs outperform the QP. For all tested values, the QP was faster: The worst performance of the QP was 34% of the time of the ILPs.

When increasing the number of phases or the number of configurations per phase, the time needed to solve the problem increases - as expected. Figure 5.5, Figure 5.6, Figure 5.7, and Figure 5.8 display the same numbers in different manners: To visualize the growth rate, each value is divided by the value for the highest input number. All these graphics also show that increasing the number of phases or the number of configurations per phase leads to an increased solver runtime.

A slight tendency towards a quadratically increasing runtime of the solver can be seen when scaling the configuration options per phase. Nevertheless, there is another clear insight: the maximum time required to solve all ILPs in the current randomized measurement setup is 959.63 s. The QP only takes a maximum of 37.28 s. This means that the mathematical description is sufficiently good that Gurobi can solve the given problems in an acceptable time for static analysis.

5.3 Viability of the Problem Description

Summarising the results from Section 5.2.2 and Section 5.2.1, the result is unambiguous: Both mathematical models can be solved sufficiently efficiently by the chosen solver, Gurobi, for the expected input range. The test results indicate that also larger problems may be solved in a reasonable time. When comparing both implementations of the same underlying problem, the nonlinear approach yields faster solver runtimes than the multiple but linear programs. Since Gurobi is not open-source software, one cannot prove the exact reasons for this. One possible factor is that the quadratic formulation is faster as the problem description only has to be read once. This includes all constraints, which are equally many for both formulations, but they have to be read n times instead of once for the linear problems. Overall, the overhead of Gurobi to solve the mathematically more complex quadratic problem is still less than solving the additional linear problems. Therefore, the suggestion arising from the results of this section is to use the quadratic problem formulation to find the optimal configuration.

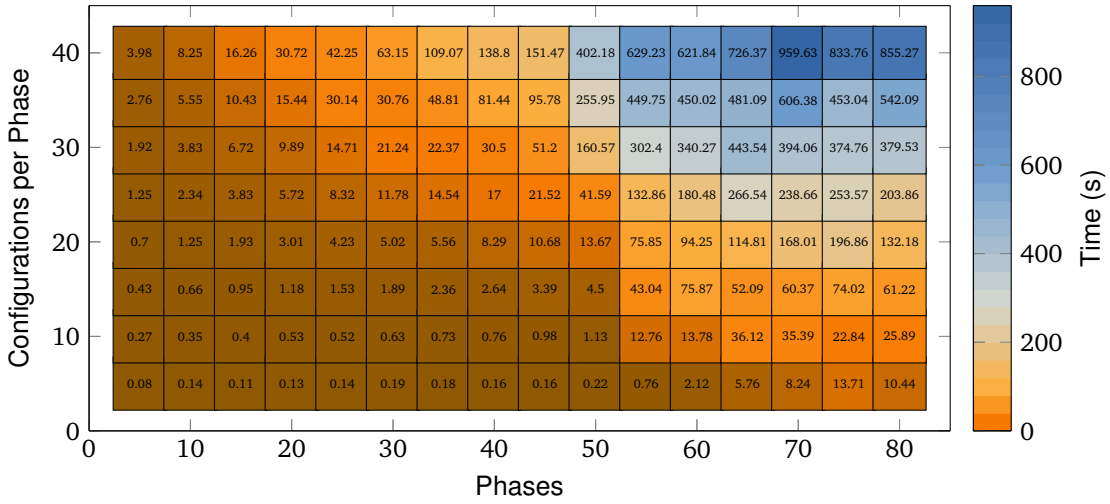


Figure 5.3 – Time of sum of all ILPs for each configuration: With increasing numbers of phases and configurations per phase, the time to solve the given problem increases.

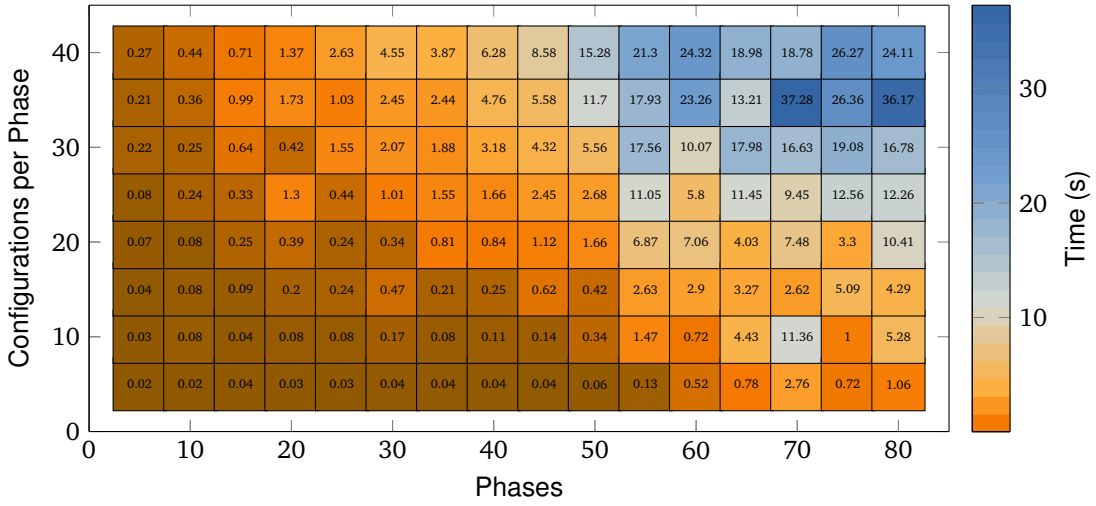


Figure 5.4 – Time of QP for each configuration: With increasing numbers of phases and configurations per phase, the time to solve the given problem increases.

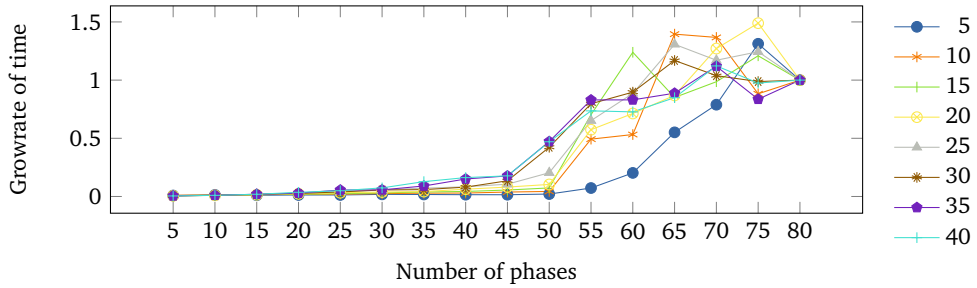


Figure 5.5 – Ratio of time needed by maximum of 80 phases by the current amount of configurations per phase to the current amount of phases by the current amount of configurations per phase, for sum of time needed by ILPs.

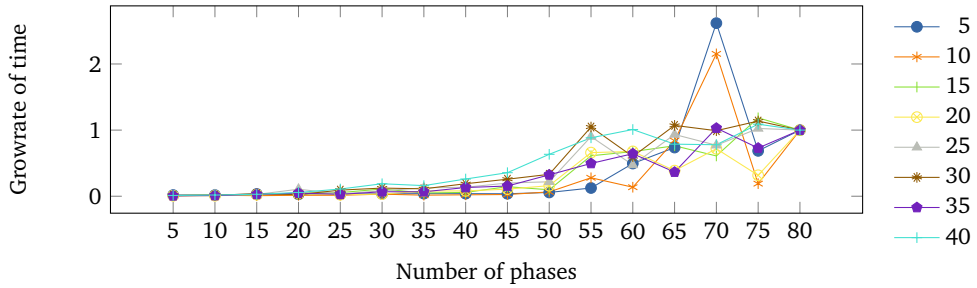


Figure 5.6 – Ratio of time needed by maximum of 80 phases by the current amount of configurations per phase to the current amount of phases by the current amount of configurations per phase, for time needed by QP.

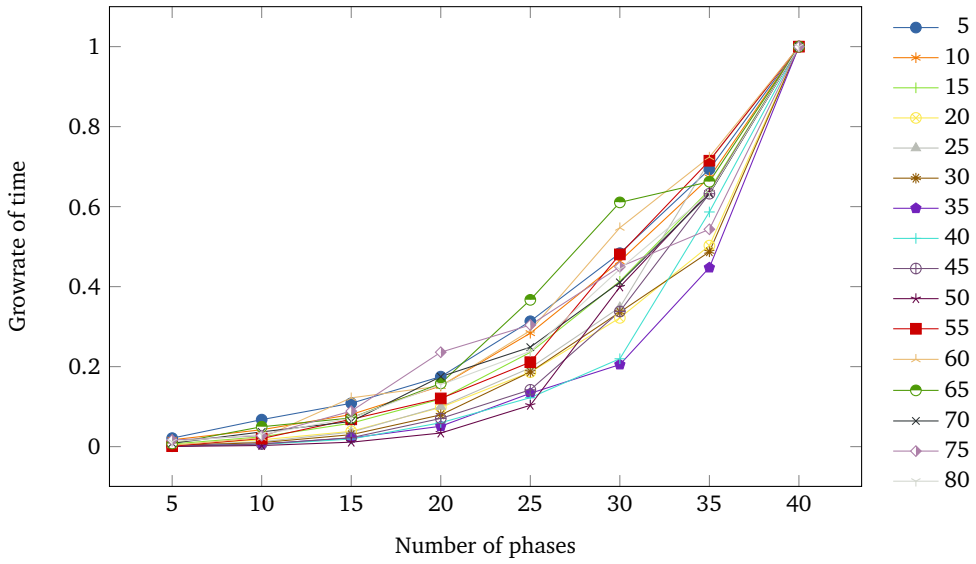


Figure 5.7 – Ratio of time needed by maximum of 40 configurations per phase by the current amount of phases to the current amount of configurations per phase by the current amount of phases, for sum of time needed by ILPs.

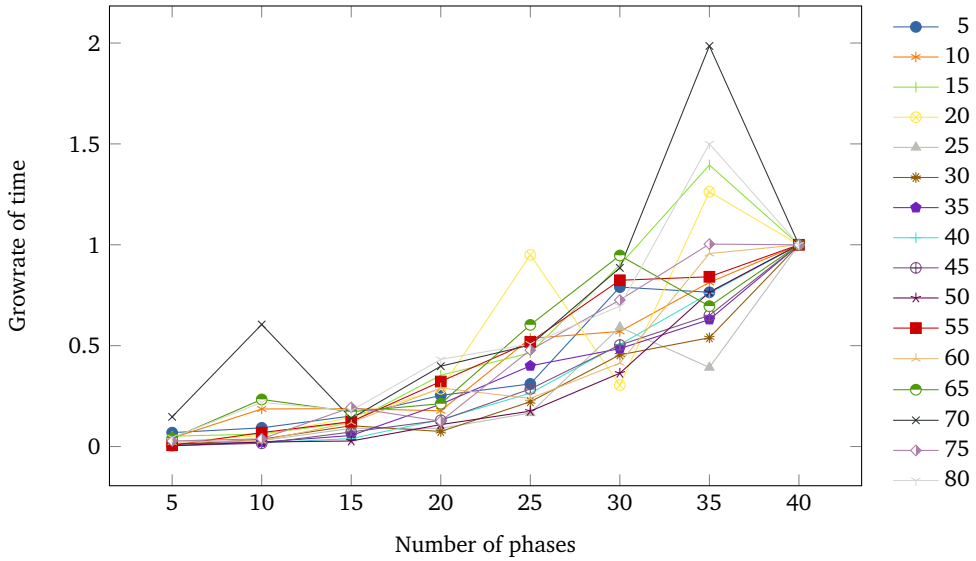


Figure 5.8 – Ratio of time needed by maximum of 40 configurations per phase by the current amount of phases to the current amount of configurations per phase by the current amount of phases, for time needed by QP.

6 IMPLEMENTATION FOR THE ESP32-C3

This chapter covers the implementation details necessary to use the approach presented in Chapter 4 on real hardware. The chosen hardware platform, which provides many different power modes, allows the configuration of these and features many peripherals, is the ESP32-C3-MINI-1 module [11]. Section 6.1 dives deeper into the ESP32-C3. It describes the hardware and software details with regards to determining the WCET of the ESP32-C3 and to making use of the given power modes. Then, Section 6.2 explains how the measurements of the time needed by single instructions and all side effects are acquired and integrated into PLATIN to determine the WCET of a program.

6.1 The ESP32-C3

A brief overview of the hardware platform of the ESP32-C3 is given in Section 6.1.1. Section 6.1.2 continues with the software and development environment. As power usage and, therefore, power management is a key point in this work, Section 6.1.3 details the available clocks for the CPU and their connections to the peripherals. Section 6.1.4 provides an insight into the available power modes of the ESP32-C3.

6.1.1 Hardware Overview

The chosen ESP32-C3-MINI-1 module contains a 32-bit RISC-V ESP32-C3 single-core microprocessor (ESP32C3-C3FH4 or ESP32C3-C3FN4) with an RV32IMC Instruction Set Architecture (ISA), which runs with a frequency up to 160 MHz. Many peripherals are included, such as WiFi, Bluetooth Low Energy, a Serial Peripheral Interface (SPI), UART, an USB Serial/JTAG controller, a temperature sensor and many more [11, Chapter 1].

6.1.1.1 Chip Details

The RV32IMC ISA [38] decodes into the following components of the RISC-V instruction set: RV32I determines the instruction size, which is 32 bits, and provides the base set of available instructions. The M standard extension adds multiplication and division instructions. The C standard extension includes compressed instructions: To reduce static and dynamic code size, common operations get 16-bit instruction encodings in addition to the 32-bit ones. It also softens the alignment constraints of instructions from 32-bit to 16-bit, so 16- and 32-bit instructions can be mixed freely to reduce the program's memory usage.

The ESP32-C3 has a 4-stage, in-order, scalar pipeline. The documentation does not list additional information about the branch prediction unit of the ESP32-C3, therefore each branching instruction has to assume the worst-case scenario of a failed prediction and a resulting pipeline flush.

6.1.1.2 Memory

The module includes 384 kB of Read-Only Memory (ROM) and 400 kB of Internal Static Random-Access Memory (SRAM), of which 16 kB can be used as a cache for 4 MB of an on-board flash. The CPU can access data via the data bus with single-, double- and 4-byte alignment, while it can access the instruction bus only in a 4-byte aligned manner [10, Chapter 3.3.1]. The SRAM can be accessed by the CPU generally within a single CPU clock cycle [10, Chapter 3.3.2]. In combination with the relaxed alignment constraints of the C RISC-V extension from 32-bit/4-byte to 16-bit/2-byte, this means that in special cases two accesses must be made to the instruction bus to fetch one instruction. Section 6.2.2 addresses this problem in more detail.

6.1.2 Espressif IoT Development Framework

The combination of the software development environment and the programming framework for the family of Espressif microcontrollers is the Espressif IoT Development Framework (ESP-IDF). It provides toolchains for all Espressif microcontrollers and workflows to develop applications. That includes boot-up code for each device, building applications, flashing, JTAG debugging, and support for monitoring running applications for Windows, Linux, and macOS operating systems. The programming reference includes an Application Programming Interface (API) for applications, e.g., an HTTP(S) server, a networking API, e.g., for WiFi and Bluetooth connections, a peripherals API, e.g., for controlling General Purpose Input/Output (GPIO) pins or setting up and using SPI, a system API for configuring system timers or set up power management, and more. All this and additional information is found in the documentation for the ESP32-C3 [8].

As stated in Section 6.1.1.2, the chip has 16 kB of SRAM, which can be used as cache for 4 MB of on-board flash. As mentioned in Section 2.2.5, creating a proper cache model can be tedious, especially if there is no documentation of what cache replacement strategy is used. As the memory usage of all tested applications is smaller than the size of the SRAM, this flash is not used, and all needed data and instructions are stored in the single-cycle accessible SRAM. In addition, the resulting hardware model will deliver more accurate runtime estimations as this additional level of inaccuracy and overestimation is left out. To achieve this, the linker is told to place all application and operating system data into the SRAM of the chip. ESP-IDF supports this with so-called linker fragment files. An example is shown in Listing 6.1.

```
1   [mapping:foo]
2   archive: libfoo.a
3   entries:
4     * (noflash)
```

Listing 6.1 – The linker fragment file to place the library called `libfoo.a` into the `noflash` area, also known as the SRAM.

6.1.3 Clock Tree

The ESP32-C3-MINI-1 board has many options regarding its clock and device configuration, making it interesting for the intended research. The clock tree, presented in the documentation [10, Figure 6-2], shows how the input clocks connect to the CPU and the peripherals. In between, dividers, multiplexers, and peripheral gates take care of which signal is passed through the tree. The ESP32-C3 has five main input clocks:

- two fast clocks:
  - `PLL_CLK`: the internal PLL clock, which runs at 320 MHz or 480 MHz.
  - `XTAL_CLK`: a crystal clock, which provides a frequency of 40 MHz.
- three slow clocks:
  - `XTAL32K_CLK`: a crystal clock at 32 kHz.
  - `FOSC_CLK`: an internal fast RC oscillator at 17.5 MHz.
  - `RTC_CLK`: an internal low RC oscillator at 136 kHz.

The CPU clock uses three of these as source clocks: `PLL_CLK`, `XTAL_CLK` or `FOSC_CLK`. For `PLL_CLK`, one has the option to choose which base clock is used (320 MHz or 480 MHz) and whether the resulting CPU clock is set to 160 MHz or 80 MHz by dividing the base clock accordingly. When using `XTAL_CLK` or `FOSC_CLK` as CPU clock source, a divider can be set to values from 1 to 1024, which divides the chosen input clock with the chosen value. This corresponds to a full CPU clock range from 17.09 kHz up to 160 MHz, so high computational power and energy savings can be achieved by setting the CPU frequency accordingly. Other clocks, namely `APB_CLK`, `LEDC_SCLK` and `CRYPTO_CLK`, depend on the choice of the `CPU_CLK`. Also, WiFi and Bluetooth can only operate when `PLL_CLK` is used as CPU clock source, but also have a low power mode for operating with the low power clocks. For all other peripherals, the documentation lists which clock sources can be used [10, Table 6-4]. For additional power savings the peripheral devices can also be clock-gated and therefore deactivated and cut off the clock tree completely. All options can be configured in the system registers of the ESP32-C3 by storing the necessary configuration values there, or the current configuration can be read from them.

6.1.4 Power Modes

A main criterion for choosing the ESP32-C3 was its many different peripheral options and power modes [10, Chapter 9]. The ESP32-C3 has nine power domains, which correspond to subparts of the chip, such as the CPU, peripherals, or RTC. They are set on or off in four predefined power modes: active, modem sleep, light sleep, and deep sleep. In the active state, all power domains are turned on. In modem-sleep, the radio-frequency circuits get switched off. In light sleep, the CPU and its clock sources (as seen in Section 6.1.3) are powered off, and in deep sleep also the digital system gets turned off so that only the PMU and a small set of peripheral devices stay turned on. To leave that mode later when sleeping, one has to define at least one wake-up source before that. Multiple options are available for waking up the CPU from light sleep: GPIOs, the RTC timer, the 32 kHz crystal clock as a timer, UART, Bluetooth, or WiFi. For deep sleep, only the RTC GPIOs (0-5), the RTC timer, and the 32 kHz crystal clock can wake up the core. When leaving light sleep, the CPU will resume operation after the instruction where it went into sleep, as the internal states of the digital peripherals, the RAM, and the CPU are preserved. For deep sleep, a complete reboot occurs:

Only the RTC controller, the RTC peripherals, and the RTC fast memory remain powered on during deep sleep. As a consequence, CPU, RAM, and all other peripherals are restarted when the wake-up occurs.

According to [11], the chip's power consumption can reach up to 350 mA with active radio frequency devices. Without these, 15 mA (with the CPU clock at 80 MHz) to 20 mA (with the CPU clock at 160 MHz) are used in modem sleep. It gets lower when using the sleep modes: In light sleep, the board consumes 130 µA. Deep sleep needs 5 µA, which equals 0.025 per cent of run mode. This shows the huge potential of the chip to operate very energy-efficient when the computational power or the integrated peripheral devices are not needed. Own measurements in Section 7.1 will evaluate these values.

6.2 PLATIN

To analyse the WCET of a program for the ESP32-C3, the Portable LLVM Annotation and Timing (PLATIN) toolkit, developed by Hepp et al. [28] for analysing the PATMOS architecture [36], was chosen as a basis for this thesis. PLATIN centres around the LLVM compiler infrastructure for compiling and analysing applications. The intermediate representation of the LLVM toolchain is used to create PLATIN's native `PML` file format to store information about the program structure, analysis results, and other meta-information in a target-machine–agnostic form for analysis [28]. It provides interfaces to external analysis tools, such as lpsolve [21], Gurobi [14] or the SWEdish Execution Time tool (SWEET) [42], and tools for flow-fact transformation or IPET-based worst-case analysis.

6.2.1 Toolchain Setup for the ESP32-C3

Hofmeier [16] already implemented a PLATIN integration for a SiFive E31 RISC-V Core on a SiFive HiFive1 development kit with a 32-bit RV32IMAC ISA. A similar toolchain for generating the `PML` files for PLATIN is used in this thesis: LLVM and its compiler frontend `clang` at version 7 support RISC-V. An extension to the backend to transform the control-flow information into `PML` files was written at the Department of Computer Science 4 at Friedrich-Alexander–University Erlangen-Nuremberg [28, 35, 36]. For integrating LLVM into the ESP-IDF build chain, `clang` compiles the application to receive the `PML` file. A library is created from the object files, which is then loaded into SRAM via a linker fragment file, as shown in Listing 6.1. The main task of the ESP-IDF infrastructure directly calls the main function of the application. As a consequence, the remaining build, flash, and boot infrastructure stay the same, but, in addition, the application can be analysed with PLATIN.

6.2.2 Timing Behaviour of the ESP32-C3

As a basis for this analysis, an architecture-specific model is needed. It specifies the timing and hardware model of the chosen hardware. To determine the WCET, PLATIN determines the cost of a path in the CFG by summing up the costs of each opcode in the control-flow representation in the `PML` file. Therefore, this thesis needs to determine the maximum time needed for single opcodes. There is only one sound source of information, which is timing information written in the manual. Unfortunately, there is no sound documentation for these available of now, as none of the documentation documents [9, 10, 11] list any information. The only available possibility is to measure the value of each instruction pessimistically.

There are two options to achieve that:

- Use performance counters provided by the hardware, e.g., amount of CPU cycles.
- Measure and/or validate the timing behaviour with an oscilloscope.

The ESP32-C3 can operate at frequencies up to 160 MHz which can get cumbersome to measure precisely with an oscilloscope. However, it has a set of performance counters available. According to the RISC-V documentation [38, Chapter 10], the RISC-V ISA provides a set of 32 64-bit performance counters. Although the ESP32-C3 implements the RISC-V ISA, these performance counters are not implemented on this RISC-V chip, and the `rdcycle` instruction to read the cycle counter throws an illegal instruction exception. Another set of performance counters, which are implemented in the `MPCER`, `MPCMR` and `MPCCR` Configuration and Status Registers (CSRs) is available for the ESP32-C3 [10, Chapter 1.4]. These can be set up to count the number of occurrences of different events, such as branches, branches taken, stores, loads, idle cycles, instructions, or - relevant for this problem - the clock cycles. To have reproducible measurements interrupts are disabled by clearing the `MIE` bit (interrupt-enable bit) in the `mstatus` register before cycle counting. The assembler macro to measure an instruction and read out the performance counter is shown in Listing 6.2.

As mentioned in Section 6.1.1.2, the CPU can access the instruction bus only in a 4-byte aligned manner [10, Chapter 3.3.1]. The ESP32-C3 has the RISC-V C extension, so the alignment constraint is loosened to support 2-byte wide instructions. This results in the problem that instructions can need two accesses to the instruction bus to load one instruction. With pipelining, this is no problem as one load fetches at least one instruction. However, when jumping or branching instructions interrupt the straight-forward execution flow, it takes two loads to get a 4-byte instruction when the jump goal targets a 16-bit aligned but not 32-bit aligned address. To simulate this behaviour, the measurement infrastructure makes sure that both cases are looked after: one with the instruction aligned to a 4-byte address and one with the instruction aligned to a 2-byte address but not a 4-byte address. In all cases, the maximum time taken is entered as a measurement for PLATIN.

The `PML` file contains simple operation codes, such as `add` or `sub`. The costs for these instructions can be directly substituted with the measured values. The LLVM compiling process also generates pseudo operations such as `PseudoRET` or `PseudoBRANCH`. Depending on the address, a different set of instructions is used for these pseudo operations. As one does not know at this level to which instructions the operation will be resolved, the worst combination of the observed values is assumed to reduce the risk of underestimated WCETs.

```
1 .macro measure_perf_count counter:req ins:vararg
2 csrw MPCCR, 0
3 \ins
4 csrr \counter, MPCCR
5 .endm
```

Listing 6.2 – GNU Assembler macro for measuring an instruction, `\ins`, and read out the value of the performance counter (`MPCCR`) into `\counter`. `\ins` can also contain more complicated commands, such as multiple instructions or jump commands, as long as the control flow returns afterwards to stop the measurement by reading out the value into `counter` in line 4.

6.2.3 Platin Evaluation

To check the correctness of the determined values from the performance counter, a new architecture model was integrated into PLATIN with the measured values. This model was tested by running a set of experiments. The set of benchmarks chosen is TACLeBench [12], a benchmark collection aimed at WCET research. Each benchmark is self-contained and does not depend on standard libraries or an operating system, which makes the collection useful for embedded systems as the ESP32-C3. Additionally, all benchmarks are annotated with flow facts the analysis of PLATIN can use and have a fixed entry point in the `main` function.

For analysis, a subset of the TACLeBench benchmark suite is used: the `kernel` benchmark subset, where each benchmark computes a computational kernel. As the ESP32-C3 does not provide a floating-point unit, floating-point instructions are emulated with instruction sequences stored in the ROM of the ESP32-C3. Linker files contain the addresses in the ROM for each of these instructions. A jump table located in the ROM links these addresses to the individual implementations. Therefore, the generation of the PML files required for the analysis with PLATIN is not possible. As a consequence, all benchmarks including floating-point operations are excluded from evaluation. Furthermore, PLATIN does not offer support for analysing recursive function calls at the time of this thesis. As a consequence, all recursive benchmarks are excluded as well. For some benchmarks, lpsolve is not able to determine a solution. These are also omitted from the evaluation. The remaining benchmarks are listed and described briefly in Table 6.1.

The results of the WCET analysis from PLATIN are compared with measurements of the performance counters, just as performed for the measurements of single instructions shown in Listing 6.2. As the ESP-IDF framework brings its own bootloader and compiler for the ESP32-C3, which does not generate the necessary information for PLATIN to perform the IPET-based WCET analysis, the benchmarks are built with the extended `clang` (already mentioned in Section 6.2.1) [28, 35, 36]. Afterwards, all necessary components for the benchmark are grouped in a library, which is then linked to the startup code. A linker fragment file, as shown in Listing 6.1, forces the code to be placed in the SRAM. For each benchmark, 1000 runs were measured. Before each run, interrupts were disabled.

Table 6.1 – TACLeBench benchmarks [12, 37]

Benchmark	Description
`binarysearch`	Binary search of 15 integers. The program is completely structured (no unconditional jumps, no exits from loop bodies), and does not contain switch statements or do-while loops.
`bsort`	Bubblesort implementation for testing the basic loop constructs, integer comparisons, and simple array handling by sorting 100 integers
`countnegative`	Counts negative and non-negative numbers in a matrix. The program features nested loops.
`isqrt`	Calculates the integer square root of a fixed number.
`jfdctint`	JPEG slow-but-accurate integer implementation of the forward Discrete Cosine Transform on an 8x8 pixel block.
`matrix1`	Generic matrix multiplication of two 10×10 matrices.
`md5`	Cryptographic hash function of Message Digest Algorithm 5.
`prime`	Prime number test on two random integers.

The results are shown in Figure 6.1. The cycle-counter measurements showed a maximum deviation of one cycle. Therefore, only the larger value is displayed. The upper bounds of `binarysearch`, `countnegative`, `isqrt`, `jfdctint`, and `matrix1` overestimate the execution time by up to a factor of 1.85. For `bsort` and `md5` the estimates reach a factor of 4. Most instructions have a fixed execution time, but some such as `DIV`, all branching or load/store operations have a varying number of cycles, of which PLATIN uses the maximum. This introduces additional pessimism, as `bsort` executes many swaps, which are load and store operations, and as `md5` also contains many load/store operations. In addition to that, `bsort` contains a triangular loop, which cannot be handled by PLATIN appropriately, and is therefore overestimated.

Overall, the upper bounds determined by PLATIN are reasonable estimates of the WCET: The measured times remain below the upper limit, and the upper limit is overestimated, but within a reasonable range. Therefore, the WCET estimates can be used to evaluate the performance of the QP in the next section.

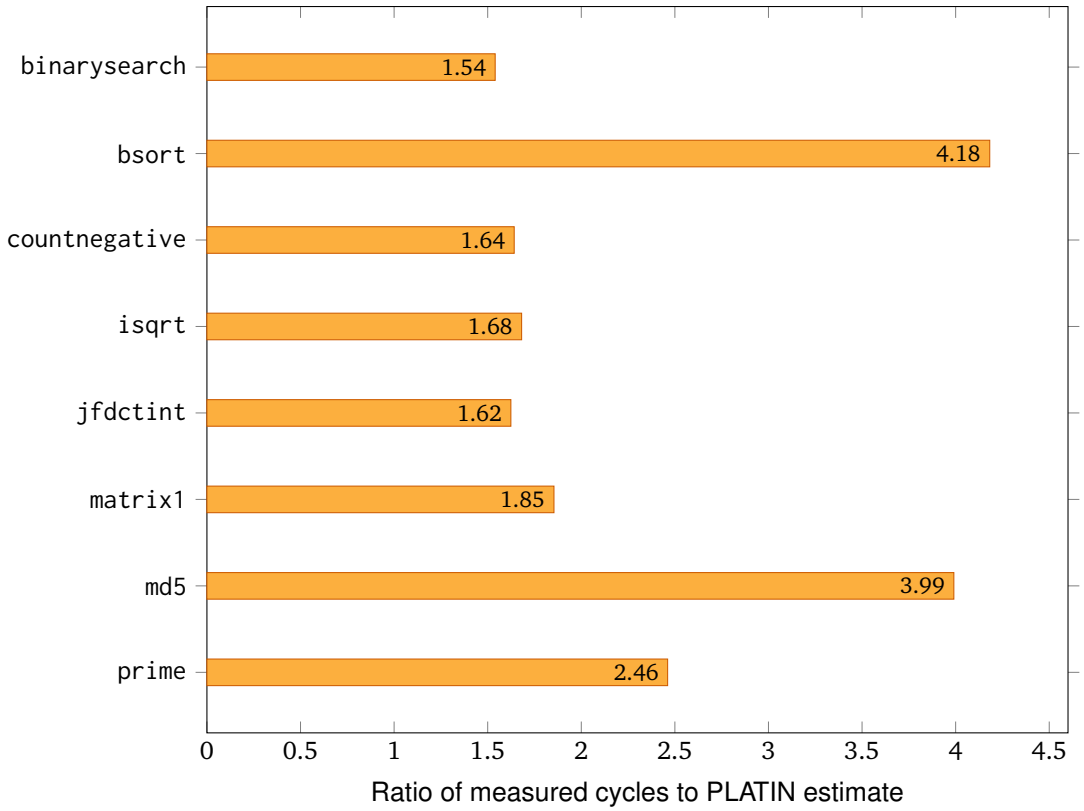


Figure 6.1 – Execution times compared to the upper bound of the execution time by PLATIN's WCET analysis. The orange bars display the measured cycle counter divided by the corresponding WCET estimate of PLATIN.

7 EVALUATION FOR THE ESP32-C3

The main goal of this thesis is to develop a generic method to reduce the energy consumption of a system for a given ordered set of periodic tasks. The mathematical model from Chapter 4 determines an optimal solution for this. It uses the power consumption of different clock-tree configurations and the reconfiguration costs for both time and energy. All those values depend on the given hardware platform. Therefore, these values must be determined before the QP can give a platform-dependent solution. The solution then returns the optimal clock-tree configurations for the task set. Measurements for the chosen hardware platform, the ESP32-C3, are performed in order to evaluate test input sets.

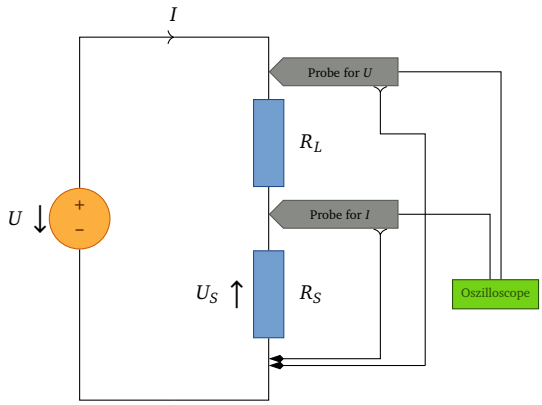
First, Section 7.1 deals with these measurements. It starts with the details of the measurement setup, followed by the measurements for active modes, peripherals and sleep modes. Afterwards, the penalties for changes of the clock-tree configuration are discussed. Section 7.2 uses the determined values to evaluate the mathematical model for the ESP32-C3 on two test programs. Finally, Section 7.3 discusses whether the approach of this thesis is applicable for the ESP32-C3.

7.1 Measurements

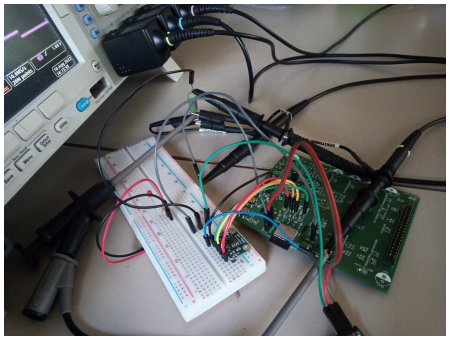
Before the QP can be given to a mathematical solver, the system-dependent values of the QP description have to be filled in. On the one hand, this means a program's tasks must be analysed to determine the WCET of the phases of each task. This thesis developed an architecture model of the ESP32-C3 for PLATIN for determining the WCET, as already discussed in Section 6.2.2. On the other hand, the power-consumption values of each configuration and the penalties of changing between them need to be known for the ESP32-C3. These are determined by the measurements in this section. First, Section 7.1.1 explains the measurement setup for determining power-consumption values of the ESP32-C3. Afterwards, all necessary values for the mathematical description from Chapter 4 are shown. The different measurements can be split up into the following categories: First, Section 7.1.2 discusses the measurements for active CPU states. Afterwards, the power consumption of the peripheral devices on the ESP32-C3 follow in Section 7.1.3, before Section 7.1.4 analyses the sleep modes. Finally, Section 7.1.5 investigates the clock-tree reconfiguration penalties on the ESP32-C3.

7.1.1 Measurement Setup

Electric power (P) is the product of voltage (U) and current (I): $P = U \cdot I$. Both U and I must be known or measured to determine power-consumption values. While a chip requires a fixed voltage



(a) Schematic display of a shunt-based measurement setup: R_S displays the shunt with a resistance of R_S, R_L is the resistance of the load. An oscilloscope with two probes is connected to this circuit: the first one measures the voltage U, the second one measures the current I using the shunt.



(b) ESP32-C3 measurement setup: The ESP32-C3 is connected to a power source, configured to 3.3 V. The probes of an oscilloscope are connected to different parts of the ESP32-C3 to measure the voltage and the current, and to react on a trigger provided by the ESP32-C3 software.

Figure 7.1 – Measurement setup.

U, the current I varies with different loads of the chip. Therefore, the current has to be measured to determine the overall energy consumption. As measurement infrastructure, an oscilloscope can be used, which measures voltage. To measure the current with an oscilloscope, the setup uses a shunt-based measurement with a low-side shunt, as displayed in Figure 7.1a. A shunt is a low-ohm resistor with high accuracy regarding its tolerance R_S and is built into the measurement infrastructure by connecting it in series with the load R_L, in this case, the ESP32-C3. As the current flows through the shunt, it generates a voltage drop. The oscilloscope measures this drop. With Ohm's law,

$$U = R \cdot I \Leftrightarrow I = \frac{U}{R}$$

dividing the measured voltage, U_S, by the size of the shunt, R_S, results in the current,

$$I = \frac{U_S}{R_S}$$

and, therefore, the following formula determines the system's power consumption:

$$P = U \cdot I = U \cdot \frac{U_S}{R_S}$$

A picture of the actual measurement setup is shown in Figure 7.1b. As oscilloscope, a Tektronix DPO4034 is used [24]. A 1 Ω shunt is connected to a `GND` pin of the board. Channel 1 of the oscilloscope, which measures the voltage, is connected to the 3.3 V pin together with the power supply. The `GND` from both probe and power supply are connected to the other end of the shunt. Channel 2 is connected to both sides of the shunt. It indirectly measures the current, as the size of the shunt is known and the oscilloscope measures the voltage over the shunt. The third channel is connected to pin 3 of the board, which acts as the trigger pin for the oscilloscope. An evaluation program on the ESP32-C3 triggers this channel by setting or clearing a GPIO pin. When channel 3

measures an edge between the low and high state of the GPIO, the measurement starts. The data of all three channels is collected by the oscilloscope. Therefore, if the time behaviour is of interest, e.g., in between two trigger points, the oscilloscope collects all data provided by the channels. The Tektronix DPO4034 is connected to a computer, which configures the oscilloscope, e.g., the sampling frequency, and starts data collection. The computer retrieves that data and stores it in a file in HDF5 format. Each measurement was run five times to detect possible outliers. The outputs, where each run collected 20,000,000 samples in the configured sampling rate, are stored in separate output files.

7.1.2 Active Modes

The chosen hardware platform, the ESP32-C3, provides many different options to choose from for its CPU frequency. On the one hand, the PLL_CLK provides either 160 MHz or 80 MHz from two different clock sources. On the other hand, there are the XTAL_CLK at 40 MHz and the FOSC_CLK at 17.5 MHz, where both can have a divider reaching from 1 up to 1024. This results in a total CPU range from 160 MHz to 17.1 kHz. While having these many CPU-clock options available equals a broad range of configurability, not all options are included for measurement and QP solving. Measuring all options takes much time, and the difference in energy consumption gets marginal. Additionally, the number of connections in the QP grows quadratically with an increasing number of available options per phase. Therefore, the problem gets harder to solve and the mathematical solver takes additional time to deliver the solution for a task set. Because of this, the CPU frequencies evaluated are limited to the following five: 160 MHz and 80 MHz from the PLL_CLK clocks, and 40 MHz, 10 MHz, and 1 MHz from the XTAL_CLK.

For the PLL_CLK, there are two different base clocks: a 480 MHz and a 320 MHz clock. When comparing these two base clocks for both 160 MHz and 80 MHz settings, a difference in energy consumption is not measurable with the current setup. As there are incompatibilities as either unintended rebooting or entering a panic state when changing from the 320 MHz clock back to the 480 MHz clock, the evaluation will only use the 480 MHz base clock.

All five CPU speeds were evaluated in a row, and each separately to verify these results. The sampling rate of the oscilloscope for this measurement is 1 MHz. During an active CPU phase test, a loop performs a sum calculation to keep the chip active. Figure 7.2 displays an oscilloscope measurement containing all five CPU frequencies. Table 7.1 lists the measured power-consumption values. As the frequency drops, the current consumption and, therefore, the power consumption also gets lower. However, the fastest clock is also the most energy efficient in terms of operations per cycle: while the 160 MHz clock reaches 5.16 MHz per Ampere, the 1 MHz clock only achieves 0.12 MHz per Ampere. Also, all other clocks do not achieve the same efficiency as the fastest available clock with 160 MHz. This phenomenon was also mentioned in Chiang et al. [7] and Bambagini et al. [5]: fast clocks are more energy efficient for CPU operations.

In summary, the most energy-efficient variant for a CPU-only task on the ESP32-C3 is a so-called *race-to-idle* behaviour: The fastest clock is selected to finish the task as soon as possible and, afterwards, to enter a sleep mode for idling. This results from the following: The energy consumed is calculated as the time needed multiplied with the power consumption of the chosen setting. The slower clocks draw less current per second. However, tasks take longer to complete with these settings. Overall, the time saved by executing a task with the fastest clock outweighs the energy savings of a slower clock for the ESP32-C3. This can change with peripherals: These often need a fixed amount of time to e.g., complete a message transmission. In this case, the power consumption per second is more important than the power consumption per CPU cycle. As a consequence, slow clocks perform better when the system has to wait, e.g., when working with peripherals [5, 7].

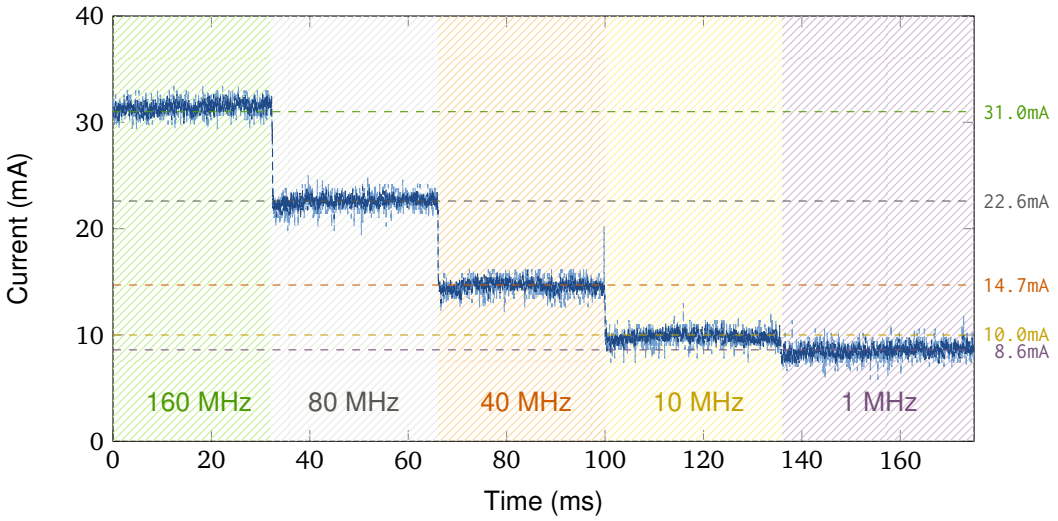


Figure 7.2 – Active phases: The graphic shows the current and the rolling average over 100 values from a range of 8,750,000 values. When switching the frequency of the CPU clock from 160 MHz to 80 MHz to 40 MHz to 10 MHz and finally, to 1 MHz, the current consumption drops, although the fastest clock remains the most power-efficient one. The dashed horizontal lines indicate the average measured values for each CPU frequency.

The following section analyses the power consumption of the peripheral devices attached to the ESP32-C3.

7.1.3 Peripherals

The ESP32-C3 lists many peripheral devices. The list includes GPIO, SPI, UART, I2C, or a temperature sensor. Since this list is extensive, this thesis restricts itself to the following subset:

SPI For communication via SPI, an SPI FRAM chip [23] was attached to the ESP32-C3. The evaluation deploys the same loop as for testing the CPU phases. However, during each iteration, the lowest 8 bit of the sum are written onto the SPI memory chip and read back. As this adds additional runtime to each loop iteration, the test decreased the number of iterations. As displayed numerically in Table 7.1 and graphically in Figure 7.3, the overall consumption did change slightly in comparison to the CPU measurements. However, the measured current is very noisy compared to without SPI communication.

Table 7.1 – Current consumption values of the ESP32-C3 with different active peripherals.

Frequency [MHz]	None [mA]	SPI [mA]	LED [mA]	I2C [avg. mA]
160	31.0	30.0	35.2	25.6
80	22.6	21.7	25.8	22.3
40	14.7	13.6	17.4	17.2
10	10.0	9.5	13.4	14.5
1	8.6	8.3	12.0	14.4

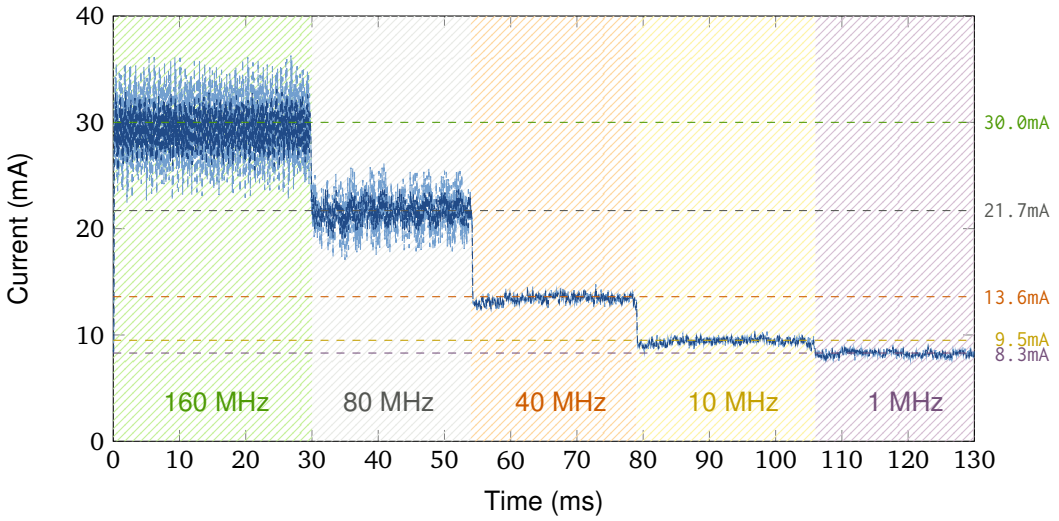


Figure 7.3 – SPI test: As for testing the active phases, all available clock configurations are tested in a row, but during each loop of active idling, the SPI FRAM is read and written once. This leads to a slightly lower average current consumption during SPI communication, but the measured current fluctuates much more, especially for the 160 MHz and the 80 MHz communication: To be able to display the measured values, the current is presented as a rolling average of 2000 and of 5000 samples from 6,750,000 values in the selected range. As in Figure 7.2, the dashed horizontal lines display the average measured values for each CPU frequency.

LED The power consumption of a single LED is tested by alternatingly setting and clearing the corresponding GPIO pin for the LED while changing the clock frequency from 160 MHz to 80 MHz to 40 MHz to 10 MHz and, finally, to 1 MHz. Figure 7.4 visualises this measurement. Table 7.1 lists the corresponding values. In summary, an active LED adds an overhead to the consumption of 3.4 mA on average.

I2C An 8-bit I/O expander for an I2C bus [27] was connected to the ESP32-C3, together with an 1.5 kΩ resistor. I2C uses pull-up resistors for the data and clock line, which consume an extra amount of power when current is drawn through the resistor. An additional drain of $3.3\,\text{V}/1.5\,\text{k}\Omega = 2.2\,\text{mA}$ per active lane is to be expected. Figure 7.5 displays I2C communication at 160 MHz. It demonstrates this drain behaviour corresponding to the active lanes: The clock line repeatedly changes from high to low, which causes the current to form a waveform: When the clock line goes low, more current is drawn due to the pull-up resistors. The same is true for the data line: Pulling this line down causes shifts in the waveform of the current. This results in the final picture in Figure 7.5. Important to mention is that the current drawn is less than during CPU operation with the same frequency. This is caused by the fact that the underlying FreeRTOS implementation blocks when it waits for the next byte according to the clock frequency of the I2C bus. A regular timer tick wakes up the chip during the blocked state to check whether enough time has elapsed so far to unblock the task. The tick time of the FreeRTOS is set to 100 Hz. Therefore, the CPU is less active than during active idling. On average, one can assume that both clock and data line are set half the time and therefore take the average of the measured state as the assumption for the power consumption.

The timing behaviour of I2C communication solely depends on the clock speed of the I2C bus, as a consequence, the time needed to complete one I2C transaction at a certain I2C clock frequency is fixed. As observed in Section 7.1.2, higher CPU frequencies are the most energy efficient one per CPU cycle, but draw more current over time. On the other side, low CPU frequencies complete less instructions per second, but consume less power. This is also seen when using I2C, where Table 7.1 and Figure 7.6 show the measurements: Lower frequencies consume less power. The time needed for one I2C message stays the same due to the fixed length of bits to be transmitted and the fixed I2C clock speed. Therefore, the slower frequencies are the optimal configuration for a single I2C phase.

These peripherals show an influence on the power consumption of the system. Also, the behaviour mentioned in Chiang et al. [7] and Bambagini et al. [5] - that slower clocks are more suitable for communication - holds true for the ESP32-C3. Nevertheless, future work includes evaluating further peripherals such as Bluetooth or WiFi. For example, the WiFi chip peaks up to 350 mA according to the documentation [11], which is ten times the baseline power consumption (i.e., active mode at 160 MHz). Such a device would be particulary interesting with regard to the saving potential compared to a pessimistic all-always-on approach. First measurements of the WiFi chip indicated a complex timing behaviour, which is not as easily determined as it is for SPI or I2C communication. Additionally, there is no documentation of the WiFi chip available as of now. Therefore, modelling it is considered as future work.

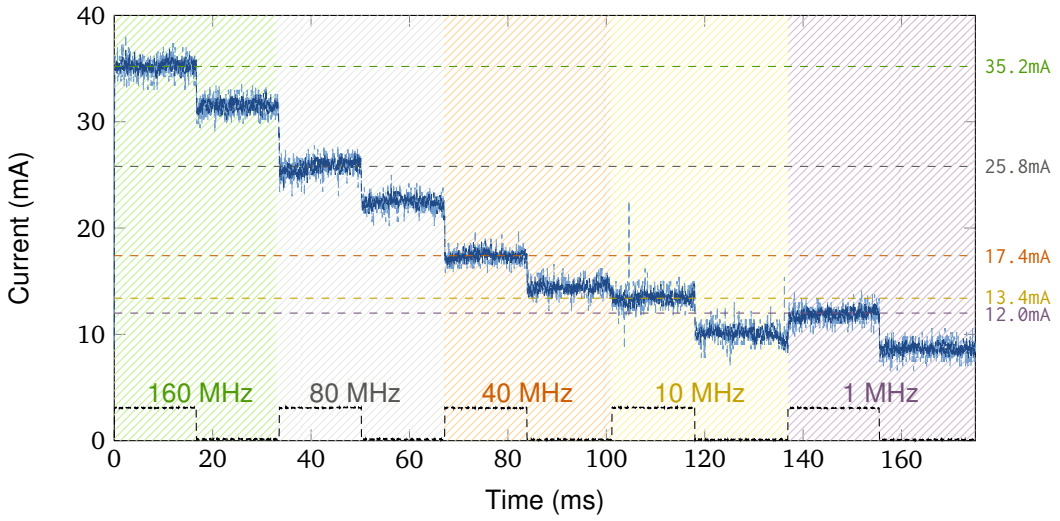


Figure 7.4 – LED test: As for testing the active phases, all available clock configurations are tested in a row, but each configuration is run with LED turned on and then off before changing to the subsequent frequency. The graph displays the current and the rolling average of 100 values from a displayed range of 8,750,000 values. The black line visualises the trigger signal, which corresponds to turning the LED on and off. It can be seen clearly that the LED adds additional current consumption, which averages to 3.4 mA of additional current consumption for all frequencies. The dashed horizontal lines display the average measured values for each CPU frequency when the GPIO pin is set to high and, therefore, the LED is turned on.

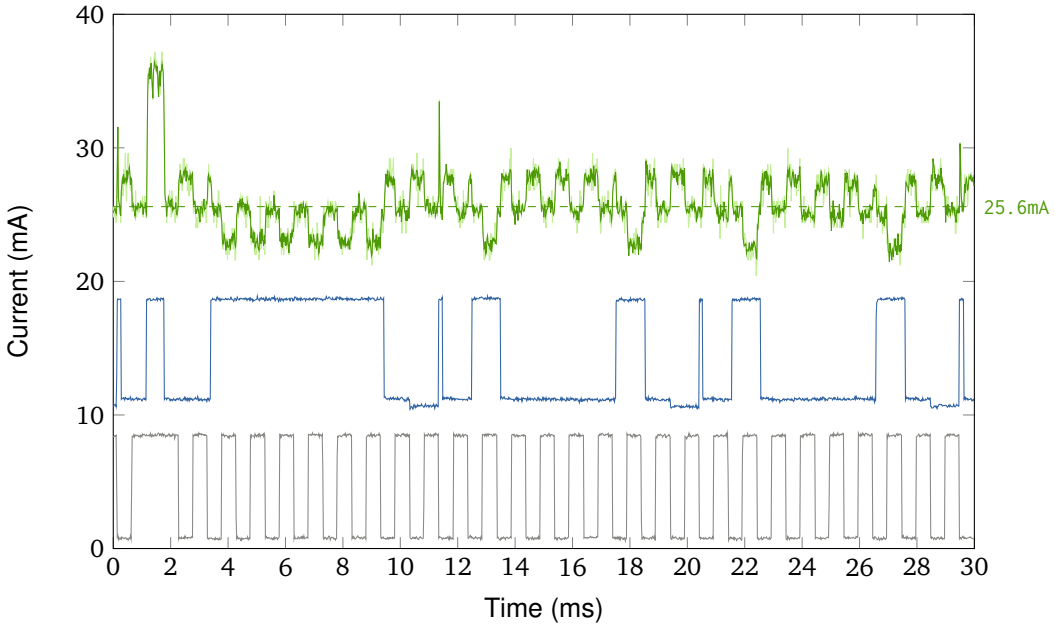
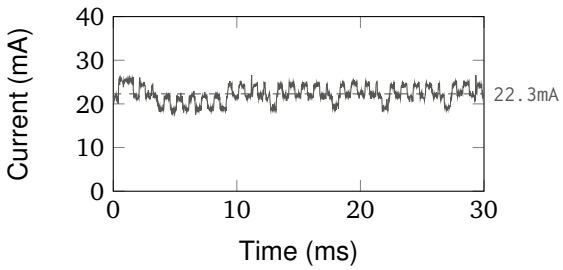
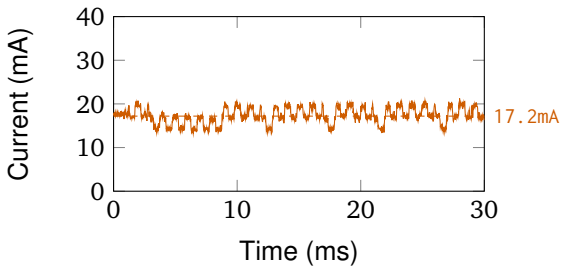


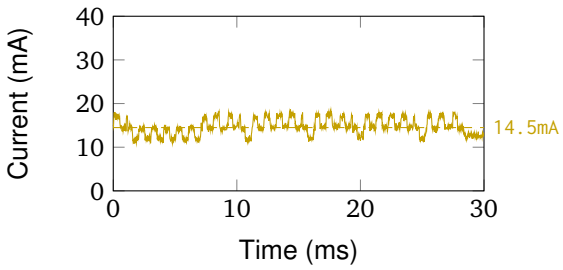
Figure 7.5 – I2C at 160 MHz: The green line displays the current, the blue line shows the data line of the I2C communication, and the black line is the clock signal of the I2C communication. To present the values, the data line and the clock signal are scaled and shifted. The current averages to 25.6 mA during I2C communication. The pull-down resistors draw extra current when a line is set to low. Therefore, fluctuations around this value - visualised by a dashed line - are caused by the changes in the data and clock signal line.



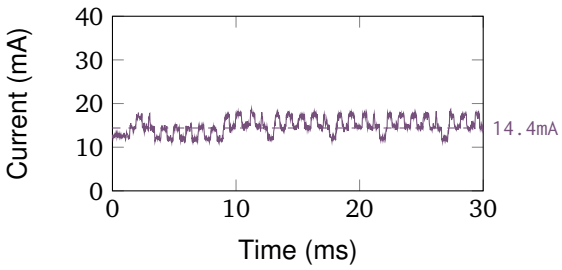
(a) I2C communication at 80 MHz.



(b) I2C communication at 40 MHz.



(c) I2C communication at 10 MHz.



(d) I2C communication at 1 MHz.

Figure 7.6 – I2C with varying CPU frequency for the same task as displayed in Figure 7.5.

7.1.4 Light Sleep and Deep Sleep

Measuring the energy consumption of light and deep sleep involved a problem with the measurement setup. Although the oscilloscope measured lower values than in the active states, the values still were significantly higher than those listed in the manual [9]. According to it, light sleep consumes 130 µA and deep sleep consumes 5 µA. The measurements returned values around 2 mA for light sleep and 1 mA for deep sleep. A closer look at the measurement setup showed that even when no probe is connected to the oscilloscope, values just below 1 mA were measured, which could not be eliminated by signal path compensation. This means that the measurement setup already introduces some noise that overlays the values to be measured for light and deep sleep. As a consequence, the values of the energy consumption for both sleep modes are taken from the manual until a different measurement setup is available.

7.1.5 Reconfiguration

The following measurements look at the penalties for reconfiguring the clock tree. These are split up into four categories: switching between frequencies (Section 7.1.5.1), configuring devices (Section 7.1.5.2), light sleep (Section 7.1.5.3) and deep sleep (Section 7.1.5.4).

7.1.5.1 Switching between Frequencies

Since switching between different CPU frequencies on the ESP32-C3 only requires the execution of a few CPU instructions (at most 21 CPU cycles), the time for the current to change is very short. Measurements have shown that the delay in the measured current introduced by the measurement setup and hardware on the board, such as capacitors, is too large to have reliable measurements for such a short period. Instead, the WCET of each transition multiplied by the previous CPU frequency is assumed as the time needed to perform the change. As the last instruction changes the frequency, the complete sequence is still executed with the old setting. The power consumption of the previous state is taken to determine the energy consumption.

7.1.5.2 Configuring Devices

SPI Before SPI communication can be used, the SPI master has to be set up by the system. After usage, it can be deinitialised. To perform these operations, this thesis uses the ESP-IDF functionalities and measures their behaviour. Table 7.2a displays the resulting time and current consumptions.

LED Turning on or turning off the LED is done by setting or clearing a bit in a memory-mapped register. As for switching between the CPU frequencies, this is too fast to be captured by the used measurement setup. Therefore, the WCET of each transition is multiplied by the current CPU frequency to determine the time. The energy consumption uses the energy behaviour of the state to be left.

I2C Similar to SPI, the I2C master has to be initialized before communicating and can be deinitialized after usage. The ESP-IDF functions are used here as well, and measurements of them were taken. Table 7.2b displays the measured values.

Table 7.2 – Time and power consumption for clock-tree reconfigurations.

(a) SPI.

Frequency [MHz]	Power [mA s]		Time [ms]	
	Init	Deinit	Init	Deinit
160	0.7148	0.7185	26.50	26.50
80	0.6594	0.5543	32.90	26.80
40	0.3987	0.3398	32.40	27.30
10	0.2771	0.2605	35.50	30.90
1	1.3885	1.3003	189.90	177.70

(b) I2C.

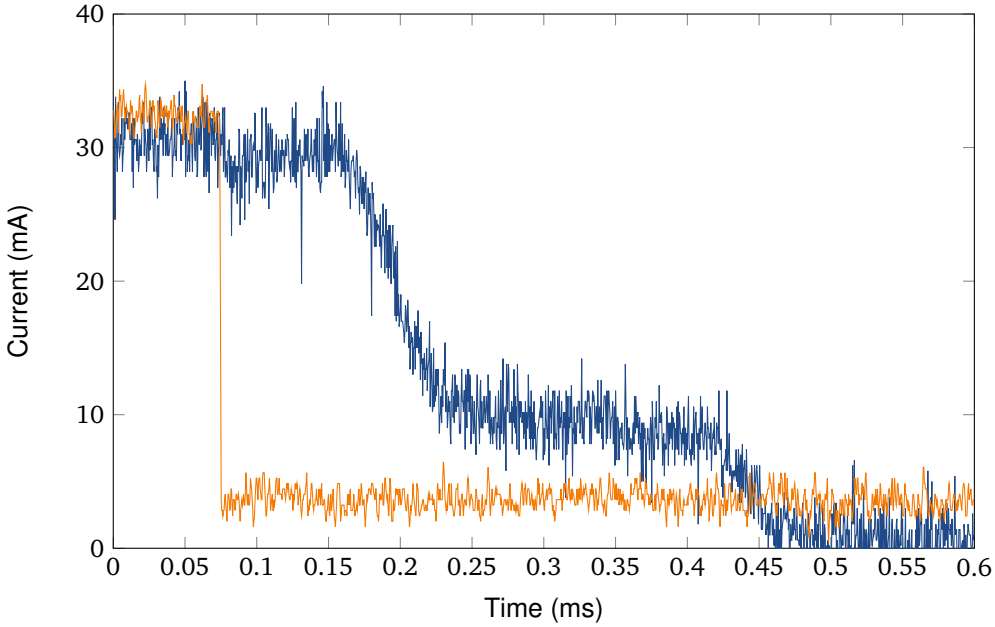
Frequency [MHz]	Power [mA s]		Time [ms]	
	Init	Deinit	Init	Deinit
160	0.0024	0.0007	0.07	0.02
80	0.0025	0.0009	0.10	0.04
40	0.0030	0.0012	0.19	0.08
10	0.0099	0.0032	0.87	0.31
1	0.0850	0.0301	8.50	3.12

(c) Light sleep.

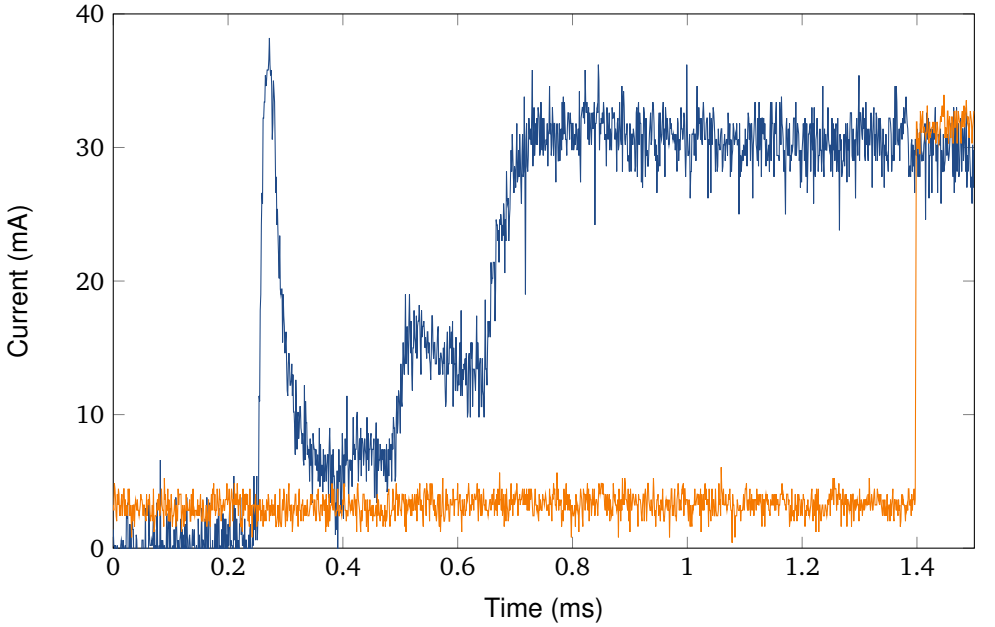
Frequency [MHz]	Power [mA s]		Time [ms]	
	Sleep	Wakeup	Sleep	Wakeup
160	0.0066	0.029	0.45	1.14
80	0.0083	0.023	1.10	1.18
40	0.0094	0.015	1.03	1.08
10	0.0072	0.014	1.28	1.48
1	0.0275	0.053	4.39	7.28

(d) Deep sleep.

Frequency [MHz]	Power [mA s]		Time [ms]	
	Sleep	Wakeup	Sleep	Wakeup
160	0.0085	6.545	0.44	296.70
80	0.0095	6.657	0.74	295.70
40	0.0111	6.816	0.99	295.30
10	0.0233	6.677	2.29	295.00
1	0.1957	6.732	19.74	295.20



(a) Transition into light sleep: After the trigger signal goes down as the last action before starting the transition into light sleep, the current stays high before dropping to its minimum after a penalty of 1.4 ms.



(b) Transition from light sleep: Before waking up after light sleep and executing the first instruction afterwards, which is pulling up the trigger signal, the power consumption goes up to the same level as before entering light sleep. This period before being active again and the corresponding power consumption are the penalties of waking up from light sleep.

Figure 7.7 – Light sleep transitions from and to 160 MHz: In both figures, the blue graph displays the current, and the orange graph shows the trigger signal.

7.1.5.3 Light Sleep

When transitioning from light sleep to normal mode and back, a non-linear progression of current consumption can be observed, where Figure 7.7 shows the switch from 160 MHz to light sleep. First, Figure 7.7a is discussed, which displays the transition into light sleep. Before starting the transfer to light sleep, the last instruction is to pull the trigger GPIO to low. After that, the current remains high until all devices are powered off. It finally drops to its lowest after around 0.4 ms. Figure 7.7b displays the wakeup process: The first change in current consumption is seen around 1.2 ms before the trigger is set back to active. After this, the current consumption increases until it reaches the level of before entering light sleep.

The area underneath the curve in both subgraphs is the observed current consumption of going to and waking up from light sleep. Table 7.2c lists these values for all frequencies. As the CPU frequency decreases, the time as well as energy consumption increase. While the increase in time is because a lower frequency means that the CPU can execute fewer instructions per second, the energy consumption increases as these lower frequencies are less energy-efficient than the higher ones. This leads to a similar result as in Section 7.1.2: The fastest CPU clock can enter light sleep with the lowest penalties for both time and energy.

When FreeRTOS is asked to go to light sleep for a certain period of time, it will wake up after that time span and resume operation in the same system setting than before. The measurements verify that behaviour. Light sleep resumes in the same state as it was entered, which is essential for modelling the ESP32-C3 with the QP. There are two options to model that: One option is to add additional constraints to the phases before entering and after leaving light sleep. Both must have the same system configuration. This, however, constrains the potential solution space of the QP, if e.g., the optimal settings for both phases differ. To circumvent this, two additional phases before and after light sleep can be introduced. This enables the possibility to switch to the optimal frequency for entering and leaving light sleep, as the reconfiguration costs to enter and leave light sleep differ for all frequencies. The disadvantage is that the model contains two additional phases. The other option is to introduce separate light sleep modes for each available configuration. For this, the reconfiguration penalties must include the penalties of entering the selected configuration in addition to the costs of changing to or from light sleep. Although this does not add extra phases, the number of configurations for the sleep mode increases, and, therefore, adds additional reconfiguration paths. Whether the first or second option is better depends on the given task and the given hardware model.

7.1.5.4 Deep Sleep

When evaluating deep sleep, it is necessary to consider that the chip always wakes up at 160 MHz. It does not restore the system settings prior to entering deep sleep. Therefore, when waking up from deep sleep, all differences must be reconfigured before the execution can continue. Also, the application needs additional preparations to reach its previous state of execution again since only the RTC controller, the RTC peripherals and the RTC fast memory remain powered on during deep sleep. These can be used to remember the previous system state. However, restoring the settings requires additional implementation effort for the developer.

Therefore, the test setup for deep sleep differs to the one for light sleep: The last instruction before going to sleep is clearing a GPIO pin as trigger signal. When waking up, the system restores the previous setting before setting the GPIO pin to high afterwards. The first transition, where the GPIO pin switches from high to low, measures the change from the previous clock-tree configuration to deep sleep. The second transition, where the GPIO pin switches from low to high, marks the change from deep sleep to the given clock-tree configuration. Different from light sleep, the phases

before and after deep sleep are not connected to each other. Therefore, the model does not need additional constraints for the deep sleep of the ESP32-C3.

The evaluation showed a similar behaviour for going to deep sleep as for entering light sleep for all five CPU frequencies. Figure 7.8 shows the transition for 160 MHz as an example. The chip takes time until the current consumption drops to its lowest, which also happens in a non-linear current consumption curve. As for the lower frequencies, each instruction takes longer to execute. As a consequence, the preparations for entering the sleep mode take more time. As mentioned in Section 7.1.2, the lower frequencies are less energy-efficient than the higher ones. This results in an increase in the overall current consumption as the frequency decreases. The measurements, shown in Table 7.2d, support this outcome.

As already mentioned, the wakeup process from deep sleep is different than from light sleep: The entire chip is powered down, except for the RTC controller, the RTC peripherals and the RTC fast memory. The ESP32-C3 supports different wakeup signals, as mentioned in Section 6.1.4. In the setting in Figure 7.8, a timer wakes up the chip after 100 ms. The first spike in current consumption appears after 100 ms. Afterwards, the chip enters the boot sequence, which takes just below 300 ms, before the trigger is set to active. The wakeup time of nearly 300 ms outweighs the reconfiguration costs, therefore, the wakeup time was about equal for all frequencies.

In comparison to active idling with lower CPU frequencies or light sleep, deep sleep adds significant penalties for reconfiguring the chip. Although the costs for entering deep sleep are comparable with the costs of entering light sleep, the wakeup process takes up to 270 times that of light sleep, while having a larger energy consumption of up to 450 times that of light sleep. However, the energy consumption during deep sleep is 3.8 % of light sleep. Whether this saving is advantageous in comparison to the overhead depends on the remaining sleep time.

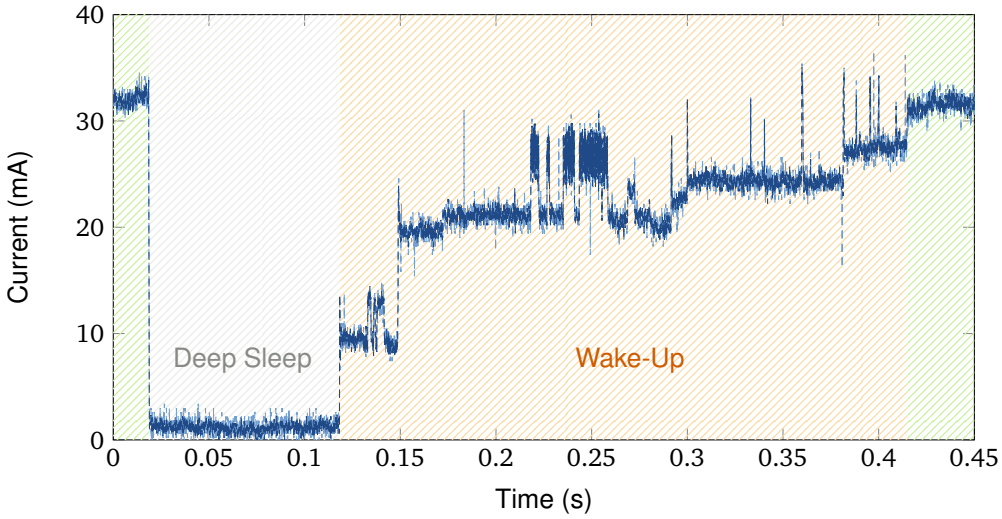


Figure 7.8 – Deep sleep transitions from and to 160 MHz: In the green sections, the CPU is active. First, the chip enters deep sleep for 100 ms. After that, the wake-up phase takes place, which takes nearly 300 ms. Finally, the CPU is active again.

7.2 Real Values for the Quadratic Program

Now, all necessary information for the QP is collected. To conclude this chapter, this section presents two evaluation scenarios. Section 7.2.1 starts with a simple example, which looks at the influence of reconfiguration penalties by evaluating different sleep modes. Afterwards, Section 7.2.2 extends the scope to a more complex evaluation scenario, which includes a peripheral device (I2C) and multiple phases.

7.2.1 Single Task

This scenario, which shall act as the baseline scenario, investigates two questions: The first one is to evaluate how the QP handles the fact that faster clocks are more power-efficient, as stated in Section 7.1.2, and whether this results in a race-to-idle behaviour for CPU computations. The second question is to investigate the influence of the penalties on the QP decisions and the resulting power consumption of the overall problem.

As mentioned in Section 7.1.5.4, the reconfiguration penalties of deep sleep outweigh other reconfiguration costs. Therefore, it is enough to analyse a single task to show the influence of penalties by using the penalties of entering different sleep modes. The given task only utilises the CPU and has a fixed execution length regarding the number of CPU cycles. Afterwards, it waits for the next period to start and enters a sleep mode. The choice of which sleep mode is the optimal one regarding the overall energy consumption depends on the duration of the sleep phase. This, in turn, depends on the given task set's period and, additionally, on the chosen CPU frequency for the active task, as the time remaining for sleeping is the total period minus the time consumed by the task. To tackle the first parameter, multiple problems with varying period lengths are evaluated. For the second one, the possible clock-tree configurations for the tasks are all five evaluated CPU clock frequencies: 160 MHz, 80 MHz, 40 MHz, 10 MHz, and 1 MHz. The available sleep modes are deep sleep with a large penalty, especially for waking up, light sleep with smaller but still measurable penalties, and 1 MHz active idling with the smallest penalties. As there is only one phase, the clock-tree configuration before and after sleep is the same. Therefore, light sleep is modelled correctly.

```
1  int fib (int x1) {
2    if (x1 == 0 || x1 == 1)
3      return 1;
4    int grandparent = 1, parent = 1, me = 0;
5    _Pragma( "loopbound min 0 max 1000001" )
6    for (int i = 2; i <= x1; ++i) {
7      me = parent + grandparent;
8      grandparent = parent;
9      parent = me;
10   }
11   return me;
12 }
```

Listing 7.1 – Iterative Fibonacci calculation: PLATIN estimates 8,000,031 CPU cycles for the example input of 1,000,000, which equals to just above 50 ms for the maximum CPU frequency of 160 MHz. The `_Pragma` acts as hint for PLATIN for handling the loop bounds regarding the WCET analysis.

The computation to be performed is the iterative calculation of a Fibonacci number, as displayed in Listing 7.1. The input number for the calculation was chosen such that the execution time with the maximum CPU frequency of 160 MHz takes just above 50 ms.

Between 50 and 60, the solver was run for each ms. Between 1000 and 75,000, a measurement was performed for every 1000 ms. Additionally, the range between 52,000 and 53,000 was investigated in more detail, the reason for this follows in a moment. Figure 7.9 shows the results. The solver chose the fastest CPU clock for all tests with a valid solution. This is the *race-to-idle* behaviour: The task is finished as fast as possible, as the fastest CPU clock is the most power-efficient and the sleep modes offer at least the efficiency per unit of time as the slowest CPU clock.

The choice of the sleep mode was as follows:

- For values smaller than or equal to 50 ms, the solver can not determine a solution, as the shortest execution time for the task itself is just above 50 ms.
- For a period of 51 ms, the time until the next period begins is spent in active idling, as the time needed to switch to a sleep mode and wake up afterwards is larger than the available time frame. Therefore, active idling is the only option available for this period length.
- At 52 ms, light sleep becomes an option, as the time penalties for entering and leaving light sleep are small enough. However, active idling is using less energy overall until a period length of 54 ms. The *break-even point* is reached at 55 ms, where light sleep consumes less energy than active idling. This is visualised in Figure 7.9b.
- For the range from 54 ms to 347 ms, active idling and light sleep are the only options. At 348 ms, deep sleep becomes available. However, the large reconfiguration penalties for deep sleep prevent the solver to choose this sleep mode, although the energy consumption in deep sleep is just 3.8 % of the energy consumption in light sleep.
- The *break-even point* between light and deep sleep is at a period length of 52,182 ms, visualised in Figure 7.9c. For all values larger than 52,182 , deep sleep is the best solution.

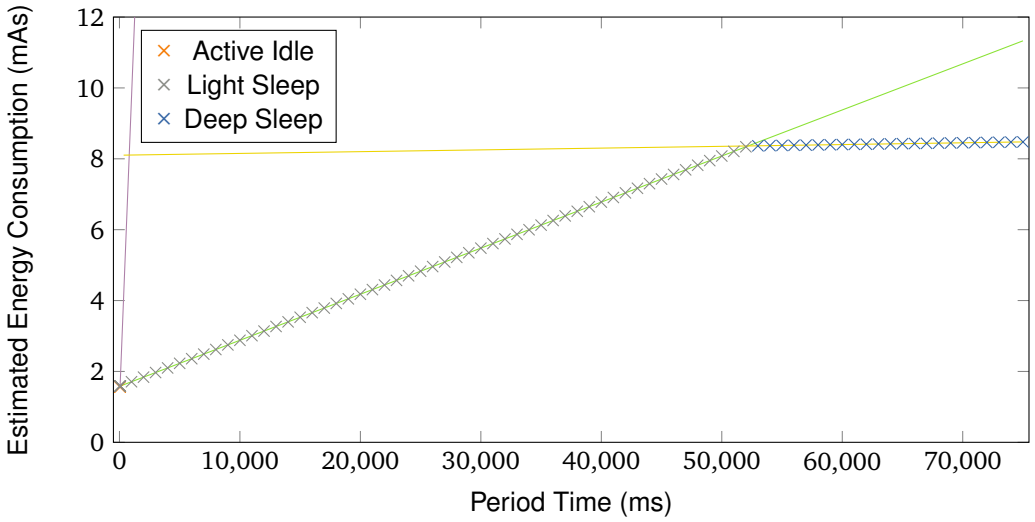
These results indicate that the clock-tree–reconfiguration penalties on the ESP32-C3 have an impact on the energy consumption of the system. Although some modes have lower energy costs, the reconfiguration adds a large overhead for both time and energy consumption. Therefore, these modes only outperform others as the total time available for this phase increases. Active idling with the lowest reconfiguration penalties was the best choice for a short sleep phase. Light sleep has a slightly larger overhead, but consumes only 1.5 % of energy consumption in sleep mode. This break-even point is passed for a sleep length of 5 ms. Deep sleep has a large penalty, especially for waking up, but only consumes 5 µA. The break-even point between light and deep sleep is reached after 52,133 ms.

This investigation showed significant influences on the optimal power consumption of the system: The baseline of the power consumption without any reconfiguration is a pessimistic *all-always-on* approach at the highest available frequency. In this example, the 160 MHz CPU frequency consumes 31 mA. The determined power-consumption savings at the break-even point between active idle and light sleep are just

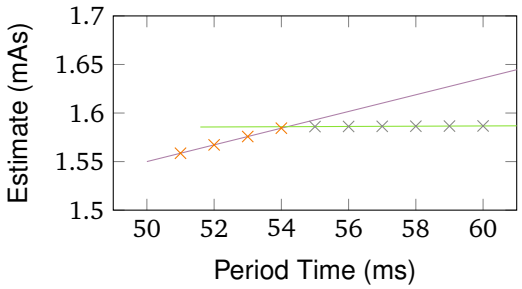
$$1 - \frac{1.593\text{mA s}}{55\text{ms} \cdot 0.031\text{A}} = 6.6\% \,. \tag{7.1}$$

However, the model predicts an energy saving of

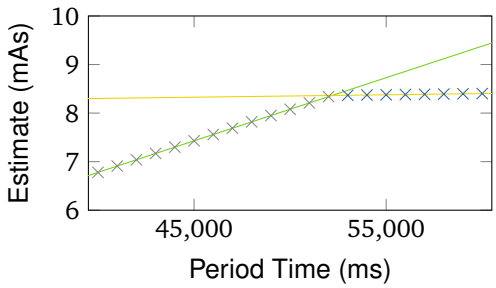
$$1 - \frac{8.363\text{mA s}}{52183\text{ms} \cdot 0.031\text{A}} = 99.5\% \tag{7.2}$$



(a) Full figure.



(b) Active idle to light sleep: break-even point between 54 and 55 ms.



(c) Light sleep to deep sleep: break-even point between 52,182 and 52,183 ms.

Figure 7.9 – Solver results for test program with one task: Between 50 and 60, the solver was run for each ms. Between 1000 and 75,000, a measurement was performed for every 1000 ms. The graphs display these values with marks. The colored lines display the theoretical energy consumption for active idle, light sleep and deep sleep when the fastest CPU frequency of 160 MHz is chosen. This was the case for all the results displayed in this figure.
As the minimum execution time of the task is just over 50 ms with the maximum CPU frequency of 160 MHz, there is no solution for values smaller than or equal to 50 ms. For a time of 51 ms, the solver chooses active idling, as the remaining time is not enough to enter light or deep sleep. For a period of 52 ms, entering light sleep is an option, but the reconfiguration penalties of light sleep are still larger than active idling. This changes at 55 ms, as displayed in Figure 7.9b. After 348 ms, deep sleep is an option for going to sleep, but light sleep is the better option until a period length of 52,182 ms. For all values above 52,182 ms, deep sleep is chosen. This transition is displayed in Figure 7.9c.

at the break-even point between light and deep sleep. These are promising results for the capabilities of this model. Also, this investigation only covered the influence of a single phase with its reconfiguration penalties. Adding more phases offers the potential for more success of the model.

7.2.2 Inter-Integrated Circuit Communication Test

The previous section covered the behaviour of the system with regard to the available time for the sleep mode. In this scenario, the behaviour of the model for a system with a peripheral device is investigated. It uses a task with five phases, as displayed in Listing 7.2. The first, third and fifth phases use the CPU and do not use any peripherals. The second and fourth phases both perform a single I2C communication, as measured in Section 7.1.3.

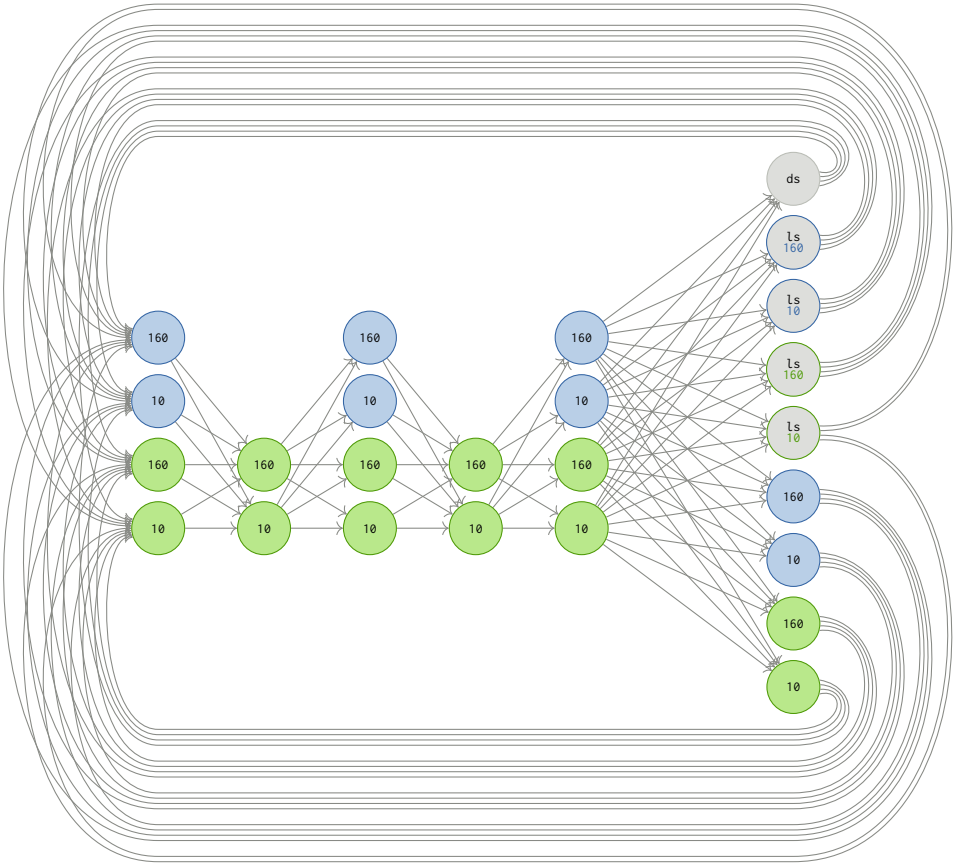


Figure 7.10 – Graph of peripheral test: Blue nodes describe the CPU frequency in MHz, green nodes the CPU frequency in MHz with a clock-tree configuration which allows I2C communication. The grey nodes are the sleep modes of the ESP32-C3. An edge between two nodes displays a potential configuration change. In the second and fourth phases, I2C communication is required. Therefore, the blue nodes cannot be used there. The ESP32-C3 has the same mode before and after light sleep. Therefore, the single light sleep mode is extended into four modes for each available configuration, which is used before and after light sleep. Additionally, all four active modes can be used for active idling during the sleep phase.

```
1 void task(int n1, int n2, int n3) {
2   /* Phase 1 */ fib(n1);
3   /* Phase 2 */ i2c_communication();
4   /* Phase 3 */ fib(n2);
5   /* Phase 4 */ i2c_communication();
6   /* Phase 5 */ fib(n3);
7 }
```

Listing 7.2 – Peripheral test: `n1`, `n2` and `n3` default to 1000000 as in the last section.

The scenario investigates two points:

1. The I2C communication is executed with the same I2C clock frequency, regardless of the CPU frequency setting. Therefore, it takes the same time for all CPU frequencies to complete the I2C transaction. The evaluation tests whether the lower power consumption for slower CPU clocks is then advantageous over the faster CPU clocks with higher power consumption.

2. During the I2C communication phase, there is an intermediate CPU phase, where I2C is not needed. Depending on the costs of a configuration with an active, but unused I2C unit and the length of the CPU phase, it may be beneficial to either leave I2C configured or turn it off.

For simplicity, this example uses only two CPU-frequency configurations of the measured five frequencies: 10 MHz and 160 MHz. The I2C-frequency configurations use the same base frequencies. Phases two and four use I2C. Therefore, these phases require a configuration which allows I2C communication. All other phases allow all available configurations. Additionally, the two sleep modes of the ESP32-C3 - light and deep sleep - are added to the available idle modes. As mentioned before, the ESP32-C3 restores the system state of before entering light sleep when leaving it. This is modelled here as follows: The single light-sleep idle mode is split up into four modes, which results in the graph shown in Figure 7.10. Each of these modes gets additional information on what configuration it takes before and after light sleep: 160 MHz as ls_{160}, 10 MHz as ls_{10}, 160 MHz with active I2C as $ls_{160,i}$, and 10 MHz as $ls_{10,i}$. While the power consumption during light sleep stays the same for all four light-sleep modes, the reconfiguration costs when entering or leaving the modes have to be adapted: For example, when changing to the sleep mode ls_{160}, the chip first has to change to 160 MHz before it can switch to light sleep. When the phase before light sleep already has the 160 MHz clock-tree configuration, there are no additional costs to the penalty of entering light sleep from 160 MHz. Otherwise, the costs of changing to 160 MHz are added to the reconfiguration penalties. Similarly, the penalties for waking up from light sleep must be recalculated.

For the values of the ESP32-C3 with a period length of 1,000,000 ms, the solver returns the optimal solution as follows: When I2C communication is active, the lower CPU clock is chosen, as the CPU frequency does not influence the runtime of the communication. During each CPU phase, the fastest available CPU clock configuration is chosen. This was also the outcome of the QP for the intermediate CPU phase.

This evaluation scenario showed that it is advantageous for the ESP32-C3 to use a lower CPU clock during I2C communication, which was the first question of this section. However, as there are no measurable disadvantages for power consumption when I2C stays powered on when not used, there is no need to turn it off and on again.

To test whether the QP formulation would choose to turn off I2C if there is additional overhead for active but unused I2C, the model of the ESP32-C3 is modified by adding extra current consumption of 10 mA for the clock-tree configurations including I2C: Instead of 31 mA for 160 MHz and 10 mA for

10 MHz, these modes now consume 41 mA and 20 mA. Whether turning off I2C is beneficial depends on the total overhead, which equals to the time span of the intermediate CPU phase. Therefore, the test varies the number of CPU cycles for the third phase. The results of the solver produce the graph presented Figure 7.11. For small values up to 11 cycles, the costs of changing to a clock with a higher frequency outweigh the benefits of the faster and, therefore, more energy-efficient clock. After that, it is better to change to a higher clock frequency, where the costs are low, but not to de- and reinitialize the I2C driver. The break-even point at which this becomes profitable is at 48,505 CPU cycles, from where the faster CPU clock with deactivated I2C is chosen for the intermediate CPU phase.

Therefore, an answer can be given to the second question of this section - that there is a break-even point where the reconfiguration costs to turn off active but unused peripherals get smaller than the additional costs of those peripherals. Although the ESP32-C3 cannot profit from the QP formulation in this case, the mathematical model is capable to further optimise such configurations. Depending on the length of the phase and the costs of the active but unused I2C unit, the benefits of different modes can outweigh the clock-tree reconfiguration penalties.

7.3 Conclusion for the ESP32-C3

All results from this chapter propose good results for the QP formulation and the hardware model of the ESP32-C3. The optimal CPU frequency for a CPU-only phase is the fastest clock for a *race-to-idle* behaviour: The task is finished with the lowest possible energy consumption, and can switch to a power-saving sleep mode afterwards. The sleep-mode selection depends on the remaining time in the current period. For short periods, an active-idle approach is the best, but with an increase in the remaining time, light sleep and deep sleep are better options. However, both sleep modes do

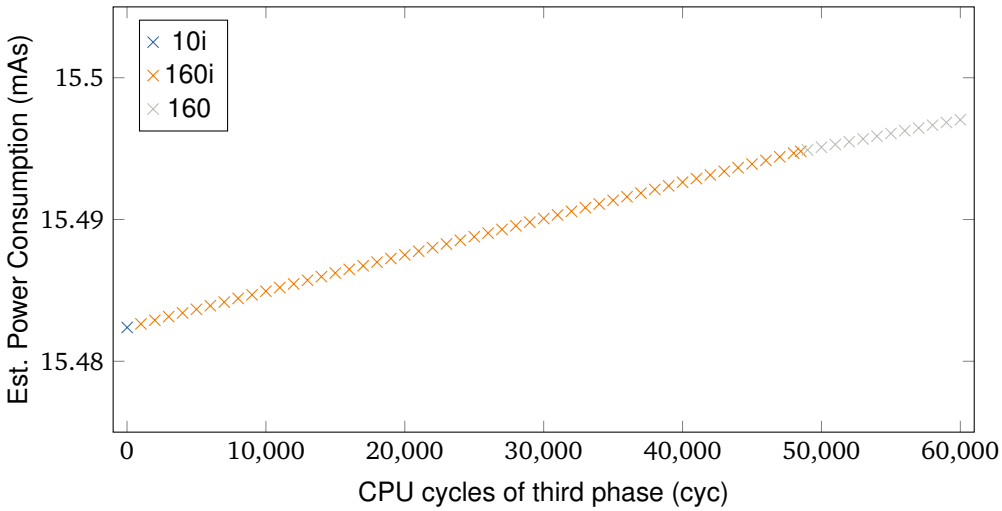


Figure 7.11 – Solver results for Listing 7.2 with I2C communication: The graph displays the chosen configuration for the third phase. For small values ≤ 11 CPU cycles, the penalties of changing from the 10 MHz with I2C setting to 160 MHz with I2C setting for the intermediate calculation outweigh the benefits of a faster clock. For values larger than 48,505, the penalties of deinitializing and reinitializing I2C are smaller than the benefits of the calculation with the 160 MHz setting in comparison to the 160 MHz with I2C setting.

not outperform the previous optimum as soon as there is enough time available to allow the higher reconfiguration penalties. For tasks with a fixed time, for example a single I2C communication, the priority is to choose the configuration with the least energy consumption per second. Here, slower CPU clocks perform better than fast CPU clocks.

For these relatively small problems the results are already promising. As larger problems incorporate more phase and a larger variety of peripheral devices with more varied reconfiguration penalities, it is expected that these offer the potential for more success of the model. More energy savings in comparison to pessimistic all-always-on approaches are likely to be achieved by measuring additional peripheral devices of the ESP32-C3, e.g., the WiFi chip. The acquisition of better measurement infrastructure is also planned to improve the measurement accuracy. As the time consumption of the single tasks are taken as the WCET to give guarantees, the tasks may complete earlier. Therefore, one can add a dynamic component to the system to save even more energy: The insights from Section 7.2.1 give clear instructions what sleep mode is the best for a certain remaining time in sleep mode. As a consequence, the system can decide dynamically whether it is worth to enter a less power-consuming sleep mode than predicted.

8 RELATED WORK

This chapter reviews related work and points out commonalities and differences to this thesis. The static analysis presented in this thesis optimised the energy consumption of a system with a fixed deadline. As a consequence, no optimisation happens jointly for both WCET and WCEC. Section 8.1 takes a look at related work for multi-objective optimisations. As this work relies on clock trees to reconfigure the system, Section 8.2 discusses other works on configuring clock trees. In Section 8.3, a runtime-based approach to reduce the energy consumption of microcontrollers is discussed, before Section 8.4 has a look at related work regarding energy-constrained real-time systems.

8.1 Multi-Objective Optimisation

This thesis optimises the energy consumption of a system with a fixed deadline. However, for energy-constrained real-time systems, both energy consumption and time are subjects to optimisation, which leads to multi-objective optimisation.

Lokuciejewski et al. [19, 20] propose a WCET-aware compiler framework to automatically determine compiler optimisation sequences to achieve highly optimised code. They investigate evolutionary multi-objective algorithms that identify Pareto-optimal solutions for multiple optimisation objectives. In embedded platforms, not only the WCET and WCEC, but also the code size is relevant, as the memory available on these chips is limited. Muts and Falk [25] discuss optimisations for function inlining, a well-known compiler-based optimisation. The idea of function inlining is to replace a function call with the function body, which eliminates the function-call overhead. However, the code size increases, as the body is inserted at every position of this function. Muts and Falk formulate this optimisation as a multi-objective optimisation problem with the WCET, code size and energy consumption as optimisation objectives. These objectives contradict each other. To find the best trade-off solutions, they also use evolutionary multi-objective algorithms.

The presented approaches also aim to minimise the costs of embedded devices. However, in comparison to this work, they do not consider device configurations and the corresponding clock-tree reconfigurations, which would further improve their optimisation goals.

8.2 Clock-Tree Configurations

Reconfiguring the clock tree is no new topic, as it is a prerequisite for power saving. Two implementations are described below.

Linux provides the Common Clock Framework [40]. The first part of the framework provides a generic interface which unifies the infrastructure of all clock nodes in the clock tree. The other part contains hardware-specific structures and implementations for particular clocks hidden behind generic callbacks. It explores the underlying hardware at boot up and dynamically builds up the clock subsystem by parsing the device tree and loading necessary modules for available devices. This results in an implementation which relies on dynamic memory allocation, recursive parsing and a generally large runtime overhead, which is not preferred for embedded SoC platforms.

The FlexClock approach [32] also presents an abstract representation of clock trees and dynamically explores and reconfigures the clock tree at runtime. The goal platform differs: While the Common Clock Framework targets laptops or desktop computers, FlexClock aims to support small embedded devices, which are usually restricted in memory and computing power. This complicates managing and (re-)configuring the clock trees, as the runtime implementation must be very efficient, and the code size limited. They break down every clock tree into three types of nodes: Scalers, multiplexers, and gates. These types were enough to model every clock tree they encountered. As such, only three node types were implemented, which reduces the code size. They also point out that more specific implementations can reduce the modelling overhead.

In this work, the clock-tree exploration happens before runtime. All changes are to be built into the application, therefore, the evaluation before the actual execution on the chip can use a more powerful computer instead of the embedded target platform to determine the optimal solution. Thus, there are no constraints on the complexity of clock-tree changes.

8.3 Power Management

Reconfiguring the clock tree can change the CPU clock of a system and, therefore, can have massive influences on the energy usage of a system. Chiang et al. [7] introduce a kernel-based dynamic clock management system for microcontrollers. By changing the clock according to the ongoing computation or the current pending I/O requests, energy can be saved by applying the following insight: Contrary to what is achieved with dynamic voltage and frequency scaling in conventional processors, faster CPU clocks are more energy efficient for CPU operations. In traditional systems, a lower clock allows for lower voltage and, therefore, lower power consumption, but the voltage remains the same in a microcontroller. This means that a faster clock requires more energy per second but less per cycle and is more efficient for CPU-intense computations, as long as the chip can enter an energy-saving mode afterwards. When waiting for answers of I/O operations, slow clocks are more efficient, as not the energy per cycle, but the overall energy consumption is of interest. Power Clocks, the introduced system, dynamically adapts to feedback about ongoing computational and peripheral operations as well as the application state and chooses a suitable clock.

It differs from this work: While achieving a low energy consumption is crucial for both, this thesis also wants to give guarantees for hard real-time systems with static analysis. This part is not a subject of the Power Clocks paper. Instead, they aim for a dynamic feedback approach, where it is impossible to give static guarantees. This might work well for standard applications on microcontrollers but makes power consumption and timing behaviour hard, if not nearly impossible, to predict.

8.4 Energy-Constrained Real-Time Systems

If a system wants to save energy, there are two main approaches: The first one is scaling the voltage or frequency of the chip, the second is putting it in an inactive state or turning it off completely. Those two techniques are referred to as Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM) in Bambagini et al. [5], where they present a survey of energy-aware scheduling algorithms proposed for real-time systems. Although the presented techniques use various approaches in regards to online or offline scheduling, overhead, or complexity, most do not account for the penalty introduced by changing the frequency or changing into an inactive state. Another work of Bambagini et al.[4] considers the switching overhead between two frequencies: The algorithm checks whether there is enough time to switch to the new frequency, execute the task under consideration at that frequency, and then switch back to the previous one. But - in contrast to this thesis - it neither considers such scalings in combination with system-wide analysis, nor aims for a system-wide optimal configuration for every task. Also, peripheral devices, whose availability is part of the clock-tree configuration, are left out of the scheduling approaches of energy-constrained real-time systems.

Another approach takes devices into account. For determining an upper bound of energy consumption, the worst-case energy consumption of the whole system can be assumed as follows: All peripherals are active all the time. This delivers an upper bound for the power consumption, but, as this usually is not the case, it overestimates the real power consumption. The SysWCEC approach [41] aims to minimise that error and give reliable upper bounds for the energy consumption by using static analysis. This splits into three steps: First, a CFG is derived from the source code. From the blocks of the CFG, SysWCEC explicitly adds transitions between these blocks induced by synchronous task activations or interrupts. This results in a state graph that includes the system's dynamic behaviour. Finally, the worst-case energy consumption of each node is calculated based on the WCET of this code block and the power states of all active devices. From these values, an ILP is formulated to derive the WCEC.

In addition to the approach of SysWCEC, this work seeks not only to provide guarantees for energy consumption but an energy-optimal solution for an ordered set of phases, similar to the blocks presented in SysWCEC. SysWCEC shows the direction in which this thesis's future work is going to advance: Instead of identifying the energy optimum for a fixed sequence of tasks, the model itself can be extended to allow flexible task arrangements as part of the optimisation strategy. Finding the optimal arrangement of multiple tasks will be required to determine the energy minimum for the entire system ultimately. Finally, merging the approach presented in this thesis with the SysWCEC approach [41] would allow an automatic energy-consumption optimisation of all tasks running in an energy-constrained real-time system.

9 CONCLUSION

This thesis describes a solution for *minimising the energy consumption* of a periodic, sequential set of tasks in an energy-constrained real-time system. The *clock tree* and its *reconfiguration* play the main role in this optimisation: Each task has a set of possible clock-tree configurations based on its required system settings. Each clock-tree reconfiguration introduces *penalties* regarding the time and energy-consumption behaviour of the system. They must be considered when determining the optimal configurations, as this can mean that the reconfiguration penalties to reach the most energy-saving configuration outweigh the additional costs of staying at a suboptimal configuration for one task. Such decisions can influence the whole task set.

For determining the optimal solution, this thesis introduces a static-analysis approach prior to runtime. A mathematical problem description takes the following parameters into account to model an optimisation program for a mathematical solver: A view on the whole system, clock-tree configurations, clock-tree reconfiguration penalties, their time and energy-consumption behaviour, all possible clock-tree configurations and the WCET for each task, and the tasks' deadline, which needs to be met to give timing guarantees. Additionally to giving guarantees for runtime and total energy consumption, the mathematical model is solved prior to runtime. Therefore, there is no additional overhead during execution.

To evaluate the model, this thesis performed a multipart evaluation. First, the practicability of the model for medium-size energy-constrained real-time systems (up to 80 tasks, each having up to 40 possible configurations) was investigated. All tests completed within a reasonable amount of time (less than one minute with Gurobi [14]). After that, the evaluation focus shifted towards applicability in the real world by analysing the ESP32-C3. A model was created for the open-source analysis tool PLATIN to calculate WCET estimates, which are verified with a benchmark suite. With measurements of the energy and time behaviour of the ESP32-C3 for the model, test scenarios showed that, compared to a pessimistic *all-always-on* approach, the optimisation achieved significant energy savings.

LIST OF ACRONYMS

ABB	Atomic Basic Block
API	Application Programming Interface
BB	Basic Block
CFG	Control-Flow Graph
CPU	Central Processing Unit
CSR	Configuration and Status Register
DPM	Dynamic Power Management
DVFS	Dynamic Voltage and Frequency Scaling
ESP-IDF	Espressif IoT Development Framework
FRAM	Ferroelectric Random-Access Memory
GPIO	General Purpose Input/Output
I2C	Inter-Integrated Circuit
ILP	Integer Linear Program
ISA	Instruction Set Architecture
IPET	Implicit Path-Enumeration Technique
LED	Light Emitting Diode
PABB	Power Atomic Basic Block
PMU	Power-Management Unit
QP	Quadratic Program
RAM	Random-Access Memory
ROM	Read-Only Memory
SPI	Serial Peripheral Interface
SRAM	Internal Static Random-Access Memory
UART	Universal Asynchronous Receiver Transmitter
WCEC	Worst-Case Energy Consumption
WCET	Worst-Case Execution Time

LIST OF FIGURES

2.1 Clock Tree Example 8

3.1 Finding the Optimal Power Configuration 13

4.1 Graph Representation of the Mathematical Description 16
4.2 Graph Representation with Sleep Modes 19

5.1 Comparison of the Solver Performance for the QP and the ILPs 25
5.2 Ratio between Time needed by Solver for QP and ILPs 26
5.3 Time of Sum of ILPs 27
5.4 Time of QP 28
5.5 Time Ratio of Configurations per Phase for ILPs 28
5.6 Time Ratio of Configurations per Phase for QP 28
5.7 Time Ratio of Phases for ILPs 29
5.8 Time Ratio of Phases for QP 29

6.1 Execution Times compared to PLATIN's WCET Analysis 37

7.1 Measurement Setup 40
7.2 Active Phases 42
7.3 SPI Test 43
7.4 LED Test 44
7.5 I2C at 160 MHz 45
7.6 I2C with varying CPU Frequency 45
7.7 Light Sleep Transitions from and to 160 MHz 48
7.8 Deep Sleep Transitions from and to 160 MHz 50
7.9 Solver Results for Test Program with One Task 53
7.10 Graph of Peripheral Test 54
7.11 Solver Results for Listing 7.2 with I2C Communication 56

LIST OF TABLES

6.1 TACLeBench Benchmarks 36

7.1 Current Consumption Values of the ESP32-C3 42
7.2 Time and Current Consumption for Clock-Tree Reconfigurations 47

LIST OF LISTINGS

2.1 BubbleSort Implementation 4

3.1 Example Task for Problem Description 12

6.1 Example of a Linker Fragment File 32
6.2 Performance Counter Measurement 35

7.1 Iterative Fibonacci Calculation 51
7.2 Peripheral Test 55

REFERENCES

[1] Neil Audsley, Ken Tindell, and Alan Burns. "The End Of The Line For Static Cyclic Scheduling?" In: *In Proc. 5th Euromicro Workshop on Real-Time Systems*. Society Press, 1993, pp. 36–41.

[2] Neil Audsley et al. "Applying New Scheduling Theory to Static Priority Pre-Emptive Scheduling." In: *Software Engineering Journal* 8 (1993), pp. 284–292.

[3] Clément Ballabriga and Hugues Cassé. "Improving the WCET computation time by IPET using control flow graph partitioning." In: *8th International Workshop on Worst-Case Execution Time Analysis (WCET'08)*. Vol. 8. OpenAccess Series in Informatics (OASIcs). 2008.

[4] Mario Bambagini et al. "Energy Management for Tiny Real-Time Kernels." In: *International Conference on Energy Aware Computing*. Nov. 2011.

[5] Mario Bambagini et al. "Energy-Aware Scheduling for Real-Time Systems: A Survey." In: *ACM Trans. Embed. Comput. Syst.* 15.1 (Jan. 2016), pp. 1–6.

[6] Franck Cassez, René Hansen, and Mads Chr. Olesen. "What is a Timing Anomaly?" In: *OpenAccess Series in Informatics* 23 (Jan. 2012).

[7] Holly Chiang et al. "Power Clocks: Dynamic Multi-Clock Management for Embedded Systems." In: *Proceedings of the International Conference on Embedded Wireless Systems and Networks (EWSN '21)*. Feb. 2021, pp. 139–150.

[8] *ESP32-C3 ESP-IDF Programming Guide*. Version Release v5.0-dev-2586-ga82e6e63d9. Espressif Systems (Shanghai) Co., Ltd. Apr. 2022. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/esp-idf-en-v5.0-dev-2586-ga82e6e63d9-esp32c3.pdf (visited on 05/03/2022).

[9] *ESP32-C3 Series Datasheet*. Version 1.2. Espressif Systems. 2022. URL: https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf (visited on 04/28/2022).

[10] *ESP32-C3 Technical Reference Manual*. Version Pre-release v0.6. Espressif Systems. 2022. URL: https://www.espressif.com/sites/default/files/documentation/esp32-c3_technical_reference_manual_en.pdf (visited on 04/28/2022).

[11] *ESP32-C3-Mini-1 Datasheet*. Version 1.0. Espressif Systems. 2021. URL: https://www.espressif.com/sites/default/files/documentation/esp32-c3-mini-1_datasheet_en.pdf (visited on 04/28/2022).

[12] Heiko Falk et al. "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research." In: *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. Vol. 55. OpenAccess Series in Informatics (OASIcs). 2016, 2:1–2:10.

[13] Christian Ferdinand and Reinhold Heckmann. “ait: Worst-Case Execution Time Prediction by Static Program Analysis.” In: *Building the Information Society*. 2004, pp. 377–383.

[14] *Gurobi Website*. 2022. URL: https://www.gurobi.com/ (visited on 06/02/2022).

[15] Reinhold Heckmann et al. “The influence of processor architecture on the design and the results of WCET tools.” In: *Proceedings of the IEEE* 91 (Aug. 2003), pp. 1038–1054.

[16] Henriette Hofmeier. “Timing Analysis for the RISC-V Architecture.” Bachelor’s thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, May 2019.

[17] Yau-Tsun Steven Li and Sharad Malik. “Performance Analysis of Embedded Software Using Implicit Path Enumeration.” In: *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference*. 1995, pp. 456–461.

[18] Jane W. S. Liu. *Real-Time Systems*. 2000.

[19] Paul Lokuciejewski et al. “Approximating Pareto optimal compiler optimization sequences–a trade-off between WCET, ACET and code size.” In: *Software: Practice and Experience* 41.12 (2011), pp. 1437–1458.

[20] Paul Lokuciejewski et al. “Multi-objective Exploration of Compiler Optimizations for Real-Time Systems.” In: *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC ’10)*. 2010, pp. 115–122.

[21] *lp_solve reference guide*. 2022. URL: http://lpsolve.sourceforge.net/ (visited on 06/02/2022).

[22] Thomas Lundqvist and Per Stenström. “Timing anomalies in dynamically scheduled microprocessors.” In: *Proceedings of the 20th IEEE Real-Time Systems Symposium* (Dec. 1999), pp. 12–21.

[23] *Memory FRAM 64K - MB85RS64V*. 2022. URL: https://cdn-shop.adafruit.com/datasheets/MB85RS64V-DS501-00015-4v0-E.pdf (visited on 09/26/2022).

[24] *MSO4000 and DSO4000 Series Digital Phosphor Oscilloscopes - User Manual*. 2022. URL: https://download.tek.com/manual/071212104web.pdf (visited on 09/26/2022).

[25] Kateryna Muts and Heiko Falk. “Multi-Criteria Function Inlining for Hard Real-Time Systems.” In: *Proceedings of the 28th International Conference on Real-Time Networks and Systems (RTNS ’20)*. 2020, pp. 56–66.

[26] James Pallister et al. “Data dependent energy modelling: A worst case perspective.” In: *CoRR* (May 2015).

[27] *PCF8574 Remote 8-Bit I/O Expander for I2C Bus Datasheet (Rev. J)*. 2022. URL: https://www.ti.com/lit/ds/symlink/pcf8574.pdf (visited on 09/26/2022).

[28] Peter Puschner et al. “The T-CREST approach of compiler and WCET-analysis integration.” In: *16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2013* (June 2013), pp. 1–8.

[29] Peter P. Puschner and Anton V. Schedl. “Computing Maximum Task Execution Times — A Graph-Based Approach.” In: *Real-Time Systems* 13 (2004), pp. 67–91.

[30] Phillip Raffeck et al. “Worst-Case Energy-Consumption Analysis by Microarchitecture-Aware Timing Analysis for Device-Driven Cyber-Physical Systems.” In: *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*. Vol. 72. 2019, 4:1–4:12.

[31] Jan Reineke et al. “A Definition and Classification of Timing Anomalies.” In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET’06)*. Vol. 4. 2006.

[32] Michel Rottleuthner, Thomas Schmidt, and Matthias Wählisch. *FlexClock: Generic Clock Reconfiguration for Low-end IoT Devices*. 2021.

[33] Fabian Scheler. "Atomic Basic Blocks – Eine Abstraktion für die gezielte Manipulation der Echtzeitsystemarchitektur." In: *Ausgezeichnete Informatikdissertationen 2011*. 2011, pp. 181–190.

[34] Fabian Scheler and Wolfgang Schröder-Preikschat. "The Real-Time Systems Compiler: migrating event-triggered systems to time-triggered systems." In: *Software: Practice and Experience* 41.12 (2011), pp. 1491–1515.

[35] Martin Schoeberl et al. *Patmos Reference Handbook*. Tech. rep. Feb. 2020. URL: http://patmos.compute.dtu.dk/patmos_handbook.pdf (visited on 09/28/2022).

[36] Martin Schoeberl et al. "Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach." In: *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*. Vol. 18. OpenAccess Series in Informatics (OASIcs). 2011, pp. 11–21.

[37] *TACLe Benchmarks*. 2022. URL: https://github.com/tacle/tacle-bench (visited on 09/06/2022).

[38] *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. Version 20191213. CS Division, EECS Department, University of California, Berkeley. Dec. 2019. URL: https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf (visited on 04/28/2022).

[39] Valentin Touzeau, Claire Maïza, and David Monniaux. "Model Checking of Cache for WCET Analysis Refinement." In: *ArXiv* (2017).

[40] Mike Turquette. *The Common Clk Framework*. Linux Kernel Documentation. 2021. URL: https://www.kernel.org/doc/Documentation/clk.txt (visited on 09/28/2022).

[41] Peter Wägemann et al. "Whole-System Worst-Case Energy-Consumption Analysis for Energy-Constrained Real-Time Systems." In: *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Vol. 106. 2018, 24:1–24:25.

[42] *WCET project / SWEET*. 2013. URL: http://www.mrtc.mdh.se/projects/wcet/sweet.html (visited on 08/17/2022).

[43] *Welcome to Python.org*. 2022. URL: https://www.python.org/ (visited on 08/31/2022).

[44] Reinhard Wilhelm et al. "The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools." In: *ACM Transactions on Embedded Computing Systems* 7.3 (May 2008).