FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Tobias Häberlein

Watwa-OS: A Minimalistic, Statically Analyzable Operating System for Resource-Constrained Real-Time Systems

Masterarbeit im Fach Informatik

30. Oktober 2024

Please cite as:
Tobias Häberlein, "Watwa-OS: A Minimalistic, Statically Analyzable Operating System for Resource-Constrained Real-Time Systems", Master's Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, October 2024.

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

WATWA-OS: A Minimalistic, Statically Analyzable Operating System for Resource-Constrained Real-Time Systems

Masterarbeit im Fach Informatik

vorgelegt von

Tobias Häberlein

geb. am 23. November 2000
in Lauf a.d. Pegnitz

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Dr.-Ing. Peter Wägemann** **Eva Dengler, M.Sc.**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. Rüdiger Kapitza**
Beginn der Arbeit:	**01. Mai 2024**
Abgabe der Arbeit:	**30. Oktober 2024**

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.

(Tobias Häberlein)
Erlangen,
30. Oktober 2024

ABSTRACT

In the domain of embedded real-time systems, energy efficiency plays an important role, as the systems used often rely on small batteries or have no reliable power source at all. This is particularly the case for systems that obtain their energy from the environment, for example, via solar cells. In order to improve energy efficiency, many new microcontroller units, therefore, offer a complex, interconnected clock configuration network called the *clock tree*. Using this clock tree, it is possible to control the frequency of every connected component and, thus, significantly influence the overall power consumption.

Adapting the clock configuration to the current workload can have a big impact on energy efficiency. Dynamic approaches already exist that automatically adjust the clock configuration during system runtime. However, they can only approximate at which point a clock-configuration change actually saves energy. Furthermore, these approaches often do not take into account the transitioning costs of switching from one clock configuration to another. WATWA-OS is a new approach that solves this problem statically, i.e., before system runtime. The approach extends an existing mechanism for static, whole-system energy analysis, called SysWCEC, and adds the ability to optimize the system's worst-case energy consumption automatically. WATWA-OS considers the impact of clock-configuration changes on all subsequent system states and can thus determine whether a configuration transition at a given point saves energy or not. In order to achieve optimal solutions, it solves several mathematical models, each representing a different possibility of clock configurations. As a result, the presented approach yields a worst-case-optimal sequence of clock configurations that minimizes the system's worst-case energy demand. WATWA-OS uses this information to create optimized instances of an operating system that automatically switch to different clock configurations at the determined points. The evaluation results show that the calculated clock configuration points are close to real measured results and can be improved by tighter upper bounds on the system's energy consumption.

KURZFASSUNG

In der Domäne eingebetteter Echtzeitsysteme spielt die Energieeffizienz eine wichtige Rolle, da die eingesetzten Systeme oft entweder auf kleine Batterien angewiesen sind oder keine zuverlässige Stromzufuhr besitzen. Dies gilt besonders für Systeme, die ihre Energie aus der Umgebung beziehen, beispielsweise über Solarzellen. Um die Energieeffizienz zu verbessern, bieten daher viele neue eingebettete Systeme ein komplexes, verflochtenes Takt-Konfigurationsnetzwerk an. Über dieses Netzwerk ist es möglich, die Frequenz aller angeschlossenen Systemkomponenten zu steuern und damit den Gesamtstromverbrauch erheblich zu beeinflussen.

Das Anpassen der Taktkonfiguration an die aktuell ausgeführten Operationen kann große Auswirkungen auf die Energieeffizienz eines Systems haben. Es existieren bereits dynamische Ansätze, die die Taktkonfiguration zur Laufzeit des Systems anpassen können. Sie können allerdings nur schätzen, ob sich ein Konfigurationswechsel an einem Punkt wirklich lohnt. Außerdem vernachlässigen diese Ansätze oft Kosten, die durch den Wechsel einer Taktkonfiguration entstehen. WATWA-OS ist ein neuer Ansatz, der diese Probleme statisch löst, also vor der Laufzeit des Systems. Es wird ein bereits existierender Mechanismus zur statischen, ganzheitlichen Systemanalyse namens SysWCEC erweitert und um die Möglichkeit ergänzt, den worst-case Energieverbrauch des Systems automatisch zu optimieren. WATWA-OS berücksichtigt die Auswirkungen eines Takt-Konfigurationswechsels auf alle nachfolgenden Systemzustände und kann dadurch ermitteln, ob durch den Wechsel tatsächlich Energie gespart werden kann. Dafür werden mehrere mathematische Modelle gelöst, wobei jedes Modell eine andere Möglichkeit an Taktkonfigurationen repräsentiert. Das Ergebnis ist eine optimale Sequenz von Taktkonfigurationen, die den worst-case Energieverbrauch des Systems minimiert. WATWA-OS verwendet diese Information, um optimierte Instanzen eines Betriebssystems zu erzeugen, die zur Laufzeit automatisch zur besten ermittelten Taktkonfiguration wechseln. Die Evaluationsergebnisse zeigen, dass die ermittelten Taktkonfigurationen nah an die tatsächlich gemessenen optimalen Konfigurationen herankommen, und dass sich diese Ergebnisse durch die Angabe besserer oberer Schranken für den Energieverbrauch verbessern lassen.

CONTENTS

1 INTRODUCTION

Today, more and more small energy-constrained embedded systems are deployed as part of the Internet of Things (IoT). In this domain, energy efficiency plays an important role, as the used Microcontroller Units (MCUs) often either rely on small batteries or have no reliable power supply at all, e.g., when they harvest energy from their environment using solar cells [20]. In order to conserve energy, many new MCUs therefore offer a complex clock network that can be configured by the application developer to fit the needs of the system [7, 23]. An important control knob of this clock network, also sometimes referred to as the clock tree of a system, is the CPU frequency. Along with other external peripherals, the CPU is typically a major power consumer in an embedded system. Choosing an adequate clock frequency based on the current workload of a system can therefore significantly reduce its energy-related demands. A key observation to consider when selecting an energy-optimal clock frequency is that lower clock frequencies are generally better suited for I/O workloads (e.g., communication over slow bus protocols such as UART or SPI), while higher clock frequencies are better suited for compute workloads [2]. However, multiple challenges arise which make selecting the optimal frequency a non-trivial task: (1) Peripheral devices may impose certain restrictions on the minimum clock frequency in order to function properly. They may even restrict the clock source (i.e., the oscillator), e.g., if frequency stability is a concern. (2) Transitioning from one clock configuration to another one can incur substantial timing and energy-related costs, which consist of a software and a hardware part. On the software side, registers need to be written and kernel subsystems may need to be reinitialized. On the hardware side, the configuration adjustment takes time and may require a stabilization phase. Because of these costs, it is not always more efficient to switch to a low-frequency configuration before performing I/O tasks, or to switch to a high-frequency configuration before performing compute-intensive tasks, especially if these tasks only take a short amount of time. A common strategy chosen by application developers is to use one static clock configuration for all tasks that can satisfy all peripheral constraints. Either the highest possible frequency is selected (*race-to-idle*, [14]), or the lowest possible frequency is selected. This way, the challenging task of determining points at which a clock-configuration change can save energy is avoided. However, since IoT applications typically consist of both I/O- and compute-intensive tasks, these approaches usually do not yield an optimal (i.e., minimal) energy demand.

Several solutions already exist which try to help the application developer by taking over the task of selecting and switching to different clock configurations [21, 2]. The main idea behind them is to integrate a frequency decision mechanism into the kernel of the Real-Time Operating System (RTOS). By default, the RTOS selects the highest possible frequency. If a thread is performing I/O tasks for a long enough time, the kernel will automatically switch to the lowest possible frequency allowed by the peripherals in use. A major disadvantage of these *online* approaches is that they are executed at

system runtime. Since they do not know how long an I/O operation will take, they can only speculate as to whether a frequency transition is worthwhile. As a result, they cannot guarantee that switching to a different clock frequency will actually reduce the application's energy consumption. In addition, these solutions do not take into account the presence of interrupts and their scheduling-related effects. For example, I/O tasks can be preempted by interrupts that cause a compute-intensive task to be executed.

In order to make optimal clock configuration decisions, a *static* approach is required that is able to determine ahead of system runtime if and when a configuration change is beneficial. This requires a whole-system perspective that considers the operating system semantics, including all possible preemptions and interrupts that may occur during system runtime. Existing static approaches are already able to optimize the Worst-Case Response Energy Consumption (WCRE) of simple periodic task sets, but so far none are able to determine optimal solutions for more advanced real-time operating systems with the presence of multiple interrupts [3].

This thesis introduces WATWA-OS, a framework for static WCRE analysis and automatic clock configuration optimization for embedded real-time systems. It takes an existing static, whole-system WCRE analysis technique called SysWCEC [25] and extends it by adding multiple decision points into the system at which it is possible to switch to another clock configuration. WATWA-OS constructs and solves formalization problems for each configuration possibility and chooses the one with the minimal WCRE. It is then able to use this information to generate an OS instance that automatically switches clock configurations at the determined points, without the involvement of the application developer.

In the following, Chapter 2 introduces the necessary background knowledge for WATWA-OS. It focuses on clock trees and on the power consumption of modern embedded systems, and then introduces the whole-system analysis technique SysWCEC, which is the underlying framework on which WATWA-OS is built. Chapter 3 presents the contributions of WATWA-OS and how SysWCEC has been modified to be able to optimize the WCRE of a system. The implementation is discussed in Chapter 4, which focuses on the tools used and the integration of the presented mechanism into them. In Chapter 5, WATWA-OS is evaluated on a RISC-based embedded hardware platform. Chapter 6 discusses the evaluation results and gives insight into how WATWA-OS could be improved in future work, before concluding with a summary in Chapter 7.

2 BACKGROUND AND RELATED WORK

This chapter provides the necessary background knowledge to understand the details of WATWA-OS and its operation. Sections 2.1 and 2.2 discuss the power consumption of modern embedded systems, the main components involved, and how dynamic optimization mechanisms try to minimize energy consumption at system runtime. Section 2.3 introduces SysWCEC, a whole-system analysis technique that builds the basis for WATWA-OS' static optimization approach.

2.1 Clock Trees

To save energy, modern embedded hardware platforms offer a wide range of frequency configuration options for each clock-dependent component [7, 23]. These are not only limited to the processor but also include internal and external peripheral devices that may be connected to the system. A major challenge with these platforms involves the modification of clock configurations. This is because changing the clock frequency of a device usually involves the configuration of several interdependent and interconnected components such as oscillators, multipliers, or frequency gates. Moreover, the frequency configuration of one device can even affect the clock frequency of one or many other devices.

The totality of all connected clock-related components of the system makes up the so-called *clock tree* of a system. An example of how such a clock tree might look like is depicted in Figure 2.1. A *source node* marks the start of each clock signal within the clock tree. This is either an internal or

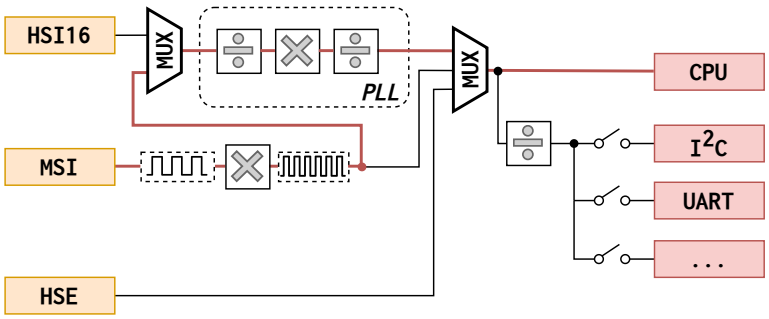


Figure 2.1 – Simplified example of a clock tree, based on the STM32L47x platform [22]. The clock signal is generated by **source nodes** on the left and passes through various scalers, multiplexers, and gates before finally arriving at the **consumer nodes** on the right.

external oscillator. The clock signal then passes through various intermediate nodes before arriving at the consumer node, i.e., the device:

1. Multiplexer nodes (**MUX**) accept multiple input signals and forward exactly one of these signals to their output port.
2. Multiplier () and divider nodes () alter the input frequency by a configurable factor.
3. Gates () either forward the clock signal to their output or block it. They are a valuable component for saving energy, as they allow devices to be turned off when they are not in use.

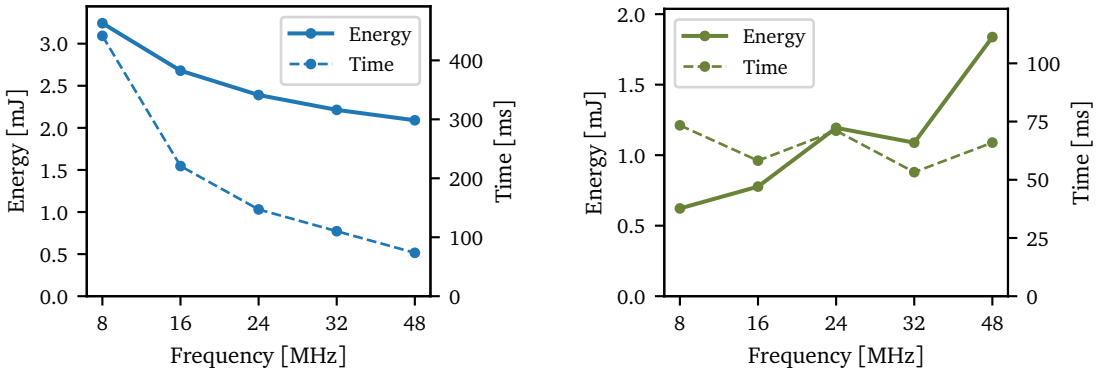
The example above illustrates one of the challenges of complex clock configuration networks: The I^2C and UART peripherals use the same clock signal that is being used for the processor of the system. Consequently, changing the processor frequency cannot be done in isolation, but all possible affected peripherals that depend on the same clock signal must be taken into account. For example, it may be necessary to recalculate and update the divider values that the UART controller of the system uses to derive a specific baud rate. On the other hand, the UART or I^2C controller may impose additional constraints, such as a minimum or maximum frequency, further complicating the selection of an appropriate clock configuration. One approach to overcoming these challenges would be to use independent clock signals for the CPU frequency and the peripherals. This, however, requires the activation of additional components in the clock tree, such as a different oscillator, which ultimately results in higher energy consumption.

Other important aspects that must be considered when switching between different clock configurations are timing- and thus energy-related transitioning costs, typically consisting of software and hardware parts. On the software side, memory-mapped registers need to be written, and it may be necessary to update OS kernel data structures, which depend on the clock frequency. Furthermore, recalibrations might be necessary, such as in the timer subsystem. On the hardware side, updating the actual register values and then transitioning to a new frequency can be a time-consuming task, especially when Phase-Locked Loops (PLLs) are involved, as these types of components typically require a frequency stabilization phase. For example, the MSI oscillator in Figure 2.1 has a stabilization time of up to 2.5 ms on the STM32L476xx microcontroller [22]. In the most extreme cases, the transition from one clock configuration to another may not even be possible without using a temporary intermediate configuration, which adds yet another time-consuming intermediate step.

Due to the above reasons, some real-time operating systems only provide partial dynamic clock configuration possibilities (e.g., ChibiOS and Zephyr). As a result, a common strategy is to determine one suitable clock configuration ahead of system runtime, which is then used throughout the entire application. This decreases the complexity of keeping track of different dependencies but is also a less flexible approach that ultimately sacrifices energy efficiency, as will further be discussed in Section 2.2. However, ideas exist to integrate complex clock configuration networks into the operating system kernel. ScaleClock by Rottleuthner et al. [21] is such a system that tries to map the previously presented hardware components of the clock tree (e.g., multiplexers, gates) in software. It provides a service that is able to automatically switch between different clock configurations at system runtime and even uses temporary intermediate clock configurations when necessary.

2.2 Power Consumption

The previous section explained that there are various ways to adjust the clock frequency of a system. The question now arises as to how this can save energy in an embedded system, which is the subject of this section.



(a) Compute-intensive workload

(b) I/O-intensive workload

Figure 2.2 – Comparison of the energy consumption for different CPU frequencies on different types of workloads on a Nucleo-L476RG [24]

2.2.1 Static and Dynamic Power

The power consumption of CMOS chips in microcontrollers consists of a static and a dynamic part [1]. Dynamic power consumption results from switching activity within the system that causes capacitors to charge and discharge. It depends on the current clock frequency being used. Static power consumption, on the other hand, is caused by leakage currents through the transistors. It occurs even when the transistors are not actively switching and is independent of the frequency.

An embedded system can save energy by dynamically adapting the clock frequency to the current type of operations being performed [2, 26]. In principle, a distinction should be made between two types of operations: compute-intensive workloads and I/O-intensive workloads. Compute-intensive tasks should be executed at the highest possible clock frequency. This way, the dynamic power consumption is high. Still, as the task is completed in the shortest possible time, the static energy consumption is low because the time spent per operation is minimized. On the other hand, I/O-intensive operations, such as communicating with external sensors via a bus protocol like UART, should be performed at the lowest possible clock frequency. In this case, the dynamic part of the energy consumption is minimal due to the low frequency. Choosing a higher clock frequency for I/O operations will not improve energy efficiency because the static portion of the energy consumption will remain the same as for compute-intensive workloads. This is because I/O operations are usually bottlenecked by a relatively slow bus protocol compared to the CPU clock frequency.

Figure 2.2 shows the difference in energy consumption at different clock frequencies for (a) a compute-intensive workload and (b) an I/O-intensive workload. The evaluation was performed on an STM32-based Nucleo-L476RG development board. For the compute-intensive workload, 10,000 multiplications were performed in a for-loop. For the I/O-intensive workload, the value of an internal temperature sensor connected via SPI has been read out continuously. As expected, increasing the clock frequency for workload (a) reduces the system's energy demands because the task can be completed more quickly. This strategy is also known as racing-to-idle. For workload (b), increasing the clock frequency does not provide the benefit of faster execution times since the task has to wait for SPI bus transactions to complete. In this case, the lowest possible energy consumption is achieved at the lowest clock frequency.

2.2.2 Dynamic Frequency Scaling

The process of adapting the clock frequency to the current system load is called Dynamic Frequency Scaling (DFS). Several approaches for small embedded systems already exist that implement DFS mechanisms in the operating system kernel.

One such approach is presented by Chiang et al. and called *Power Clocks* [2]. Power Clocks is a mechanism implemented in the scheduler of the real-time operating system that automatically switches to a low clock frequency when I/O operations are performed. The operating system is able to track current peripheral operations by detecting driver function calls. If the I/O operation takes longer than a predefined minimum amount of time, the OS switches to a low frequency. After the driver call has finished, it switches back to a high clock frequency. A particular advantage of Power Clocks is that it can also take into account the different requirements and constraints imposed by the peripheral drivers, such as minimum or maximum frequencies. This allows the OS to select the most appropriate and permissible clock frequency from a list of pre-selected configurations.

A different approach presented by Rottleuthner et al. called *ScaleClock* [21] takes care of frequency scaling on a per-thread basis. In an initialization phase, ScaleClock tries out every application thread at different clock frequencies. It then analyzes how the execution time of a thread scales in comparison to the clock frequency. If the execution time only slightly decreases or does not decrease at all as the clock frequency increases, then the thread should be executed at a low clock frequency. This is the case for threads running I/O-intensive workloads, as discussed in the previous section. Otherwise, if the thread execution time scales well with the clock frequency, the thread will be executed at a high clock frequency. One advantage of ScaleClock over Power Clocks is that it is a more general approach that considers not only peripheral driver function calls but also looks at frequency scaling of a thread depending on its execution time. Compared to Power Clocks, though, the frequency scaling at thread granularity does not result in optimal energy consumption when threads consist of both compute-intensive *and* I/O-intensive workloads.

A major drawback common to both presented approaches is that they do not adequately consider the transition costs of switching from one clock configuration to another one. As discussed earlier, the timing delays of switching between clock configurations can have a significant impact on the energy consumption of the system. As a result, it may not always be better to switch to a lower frequency if the subsequent workload is not long enough to amortize the transition costs. Power Clocks attempts to work around this issue by waiting a certain amount of time before switching to another clock configuration, assuming that it is likely to be worthwhile and that the workload will continue for a longer period of time. However, since none of these approaches knows exactly how long an I/O or compute task will take, they can only estimate whether or not a clock-configuration change is reasonable.

To accurately predict the impact of a clock-configuration change on the energy consumption of a real-time system, a static analysis of all system states is required. It is not sufficient to look at a block of I/O- or compute-intensive operations in isolation, since these operations may be part of a thread that can be preempted or interrupted. Instead, a whole-system analysis is necessary that considers the impact of thread preemption, including the overhead of OS function calls, to determine if and when a clock-configuration change is actually useful. This is where WATWA-OS comes into play: It uses an existing whole-system analysis technique called SysWCEC [25], which is able to give an upper bound on the Worst-Case Response Energy Consumption (WCRE) of a real-time system. WATWA-OS extends this technique by analyzing the effect of different frequency decisions and then choosing the best, i.e., minimal, WCRE bound and its corresponding clock configurations in order to optimize energy consumption.

2.2.3 Static Energy Optimization

There already exist static approaches that try to optimize the energy consumption of an embedded system by finding optimal sequences of clock configurations. One such approach is CRÊPE presented by Dengler and Wägemann [3]. It formulates a mathematical optimization problem in order to select energy-minimal clock configurations while still meeting given deadlines. CRÊPE can find worst-case optimal clock configurations for systems with a periodic task model and with, at most, one preempting Interrupt Service Routine (ISR). In addition, it is able to conserve energy by switching to a low-power sleep state during idle phases. Another advantage of CRÊPE is that it can defer the execution of an ISR. For example, if the system is located in a high-power state, it may be more energy efficient to postpone the ISR execution and wait until the system switches to a low-power clock configuration.

While CRÊPE offers many advantages over the dynamic approaches presented in the previous section, its applicability is limited by the restrictive task model and the maximum number of ISRs allowed. WATWA-OS is similar to CRÊPE as it formulates mathematical optimization problems to find a worst-case optimal sequence of clock configurations. Unlike CRÊPE, WATWA-OS is a more generic approach that allows for a less restrictive task model and supports multiple interrupts. However, WATWA-OS does not yet consider sleep states or the deferred execution of ISRs.

2.3 SysWCEC

This section introduces SysWCEC [25], a static WCRE analysis technique for entire real-time systems that considers all active (peripheral) devices and their states in order to give better worst-case energy bounds. WATWA-OS uses this mechanism and extends it to support multiple clock-configuration decision points, at which either a high or a low frequency can be chosen.

2.3.1 Worst-Case Response Energy Consumption

A key challenge for energy-constrained real-time systems is ensuring enough energy is available to complete a given task set. Calculating the Worst-Case Response Energy Consumption (WCRE) of a system before its runtime is a common way of sizing the necessary power supply. The Worst-Case Response Energy Consumption (WCRE) of a task, in this case, includes the Worst-Case Energy Consumption (WCEC) of the task itself, including all other OS-related interruptions and preemptions that might occur during its runtime. A common way of calculating the WCRE of a task is to use a bottom-up approach: This includes calculating the WCEC of the task itself and of all other tasks and interrupts that can theoretically occur during its runtime and then summing up these energy consumptions. For the WCEC of one task, it is assumed that all system devices (i.e., the processor and all peripherals) are always on and in their highest power state. This technique provides a safe upper bound on the Worst-Case Response Energy Consumption (WCRE) of a task. However, it also has two major drawbacks that cause over-approximations:

1. In real systems, peripheral devices are only in their highest power state for a short period of time. Transceivers, for example, only use the most power during periods of communication. As these devices can consume a lot of power, even more than the CPU itself, this all-always-on approach overestimates power consumption by a significant factor.

2. The summation of all WCECs of all possible preemptions does not consider which scenarios are actually possible. For example, in an operating system with simple fixed-priority preemptive scheduling, a task can only be preempted by higher-priority tasks. Therefore, the WCEC of all lower-priority tasks would not have to be taken into account.

2.3.2 Minimizing Over-Approximations

SysWCEC is a context-sensitive approach that addresses the abovementioned issues by considering the entire system, including all possible power states. Its main idea is to construct a state-transition graph that includes all possible system execution paths and power states. The semantics of OS system calls are integrated into the construction algorithm, which, therefore, only considers feasible execution paths. SysWCEC then uses the Implicit Path Enumeration Technique (IPET) [16, 18] to transform the state-transition graph into a mathematical optimization problem, which can be solved by a mathematical solver to obtain the WCRE.

To ensure reasonable solving times and to keep the number of system states from exploding, SysWCEC only considers small embedded systems, where energy is a limited resource. These systems consist of a single-core processor that does not offer instruction or data caches and only has a few pipeline stages. Another requirement of SysWCEC that these systems meet is the absence of timing anomalies. This ensures that the WCEC of two non-preemptable tasks can safely be added together to obtain their combined energy demand, i.e., $\mathrm{WCEC}(t_1 + t_2) = \mathrm{WCEC}(t_1) + \mathrm{WCEC}(t_2)$. Finally, the Real-Time Operating System (RTOS) and its scheduling-related behavior have to be reasonably deterministic in order to keep the number of possible execution paths through the system to a minimum and thus ensure low WCRE bounds. Since WATWA-OS is based on the SysWCEC approach, it imposes the same requirements on the real-time systems to be analyzed. However, the system requirements do not limit the approach's applicability, since most small real-time systems, especially those with limited energy budgets, already meet the imposed constraints. This includes the hardware constraints as well as the constraints on the RTOS, whose scheduler is often required to be deterministic by relevant industry standards (e.g., OSEK, Autosar, POSIX.4). The OSEK standard [17], for example, mandates a fixed-priority preemptive scheduling strategy, which is also implemented in the OS instances generated by WATWA-OS.

2.3.3 Abstracting Source Code Basic Blocks

In order to analyze all possible execution paths of the system, SysWCEC needs a Control-Flow Graph (CFG) that models all application and OS code, as well as task interdependencies. Analyzing every basic block of the CFG generated by the compiler or even every instruction, however, would be overly fine granular and ultimately result in optimization formulations with too many variables. Instead, SysWCEC introduces a new abstraction and uses a graph that consists of so-called Power Atomic Basic Blocks (PABBs). PABBs consist of multiple basic blocks but must always have exactly one entry and one exit block. The idea is that inside one PABB, the system's maximum power consumption does not change, i.e., the power state of all devices inside the system stays the same. A device, in this case, refers to any power-consuming component inside the embedded system, including the processor. The power state of a device can only be changed by a *device system call*, or device syscall for short. These device syscalls are typically the last basic block within a PABB. PABBs may be split arbitrarily for optimization purposes as long as the above requirements are still met. WATWA-OS, for example, also splits PABBs at scheduling-related syscalls, like task activations or terminations.

Figure 2.3 shows how the basic blocks of a CFG for one task can be grouped into PABBs. `PABB1` contains the first four basic blocks. The exit block in `PABB1` is `BB4`, which calls a device syscall that turns on the UART controller and thus increases the power demands of the system. Therefore, `BB5` has to be placed in a new PABB. Basic Blocks `BB1`-`BB4` could theoretically each be placed in a separate PABB. Still, it is generally advisable to create PABBs with a maximum number of basic blocks to reduce the number of nodes in the graph that have to be considered later in the analysis. It

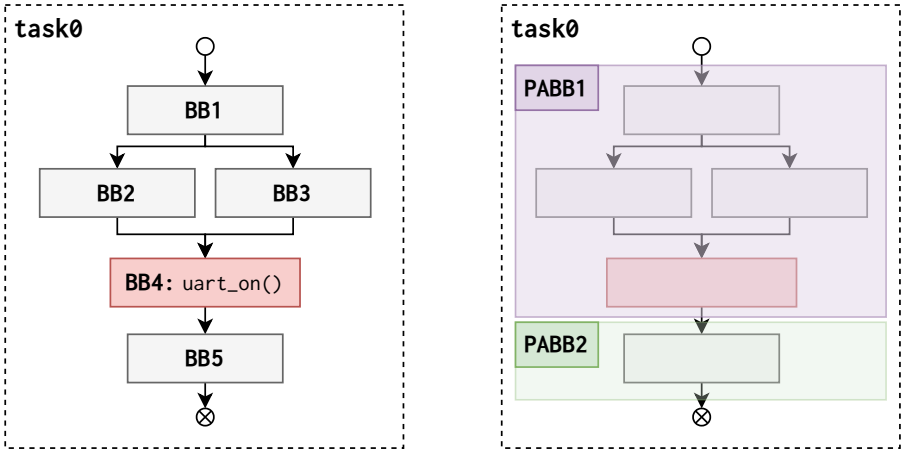


Figure 2.3 – Example of how basic blocks in the original CFG of a task are merged and transformed into multiple PABBs

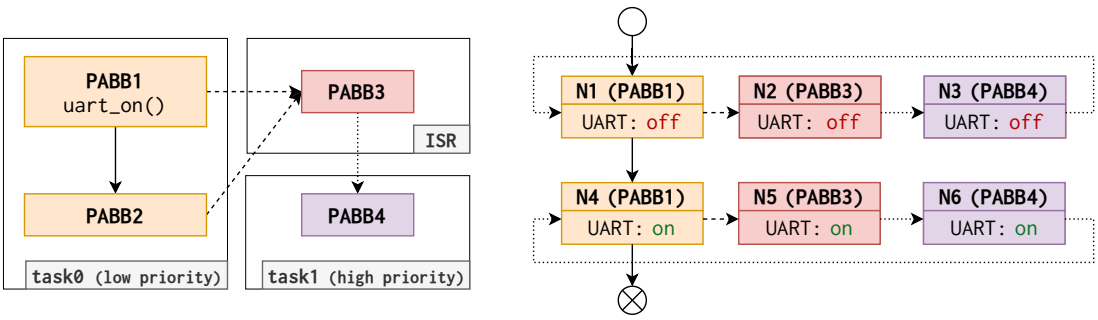
should be noted that this way of grouping multiple basic blocks can even put entire control structures of the original source code into a single PABB. This also includes loops and if-then-else statements (cf., nodes BB2 and BB3 in Figure 2.3).

SysWCEC creates PABBs for every task and function of the real-time system being analyzed. Since it is a whole-system analysis technique, PABBs contain not only the basic blocks of the application, but also the blocks and thus implementation of the operating system kernel. The relations between multiple PABBs are modeled as edges in the system's PABB graph. These include intra- and inter-task relations, such as preemptions by interrupts or activations of higher-priority tasks.

2.3.4 The State-Transition Graph

At this point, the PABB graph of the system is an abstraction from the basic blocks of the application source code that models the interaction between different tasks and interrupts. As a next step, SysWCEC uses the PABBs to create a new graph that contains all possible execution paths and system states, called the Power-State-Transition Graph (PSTG). A node, i.e., system state, in the PSTG includes several pieces of information: (1) the currently executed PABB, (2) the state of all active devices in the system, (3) the current state of the RTOS, including the scheduler state (e.g., runnable tasks and a pointer to the last executed PABB of a task). The PSTG construction starts at a defined entry node with a known system state, which is set up by the OS boot code. From this node, the construction algorithm follows the edges of the PABB graph to create new successor nodes in the PSTG. The construction is context-sensitive, which means that the state information of previously visited nodes is taken into account when creating a new successor node. Furthermore, the semantics of the operating system are implemented into the algorithm. This way, all possible system states and their execution paths are explicitly enumerated, while infeasible paths are avoided, such as the preemption of a high-priority task by a low-priority task.

Figure 2.4 shows how a PSTG is constructed from a PABB graph. The example application consists of one low-priority task with two PABBs that can be interrupted. The Interrupt Service Routine (ISR) causes a higher-priority task to be activated, which will be executed before the ISR returns to the low-priority task. The resulting PSTG in Figure 2.4b contains multiple nodes that execute the same PABB (e.g., N2 + N5), once with the UART controller turned off and once with the



(a) PABB graph **(b)** PSTG

Figure 2.4 – PABB graph and PSTG of an application consisting of a low-priority task and a high-priority task that is activated through an ISR

UART controller turned on. This is important because the different states of the UART controller result in a different maximum power consumption for a node. Further, a distinction is made in the PSTG between different types of edges. Local edges (solid lines) connect two nodes that belong to the same task, such as N1→N4. Interrupt edges (dashed lines) connect a normal interruptible node to the entry node of an interrupt (cf., N1→N3). Scheduling edges (dotted lines) connect two nodes of different tasks, e.g., when a higher-priority task is activated, or the current task terminates its execution (cf., N3→N1). This distinction is useful when formulating a mathematical optimization problem in the next step.

2.3.5 ILP Formulation

At this point, the PSTG contains all feasible states and execution paths that can occur in the system. In the next step, this graph is transformed into an Integer Linear Program (ILP) by using the Implicit Path Enumeration Technique (IPET) [16, 18]. The ILP is a mathematical optimization problem that seeks to find the Worst-Case Response Energy Consumption of the real-time system. IPET refers to a technique that essentially transforms all nodes and edges of the graph into variables and constraints in the ILP.

The idea is that each node $v \in V$ of the PSTG gets assigned a variable in the ILP that corresponds to its execution frequency $f(v)$. In SysWCEC, graph edges are also allowed to contribute to the system's total energy consumption, so each edge $e \in \mathscr{E}$ also gets a frequency variable. The objective of the ILP is then to maximize the Worst-Case Energy Consumption *WCEC* of all nodes and edges multiplied by their execution frequency:

$$WCRE = \max\Big(\sum_{v \in V} WCEC(v) \cdot f(v)\Big) + \max\Big(\sum_{e \in \mathscr{E}} WCEC(e) \cdot f(e)\Big) \tag{2.1}$$

To obtain the *WCEC* for a single node, an energy model must be defined. For the small real-time systems used in SysWCEC and WATWA-OS, the Worst-Case Execution Time (WCET) multiplied by the maximum power consumption of a node is used to calculate its WCEC:

$$WCEC(v) = WCET(v) \cdot P_{max}(v) \tag{2.2}$$

The maximum power consumption P_{max} of a node depends on the power state of all active devices. It can be determined using accurate measurements, or by consulting the documentation of the hardware platform. For example, node N1 in Figure 2.4b has a higher power consumption

than node N4, as the enabled UART controller increases the system's overall power consumption. The WCET can be determined using existing techniques, e.g., by combining the execution time of the instructions being executed in the PABB of a node. The execution time of an instruction also depends on the power state of the system, more specifically the CPU clock frequency.

Structural Constraints

So far, the ILP is unbounded since each node and edge can be executed an infinite number of times. To bound the ILP, structural constraints have to be added that model to the relations between different nodes and edges in the PSTG. The most important constraint is that the combined frequency of all incoming edges of a node must be equal to the frequency of all outgoing edges of a node. This sum must also be equal to the frequency of the node itself:

$$\forall v \in V: \quad \sum_{e \in \mathcal{E}_{incoming,v}} f(e) \quad = \quad f(v) \quad = \quad \sum_{e \in \mathcal{E}_{outgoing,v}} f(e) \tag{2.3}$$

This constraint ensures that a node can only be executed if at least one of its incoming edges is executed. More specifically, the node can be executed only exactly as often as all of its incoming edges are executed.

Two special edges are inserted into the ILP: An entry edge that connects to the entry node and an exit edge that connects to the exit node. The frequency of each of these two edges is set to 1. These two constraints ensure that the application is started and exited exactly once.

Basic Interrupt Handling

Interrupts pose a particular challenge when trying to find the WCRE of a real-time system, as they create loops in the PSTG. Without further intervention, these loops cause the ILP to become unbounded. This can be seen in the example PSTG in Figure 2.4b, where the interrupt in PABB3 could theoretically occur an unlimited number of times, causing the upper bound for the energy consumption to also be infinite. In practice, however, interrupts rarely occur in an unpredictable manner, but are usually tied to a minimum Inter-Arrival Time (IAT). This is why in SysWCEC and WATWA-OS, the application developer is able to annotate interrupts with their minimum IAT, resulting in a finite upper WCRE bound. In the ILP, the minimum IAT is used to derive the execution frequency $f(I)$ of an interrupt I. For this, the Worst-Case Response Time $WCRT$ of the system is put in relation to the execution frequency of an interrupt and its minimum Inter-Arrival Time $IAT_{min,I}$.

$$IAT_{min,I} \cdot (f(I) - 1) \leq WCRT \tag{2.4}$$

In essence, this inequality constraint captures the effect that the longer the execution time of the system, the more often an interrupt can occur and vice versa. In the worst case, an interrupt happens right at the start of the system at time $t = 0$, and then again every $IAT_{min,I}$. The Worst-Case Response Time (WCRT) of a system is determined similarly to its WCRE, i.e., by combining the execution frequencies of all nodes and edges and their Worst-Case Execution Times:

$$WCRT = \Big(\sum_{v \in V} WCET(v) \cdot f(v)\Big) + \Big(\sum_{e \in \mathcal{E}} WCET(e) \cdot f(e)\Big) \tag{2.5}$$

It is important to consider that the solution of $WCRT$ is not a general upper bound on the system's Worst-Case Response Time but only in the case of the worst energy consumption. This is because the same frequency variables are used as for the main objective of the ILP. Thus, in some cases, the actual Worst-Case Response Time of the system may be higher than the one determined in this ILP.

Other Interrupt Constraints

All of the ILP constraints presented so far result in a safe upper WCRE bound for the example system presented in Figure 2.4b. However, there is still a problem caused by the interrupt nodes, which leads to an unnecessary over-approximation of the WCRE. Assume that in the solution of the corresponding ILP for Figure 2.4b, interrupt node N2 is executed 5 times. Because of the structural constraints described earlier, node N1 must thus be executed 6 times since the entry edge is taken exactly once, and the return edge from N2 to N1 is taken as often as the interrupt occurs, i.e., 5 times. In reality, though, node N1 is only executed in its entirety once since an interrupt only pauses the execution of a node but continues from the same point that the interruption took place.

This is why SysWCEC extends the existing constraints for the frequency of a node by subtracting all completed interrupt-resume cycles. For this, the previously presented distinction of edges in the PSTG comes in handy. Essentially, the frequency of a node is adapted to the following formula:

$$f(v) \quad = \sum_{e \in \mathcal{E}_{incoming,v}} f(e) \quad - \sum_{e \in \mathcal{E}_{outgoing,v,async}} f(e) \tag{2.6}$$

$\mathcal{E}_{incoming,v}$ is the set of all incoming edges to node v. $\mathcal{E}_{outgoing,v,async}$ only contains the frequency variables for outgoing edges to interrupt entry nodes. In the example above, this would lower the execution frequency of node N1 to:

$$f(\texttt{N1}) = f(\texttt{entry} \rightarrow \texttt{N1}) + f(\texttt{N4} \rightarrow \texttt{N1}) - f(\texttt{N1} \rightarrow \texttt{N3}) = 1 + 5 - 5 = 1$$

Loops in the PSTG

In addition to interrupts, other loops may occur in the PSTG that stem from the application's original control flow graph. This is commonly the case, for example, when a device syscall is called inside a loop. In this case, the CFG basic blocks of the loop cannot be merged into a single PABB or PSTG node, since the power state of the system changes after the device syscall has been executed.

Figure 2.5 shows a possible PSTG that contains a loop. Starting from node N1, either the left path through node N5 or the right path through the loop consisting of the nodes N2, N3 and N4 can be taken. It is apparent from the graph that only either the left path or the right path can be taken. However, with the current ILP constraints, nodes of both paths can still be taken. A possible solution for the corresponding ILP is depicted in the figure. Each node is executed once. As for the edges,

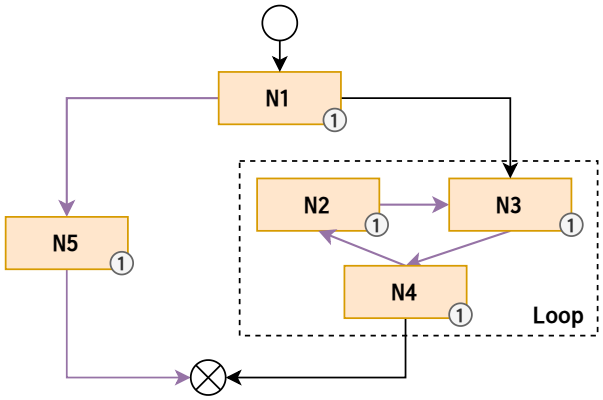


Figure 2.5 – PSTG that contains a loop consisting of nodes N2, N3 and N4. The numbers in circles indicate how many times a node is taken in the solution of the corresponding ILP.

the path from N1 to N5 and from N5 to the exit is taken once. Additionally, each path inside the loop on the right is taken once. The depicted node and edge frequencies are a valid solution for the corresponding ILP. In particular, this solution satisfies all structural constraints imposed, i.e., the execution frequency of the incoming edges is always equal to the execution frequency of the outgoing edges in the graph.

To prevent such cases from happening, a further constraint in the ILP is needed that prevents loops from being executed in certain situations. In SysWCEC, this is achieved with the following constraint:

$$\sum_{e \in \mathscr{E}_{backlink,L}} f(e) \quad \leq \quad M \cdot \sum_{e \in \mathscr{E}_{entry,L}} f(e) \tag{2.7}$$

Basically, for each loop L, the frequency of all backlink edges $\mathscr{E}_{backlink,L}$ must be smaller than the frequency of the entry edges $\mathscr{E}_{entry,L}$ into the loop multiplied by a sufficiently large constant M (SysWCEC uses $M = 10{,}000$). This ensures that a loop is only considered if its entry is taken at least once. If the frequency of the entry edge is 0, then the loop will not be taken. For the example in Figure 2.5, this would add the following constraint:

$$f(\texttt{N2} \rightarrow \texttt{N3}) \leq 10{,}000 \cdot f(\texttt{N1} \rightarrow \texttt{N3}) \tag{2.8}$$

2.3.6 Solving the ILP

Until now, SysWCEC has constructed a PSTG from the original application source code and then transformed this graph into a mathematical optimization problem, an ILP. The goal of this optimization problem is to determine an upper bound for the WCRE of the analyzed system. As a final step, SysWCEC passes the ILP formulation to a mathematical solver, which is capable of finding worst-case values for all variables, i.e., node and edge frequencies. There are several mathematical solvers capable of solving the generated ILPs. SysWCEC currently supports the open-source solver lp_solve[1] and the commercial solver Gurobi [8].

WATWA-OS is based on the SysWCEC approach. Therefore, it reuses all mechanisms presented in this section in order to compute the WCRE of a real-time system. The next chapter focuses on how this approach has been extended to be able to minimize the worst-case energy consumption.

[1] https://lpsolve.sourceforge.net/

3 DESIGN

WATWA-OS' goal is to generate OS instances that are able to automatically switch to the most energy-efficient clock configuration depending on the current workload, thereby minimizing the energy consumption of the system. In the following, three challenges that arise from this goal are introduced, and WATWA-OS' solution to these challenges is presented. The specific challenges concern the computation of more accurate energy consumption bounds (Section 3.1), the generation of optimized Operating System (OS) instances (Section 3.2), and the time required to analyze and optimize a system (Section 3.3).

3.1 Challenge #1: Accurate WCRE Bounds

WATWA-OS is based on the approach of SysWCEC [25]. SysWCEC transforms the Control-Flow Graph of an application into a Power-State-Transition Graph (PSTG) containing all possible system states, including the power states of all connected devices. It then uses this PSTG to formulate a mathematical optimization problem, i.e., an ILP, whose solution is the WCRE of the system. To find the worst-case optimal energy consumption, WATWA-OS calculates multiple WCREs corresponding to different sequences of clock configurations, and chooses the lowest one.

As will be shown in the evaluation, the quality of WATWA-OS' predictions strongly benefits from accurate WCRE bounds (cf., Section 5.4). A Challenge that arises from this realization is the calculation of tight WCRE bounds. Although the construction rules of SysWCEC already provide good solutions in the majority of cases, there are still some situations that have arisen during the development of WATWA-OS where additional constraints or clarifications were necessary. These cases are addressed in the following.

3.1.1 Spurious Loop Activations

A problem that came up during the development of WATWA-OS concerns SysWCEC's handling of loops in the PSTG. Currently, SysWCEC adds a special constraint in the corresponding ILP for every loop inside the graph. This constraint prevents a loop from being executed if none of its entry edges are taken. A loop entry edge is defined as an edge that originates from a node outside the loop and points to an edge inside the loop. This definition, however, is not sufficient to prevent unwanted loop activations.

An example of this is shown in Figure 3.1, which contains the PSTG of an application with one interrupt (node `N6`). SysWCEC's loop-detection algorithm detects two loops in this graph: One loop containing nodes `N2`, `N3` and `N4` and one loop containing nodes `N3` and `N6`. Since node `N6` is not part

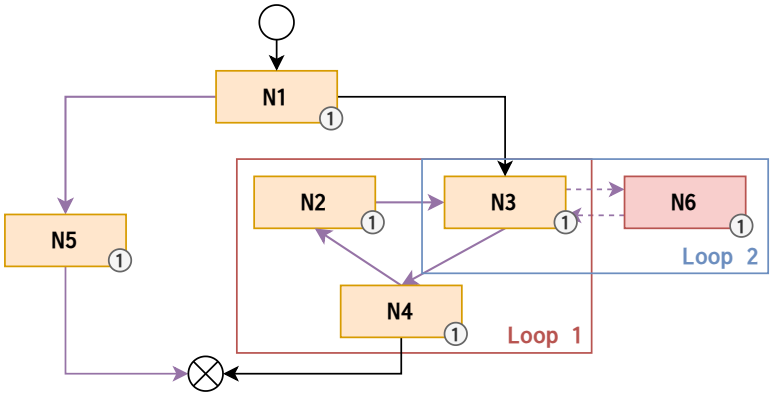


Figure 3.1 – PSTG that contains two loops that both share node N3

of loop 1, the edge N6→N3 counts as an entry edge into loop 1, alongside edge N1→N3. Similarly, the edge N3→N6 counts as entry edge for loop 2.

The depicted node and edge frequencies shown in circles represent a valid solution for the corresponding ILP of this graph. As can be seen, even with the additional loop constraints of SysWCEC, it is still possible that the left path through node N5 is taken while both loops are executed. This situation is not feasible in the real system since only the right or the left path of the graph can be taken, not both. The cause of this problem is that loops 1 and 2 can effectively activate each other without there being an actual path to them. This is because edge N6→N3 counts as a valid entry edge for loop 1, even if loop 2 is only reachable from loop 1.

As a workaround, WATWA-OS adapts the definition of an entry edge into a loop: An edge NX→NY is considered an entry edge of loop L, if NX is a node outside L and NY is a node inside L. If this edge, however, is only reachable from within the loop, i.e., all possible paths to NX go through a node of loop L, then it is not counted as an entry edge. This definition of an entry edge considers N1→N3 to be a valid entry for loop 1, but not N6→N3 since the only way to reach node N6 is through node N3. This way, only the left or right path can be taken through the graph in Figure 3.1.

3.1.2 State Changes during Interrupts

Another issue that can lead to an under-approximation of the WCRE concerns interrupts that change the power state of a device. Figure 3.2 shows an example PSTG of a system consisting of one task and one interrupt. Node N1 can be interrupted by node N2, which then calls `uart_off()` to turn off the UART controller. Since the interrupt causes a change in the power state of the system, it cannot

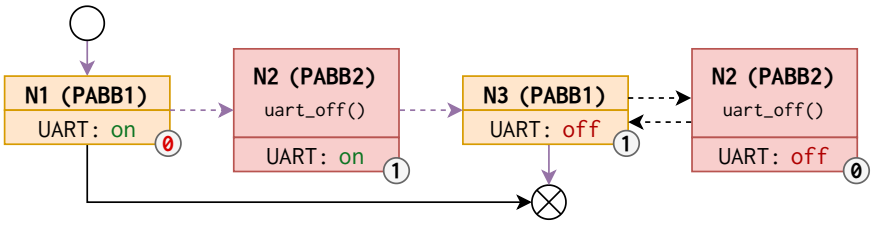


Figure 3.2 – Node N1 is interrupted by the interrupt in N2 that returns to another node N3 after changing the system state.

return to the previous node N1, but must return to a new node N3 that executes the same PABB as node N1 but has a different power state.

The node frequency calculation of SysWCEC can lead to an under-approximation in this particular case, where an interrupt does not return to the same node that was suspended. This is because SysWCEC subtracts the number of times a node has been interrupted from the node frequency. In the example, the frequency of node N1 is 0 since the number of interrupts (N1→N2) is subtracted from the input edge frequencies (entry→N1). The frequency of node N3, on the other hand, equals 1 because it is active after the interruption in node N2 (N2→N3 = 1). The idea behind SysWCEC's interrupt handling is that PABB1 is only executed once, regardless of how often it has been interrupted. However, the current algorithm incorrectly chooses N3 as the node that executes PABB1 instead of N1. This causes an under-estimation of the WCRE because the power consumption of node N3 is lower than that of N1.

To solve this problem, WATWA-OS uses a mechanism that was originally proposed by SysWCET [4], an automated WCRT analysis approach on which SysWCEC is based. The main difference is that SysWCET does not subtract all interrupt activations from a node but only considers completed interrupt-resume cycles. This is done by introducing new variables and constraints in the ILP that track the number of times a node has been interrupted and not resumed. If a node has been interrupted more times than it has been resumed, then the number of additional interruptions is added back to the node frequency. For the example in Figure 3.2, this means that the frequency of node N1 is adjusted to 1 since it is interrupted by node N2 but not resumed by it.

This approach solves the problem of under-approximation but still does not provide the optimal solution for node frequencies, as both nodes N1 and N1 are now counted. In the future, the ILP construction could be further refined by detecting these edge cases and selecting only the node with the highest power consumption (i.e., node N1 instead of node N3 in Figure 3.2).

3.2 Challenge #2: Generating Optimized OS Instances

WATWA-OS is a framework for creating, analyzing, and optimizing OS instances for embedded real-time systems. This section introduces the Operating System generated by WATWA-OS, including its semantics and main features. It then discusses how the SysWCEC analysis approach has been extended by different Clock-Configuration Decision Points to minimize the WCRE of an OS instance.

3.2.1 Operating System

WATWA-OS is capable of generating instances of a small operating system with a simple fixed-priority preemptive scheduler. The OS supports tasks with different priorities, but only one priority per task. This keeps the scheduling behavior deterministic, which greatly simplifies subsequent analysis steps. Tasks can affect OS scheduling with two central system calls:

- `activate(t)` marks a new task `t` as runnable. If the priority of `t` is greater than the priority of the current task, then it will be executed immediately. Conversely, if the priority of `t` is less than that of the current task, then `t` will be executed after all tasks with higher priorities have been completed.
- `terminate()` stops the execution of the current task. The OS scheduler will then look for the next runnable task with the highest priority and start executing it.

WATWA-OS also supports non-nestable Interrupt Service Routines (ISRs), which are similarly able to activate tasks using a special syscall `activateFromIsr(t)`. High-priority tasks activated in an ISR

will not be executed immediately, but only after the ISR has been completed. Device interaction is also handled by system calls in WATWA-OS. For instance, interacting with the UART subsystem can be achieved with various `uart_`-syscalls (e.g., `uart_{on, off, init, write}`). Interaction with the clock subsystem is also possible. Later on, during the construction of the PSTG, WATWA-OS will use these syscalls as delimiters for PABBs.

To analyze the WCRE of an OS instance, some information about the involved tasks and ISRs must be obtained statically before system runtime. This includes the priority of all tasks, the start task, and the minimum Inter-Arrival Time of all ISRs. Industry standards for embedded operating systems, such as OSEK-OS [17], define special configuration files for this purpose in which all relevant OS objects are statically declared. In WATWA-OS, the application developer instead annotates tasks and ISRs directly in the source code files using compiler pragmas at their entry functions. Since the PSTG generation algorithm is implemented as an extension to the compiler, it can make use of these directives directly and annotate the corresponding nodes in the graph accordingly. For ISR functions, the developer can optionally define periodic alarms that are automatically set up by the OS boot code with a configurable interval.

3.2.2 Minimizing Energy Consumption

The presented Operating System is deterministic and, therefore, analyzable by WATWA-OS. To determine the WCRE of a single OS instance, WATWA-OS proceeds exactly as SysWCEC does: It transforms the CFG into a PSTG and then constructs a mathematical optimization problem to obtain an upper bound on the worst-case energy consumption. To find an optimal sequence of clock configurations that minimizes energy consumption, WATWA-OS modifies the PSTG and the resulting mathematical problem in certain ways. These modifications are presented below.

Clock-Configuration Decision Points

The main idea behind WATWA-OS is to add Clock-Configuration Decision Points (CCDPs) to the state-transition graph of a real-time system. At these CCDPs, the system can either keep its current clock configuration or switch to another. In the PSTG, a CCDP is represented by artificial nodes and edges that take care of the clock-configuration change. However, these nodes and edges do not yet correspond to an actual clock-configuration change in the application's source code. Instead, CCDPs can be considered more of a placeholder in the source code that can be replaced with an actual clock configuration syscall if the solver determines that doing so will save energy.

The question now arises as to where CCDPs should be inserted in the graph. As of now, WATWA-OS considers three situations at different levels of granularity:

- **Before and after device syscalls:** In this context, a device syscall refers to an operation that interacts with system peripherals, such as sending or receiving messages via UART. As discussed earlier, these I/O operations are best performed at a low clock frequency to save energy, but only if the duration of the operation is long enough to amortize the cost of switching to a different clock configuration. By analyzing different frequency choices, WATWA-OS can determine which permutation of clock configurations ultimately yields the best WCRE.

- **Before and after loops containing a device syscall:** A single device syscall may not be long enough to justify switching to a lower clock frequency. A good example of this is the `uart_write()`-syscall in WATWA-OS. It transmits only one character over the UART bus, which can be a very short operation depending on the baud rate. In most cases, however, this syscall is not just called once, but it is used in a loop to send whole strings of characters. In these

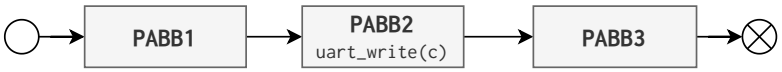
situations, it may be beneficial to switch to a lower clock frequency before the loop and then possibly switch back to a higher clock frequency after the loop.

- **At the start and end of a task:** For the same reason as in the previous point, it may be beneficial to run entire tasks with a particular clock configuration, for example when the application is divided into compute- and I/O-tasks. Determining clock frequencies at the task level is also done by the ScaleClock approach [21] introduced in Section 2.2.2. However, unlike ScaleClock, WATWA-OS is able to determine the optimal clock configuration for a task statically prior to system runtime, while ScaleClock runs its determination algorithm dynamically at runtime.

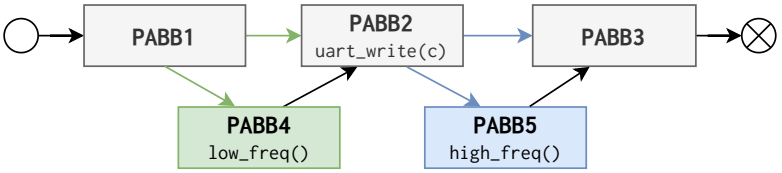
It is important to consider that each Clock-Configuration Decision Point doubles the number of possible configuration sequences. Therefore, as will be demonstrated in the evaluation, it is not always beneficial to add CCDPs whenever possible, as this can significantly increase the analysis time (cf., Section 5.5.3).

Example

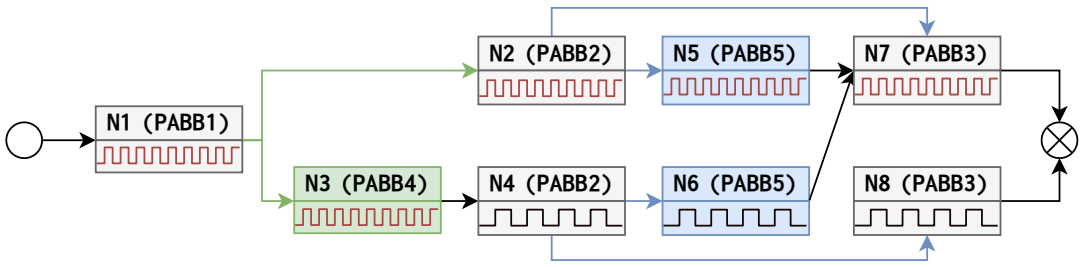
Figure 3.3a shows the PABB graph for a task consisting of three PABBs. Because PABB2 issues a device syscall (`uart_write()`), WATWA-OS will insert CCDPs before and after it, as seen in Figure 3.3b. Between PABB1 and PABB2, a new PABB4 is inserted that switches to a clock configuration with a lower, more energy-efficient frequency. In addition, two new edges are inserted, connecting PABB1, PABB4, and PABB2, while the original edge between PABB1 and PABB2 remains in place. As a result,



(a) Base PABB graph



(b) PABB graph with CCDPs



(c) PSTG with CCDPs

Figure 3.3 – PABB graph and PSTG for a sample task that consists of 3 PABBs

there are now two alternative ways to reach PABB2: Either directly through PABB1 or through PABB4 (cf., green edges in the figure). This information is stored in the graph for later analysis, as only one of these alternative edges can be taken in the final ILP. In the same way, a new PABB5 is created that switches back to a high clock frequency after the device syscall (cf., blue edges in the figure). In total, the two CCDPs result in four possible paths through the graph. Each additional decision point doubles this number again, increasing the number of possible paths exponentially.

Figure 3.3c shows the resulting PSTG constructed by enumerating all possible system states. As can be seen, many PABBs now exist as two nodes: Once with a high-power state and once with a low-power state. Furthermore, the alternative edges between PABB2, PABB3, and PABB5 also exist twice (cf., blue edges). This, however, does not mean that there are now more alternative paths through the graph. In this instance, it is still only possible to go either from PABB2 to PABB5 or from PABB2 to PABB3, resulting in four possible paths through the graph.

Constructing ILPs

The example in the previous section introduced multiple alternative paths through the PABB graph and PSTG of a real-time system. In the next step, WATWA-OS has to determine the best permutation of all alternative edges that minimizes energy consumption. This requires modifying the ILP construction mechanism to find the minimum WCRE among all alternative paths.

In its current state, transforming the PSTG into an ILP by only using the rules of SysWCEC (cf., Section 2.3) does not yield the desired results. Instead, the ILP solver returns the *worst* WCRE among all possible paths through the graph, exactly the opposite of what is being sought. To solve this problem, WATWA-OS essentially creates one ILP for each alternative path through the graph, which can then be solved in the same way as before to obtain the WCRE. Among all the solutions, WATWA-OS then chooses the one with the minimal WCRE. To find all possible alternative paths through the graph, it uses the information about alternative edges previously stored in the PSTG. These alternatives are then expressed as additional constraints in the ILP, each disabling certain edges that are not part of the current alternative. Thus, the constructed ILPs for all alternatives are mostly identical, except for a few constraints that deactivate certain edges. The deactivation of an edge in the PSTG is achieved by setting its corresponding frequency variable in the ILP to zero.

Regarding the example in Figure 3.3c, ILPs corresponding to the following four paths are constructed and solved. Edges to unmentioned PABBs are disabled in the ILP.

- ILP 1: PABB1 → PABB2 → PABB3
- ILP 2: PABB1 → PABB4 → PABB2 → PABB3
- ILP 3: PABB1 → PABB4 → PABB2 → PABB5 → PABB3
- ILP 4: PABB1 → PABB2 → PABB5 → PABB3

It is important to consider that, conceptually, edges are disabled at the *PABB* layer. However, since the ILP works with nodes and edges on the *PSTG* layer, the PSTG edges corresponding to the PABB edges must be identified and disabled.

Integrating Clock-Configuration Changes into the OS

Up to now, the Clock-Configuration Decision Points added to the PSTG by WATWA-OS do not yet actually correspond to any executed instructions. Instead, empty basic blocks have been inserted in the control-flow graph of the application by the compiler that serve as placeholders. After finding the

solution with the minimal WCRE among all solved ILPs, WATWA-OS is able to reconstruct the basic blocks of the original CFG that need to be replaced by actual clock configuration syscalls. The final binary created by WATWA-OS is an OS instance that automatically changes the clock configuration of the system at the determined points.

3.3 Challenge #3: Analysis and Optimization Time

The previous section introduced the main concepts of WATWA-OS' optimization mechanism. For each possible clock configuration sequence, a separate ILP is solved. Among all those ILPs, the one with the lowest WCRE bound represents the worst-case optimal clock configuration sequence.

Solving multiple ILPs can incur a significant time overhead. Therefore, a key challenge in the design of WATWA-OS was to reduce the solving time. This section introduces two approaches employed by WATWA-OS to keep the solving time to a minimum.

3.3.1 Multi-Scenario ILPs

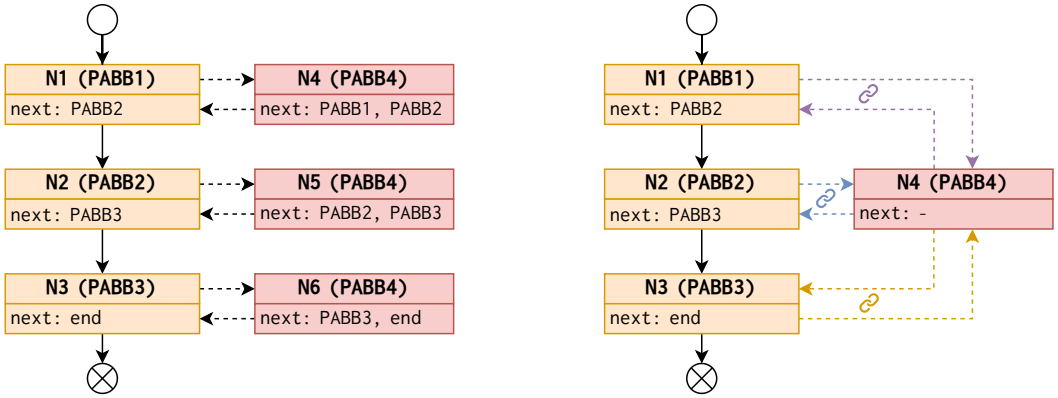
WATWA-OS generates and solves several similar ILPs that differ only in a few constraints. This can take a long time, especially when the number of nodes and edges in the PSTG is large. To speed up the solving process, WATWA-OS uses a special feature of the industrial ILP solver Gurobi [8] called *multi-scenario ILPs*.

A multi-scenario ILP consists of a base model and one or more scenarios, which introduce minor modifications to the set of constraints defined in the base model. This feature aims to speed up the solving time by allowing Gurobi to reuse information from previously solved scenarios. Moreover, the overhead of specifying all node and edge frequency variables for each ILP is completely eliminated. Multi-scenario ILPs are a perfect match for the ILPs generated by WATWA-OS. As a base model, the ILP that represents the entire PSTG, including all alternative edges, can be used. Then, for each possible alternative path through the graph, a new scenario can be created in Gurobi where all edges that are not part of the current alternative are disabled. This can be accomplished by modifying the pre-existing edge constraints, specifically by setting the edge frequency to 0.

3.3.2 Merging PSTG Nodes

A node in the PSTG provides a detailed view of the current system state, including the power state of all devices and the scheduling state of the OS. The scheduling state includes, among other things, information about all tasks and their next PABB to be executed. This detailed information is necessary when constructing the graph so that all possible system states are considered. However, the resulting PSTG usually ends up with many similar nodes and edges, each with its own frequency variables and constraints in the resulting ILP. Since the solving time of an ILP increases as more variables are added, reducing the number of nodes and edges in the PSTG was a concern in the design of WATWA-OS.

Figure 3.4 shows how WATWA-OS can reduce the number of nodes in the PSTG by merging similar states together in certain situations. The graphs were generated from a task consisting of three interruptible PABBs and one ISR. Figure 3.4a depicts the normal PSTG that is generated by SysWCEC. There is one interrupt node for each interruptible node in the graph (`N4`, `N5`, and `N6`). This is necessary because an interrupt node stores its return target in its scheduling state. However, since all of these three interrupt nodes have the same power state and execute the same PABB, it is possible to merge them into a single node, thus reducing the number of variables in the resulting



(a) Normal PSTG **(b)** Merged PSTG

Figure 3.4 – Two PSTGs of a task consisting of three interruptible nodes and one ISR

ILP. This is exactly what WATWA-OS does, as seen in Figure 3.4b. In the presented example, the number of duplicated nodes is relatively small, resulting only in a slight reduction in the number of variables. In more realistic scenarios, though, the benefits become more apparent, especially when ISRs activate higher-priority tasks that themselves consist of multiple PABBs.

WATWA-OS is able to detect interrupt-induced sub-graphs in the PSTG and merge them together. This merging process must be performed as last step in the PSTG construction, as it invalidates the scheduling state of the involved nodes.

A new problem that arises when merging multiple nodes can be seen in Figure 3.4b. Without further intervention, the ILP solver would now be able to enter the interrupt node N4 through node N1 and then exit to node N3. This is not valid, as each interruption must resume the same PABB that it suspended. To overcome these inaccuracies, WATWA-OS introduces new ILP constraints that prevent the solver from taking invalid paths. Essentially, the edge frequency from one node to an interrupt must be equal to the sum of all edge frequencies that go back to the original node. In the example, this results in three new constraints:

$$
\begin{aligned}
f(\mathsf{N1} \rightarrow \mathsf{N4}) &= f(\mathsf{N4} \rightarrow \mathsf{N1}) \\
f(\mathsf{N2} \rightarrow \mathsf{N4}) &= f(\mathsf{N4} \rightarrow \mathsf{N2}) \\
f(\mathsf{N3} \rightarrow \mathsf{N4}) &= f(\mathsf{N4} \rightarrow \mathsf{N3})
\end{aligned}
$$

3.4 Summary

WATWA-OS' goal is to generate optimized OS instances that automatically switch to the most energy-efficient clock configuration depending on the performed workload. This chapter dealt with how the existing SysWCEC approach by Wägemann et al. has been modified and extended to achieve this goal, and the challenges along with their solutions were presented.

The main idea behind WATWA-OS' approach is to create multiple ILPs, each representing a different sequence of clock configurations in the system (cf., Section 3.2). All ILPs are then solved by a mathematical solver, and the solution with the lowest WCRE bound is selected. WATWA-OS introduces additional constraints to SysWCEC's mechanism to obtain more accurate and tighter WCRE bounds (cf., Section 3.1), which ultimately result in higher-quality clock configuration predictions. Finally, WATWA-OS makes an effort to reduce the solving time of all ILPs by reducing the number of nodes in the PSTG and utilizing solver-specific features (cf., Section 3.3).

4 IMPLEMENTATION

This chapter gives an insight into the implementation of WATWA-OS. It examines the tools used and how they were extended to generate optimized OS instances. Section 4.1 provides an overview of all tools needed for WATWA-OS' approach. Details of the Operating System are then presented in Section 4.2. The construction of the PSTG, WATWA-OS' main data structure, is presented in Section 4.3. Finally, the transformation of the PSTG into an ILP is shown in Section 4.4, and the optimization mechanism is discussed in Section 4.5.

4.1 Framework Overview

WATWA-OS uses several tools to get from the application and OS source code to the final optimized executable binary. The starting point is an existing adaptation of SysWCEC [25] called WoCA [19], which already implements many of the required mechanisms, such as a part of the PSTG construction algorithm. For WATWA-OS, the WoCA source code was modified and extended to include support for multiple clock configurations and node merging.

Figure 4.1 is an overview of the tools that are involved in the creation of an OS instance. There are two primary components: The WATWA-OS Analyzer and the WATWA-OS Optimizer.

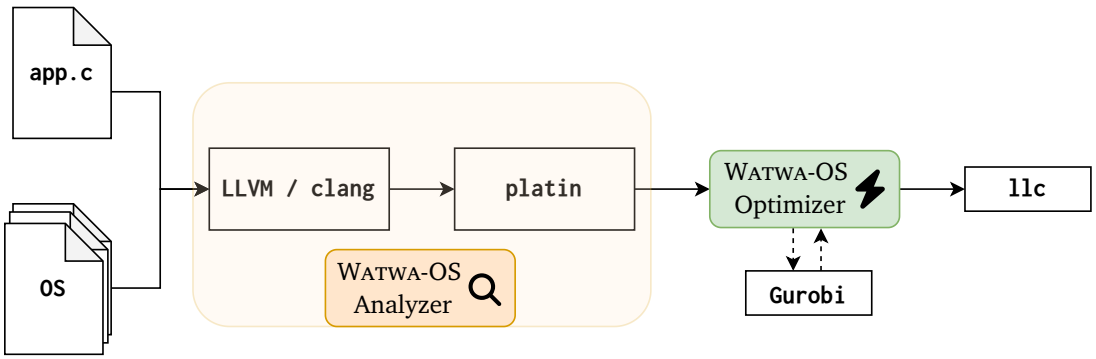


Figure 4.1 – Overview of all involved tools in WATWA-OS (simplified)

WATWA-OS Analyzer

The WATWA-OS Analyzer creates the data structures necessary for analysis (i.e., the PSTG) and formulates all variables and constraints needed for the mathematical optimization problem. The implementation of the analysis process is split up into the following two software components:

- **LLVM / clang**: LLVM [15] is the compiler framework used to build the executable binaries. A new module pass has been added that integrates the PSTG construction mechanism. This integration directly into the compiler has the advantage that internal information and data structures, such as the CFG, can be reused to facilitate the construction. For the same reason, the OS code generation is also handled by the LLVM compiler. Since WATWA-OS uses pragmas to define tasks and ISRs, handling the OS code generation directly in the compiler where these pragmas are parsed makes sense. In addition to a binary file, LLVM now also outputs a .pml file containing all the necessary information for subsequent WCET and WCEC analysis/optimization. It should also be noted that the compiler is actually invoked twice. The first time, the OS is only compiled into the LLVM intermediate representation (LLVM-IR). This IR file will be used later to inject calls to methods for changing the clock configuration. The second time, the LLVM-IR code is compiled into an executable file using `llc`.
- **`platin`**: `platin` [9] is an analysis framework that can use the .pml files generated by LLVM to construct an ILP, which can then be used to obtain the WCRE of the application. It has been extended to output a JSON file with all the information about the ILP and the possible clock frequency alternatives.

WATWA-OS Optimizer

The WATWA-OS Optimizer is an application that uses the JSON file generated by `platin` to create a multi-scenario ILP and invoke the Gurobi ILP solver [8]. It determines the best alternative among multiple clock configuration decisions and then injects the IR file created by LLVM with calls to clock-configuration changes at the appropriate points. This optimized IR file is then compiled by LLVM and linked with the OS to obtain a final binary that can be flashed on the hardware platform.

4.2 Operating System

WATWA-OS generates small OS instances that support interrupts and use a fixed-priority preemptive scheduler (cf., Section 3.2.1). The OS is divided into a hardware-independent and a hardware-dependent part. The hardware-independent part consists of all scheduling and initialization routines, e.g., methods for initializing tasks and ISRs at boot time. The hardware-dependent part, on the other hand, consists of all hardware- and architecture-specific operations, such as the UART driver code and setup code for periodic timer ISRs.

In WATWA-OS, the application developer declares and specifies tasks and ISRs directly in the source code using pragmas. Listing 4.1 shows how such a pragma can be used to declare the OS start task with a priority of 10. The clang compiler frontend has been extended to be able to parse these pragmas and map them to LLVM-IR function attribute-value pairs (cf., Listing 4.2). In the LLVM backend, a new *OSGenerator* pass has been added that, in turn, reads these function attributes and uses them to generate a new source code file containing all the necessary OS objects (i.e., tasks, interrupts, and their corresponding data structures). This file is compiled and linked with all other OS binary files in a final step to create a complete OS instance. Using key-value pairs to specify task

```
#pragma os task "id=1,priority=10,start=true"
void task0() {
  ...
}
```

Listing 4.1 – Application Source Code

```
define dso_local void @task0() #0 {
  ...
}
attributes #0 = {
  "os-task"="id=1,priority=10,start=true"
}
```

Listing 4.2 – Corresponding LLVM IR output

or ISR parameters provides a flexible approach that can easily be extended to incorporate more configurable parameters. However, if at some point more complex configuration options are desired, such as shared objects between multiple tasks, dedicated and structured configuration files should be used as proposed by industry OS standards such as OSEK-OS [17].

4.3 PSTG Generation

The main data structure of WATWA-OS, the PSTG, is constructed in an LLVM pass. This section details how to get from the basic blocks of the original application Control-Flow Graph to a PABB graph and finally to a full PSTG that can be analyzed in subsequent steps.

4.3.1 Preparing Basic Blocks

Splitting

At first, the WATWA-OS Analyzer starts with the CFG of the application, which consists of several LLVM bitcode basic blocks. However, before these basic blocks can be merged into PABBs, certain blocks must be split, and even new empty basic blocks may be created. This is because at certain points in the code, the ILP solver should later be able to decide whether it is beneficial to switch to a different clock configuration. Therefore, wherever possible, a new empty basic block must be created, which may later be filled with a call to a clock configuration function. These points include before and after device syscalls or loops that contain device syscalls, as well as at the start and end of a task. The newly created basic blocks are specially labeled according to the specific clock configuration that they refer to. For example, a basic block *before* a device syscall may reflect a possible switch to a low-frequency clock configuration, while a basic block *after* such a syscall may reflect a possible switch to a high-frequency clock configuration.

Special care must be taken when adding new basic blocks inside loops, as this may invalidate previously computed loop information that is needed later by the PSTG construction mechanism and other LLVM passes. The WATWA-OS Analyzer automatically adjusts all loop information when splitting basic blocks belonging to one or more loops. In addition, basic blocks are split at each OS syscall. This ensures that calls to scheduling syscalls are located in their own basic block.

Merging

As a second step, LLVM basic blocks are merged into PABBs that form a single-entry single-exit region. This is done to reduce the number of nodes in the final PSTG. The WATWA-OS Analyzer, therefore, attempts to merge multiple basic blocks between syscall barriers. Syscalls and the specially labeled clock configuration blocks of the previous step form a barrier that the merging algorithm cannot

cross. The merging mechanism can even consolidate entire control flow structures, such as loops, thus significantly reducing the number of nodes in the final graph.

4.3.2 Frequency Alternatives

The current PABB graph consists of nodes with real basic blocks and artificial nodes representing a possible clock-configuration change. One current issue is that the artificial nodes are always executed, i.e., there is no way to skip the execution of a clock-configuration change. Therefore, when constructing the PSTG, additional edges are inserted, connecting the predecessor of an artificial node to the successor of the artificial node. This makes it possible to skip the execution of a clock-configuration change node.

An example of this is shown in Figure 4.2. PABB2 is an artificial node that has been inserted to change to a lower-frequency clock configuration before the `uart_write()` device syscall is called. The WATWA-OS Analyzer adds a new edge in this graph connecting PABB1 and PABB3, thus providing the ability to skip the execution of PABB2. The two edges from PABB1 to PABB2 and from PABB1 to PABB3 form a clock configuration alternative. This means that two ILP scenarios will be created later on: One where the edge PABB1→PABB2 is active and one where the edge PABB1→PABB3 is active. The other edge is deactivated, respectively. The information about these two alternative edges is stored as metadata for PABB1 and will be used later to create the multi-scenario ILP.

4.3.3 State Enumeration

In the next step, all possible states of the PABB graph are enumerated to obtain the final PSTG. At the beginning, each node in the graph is given an initial state and outgoing state. The outgoing state reflects the state of the system that applies after the node is executed. For example, a node that calls the `low_freq()` function may have a high-frequency state (because it executes at a high frequency) but a low-frequency outgoing state (because it switches to a low frequency). The state enumeration algorithm then visits every node, starting from the entry node. For each node, its state is set to the outgoing state of its predecessor node(s). Note that this is only possible if all predecessor nodes have the same outgoing state. Otherwise, a conflict will occur that must be resolved, which will be discussed later. Then, the outgoing state of the current node is updated. For this, the state is copied to the outgoing state. Afterwards, if the node calls a syscall, the semantics of that syscall are applied to the outgoing state.

An example of a partial state enumeration for three PABBs is depicted in Figure 4.3a. The state enumeration algorithm visited the nodes in the following order: PABB1, PABB2, PABB3. However, at PABB3, the algorithm encountered a problem: The outgoing state of its predecessor PABB1 is different from the outgoing state of its predecessor PABB2. Thus, it is not clear which state should be assigned to PABB3. When the state enumeration algorithm encounters such conflicts, it remembers the current node and all incoming states and then chooses the state of **one** previously visited predecessor to be able to continue the enumeration. In this case, the analyzer memorizes that a conflict in PABB3 occurred with two different incoming states, and the outgoing state of PABB1 is used to continue.

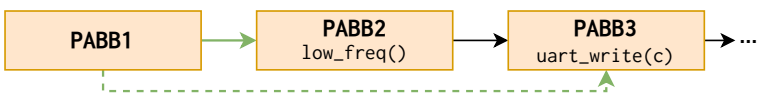
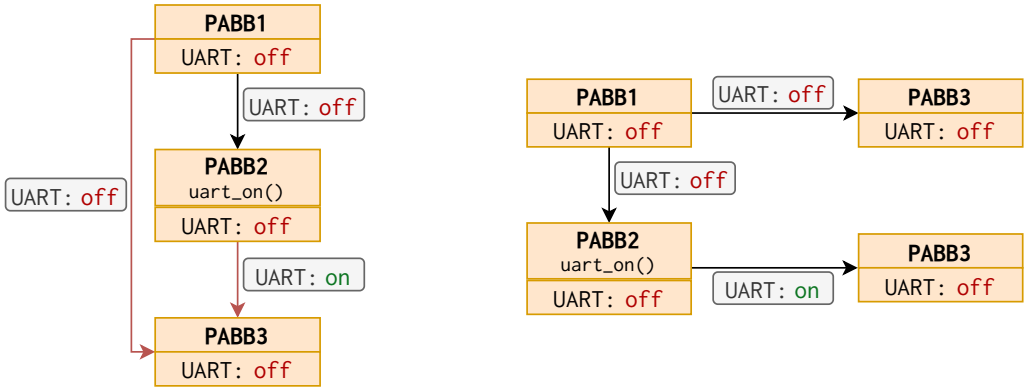


Figure 4.2 – Alternative Edges in the PABB graph



(a) Conflict at PABB3 **(b)** After conflict resolution

Figure 4.3 – Example for a state enumeration with 3 PABBs.

After all nodes have been visited, the state enumeration algorithm resolves any remaining conflicts. This is done by creating a copy of each conflict node for each different input state. In the example, PABB3 is duplicated so that there is a version with the outgoing state of PABB1 and a version with the outgoing state of PABB2, as seen in Figure 4.3b. The edges in the graph are adjusted accordingly. If conflicts have occurred during an iteration of the algorithm, then the enumeration is repeated on the new version of the graph. This is necessary because new conflicts may occur in the next iteration. When no more conflicts exist, the state enumeration is complete and the PSTG has reached its final form.

The presented state enumeration technique is used for all device syscalls. The WATWA-OS Analyzer uses a similar algorithm when enumerating all possible scheduling-related paths through the system. The only difference is that these syscalls themselves can now affect the edges in the PSTG. This is the case, for example, when a PABB activates a higher-priority task. In this case, the existing edge to the successor of the PABB is removed, and a new edge to the high-priority task is inserted. The previous successor is then saved as return target in the scheduling state of the new successor.

It is important to note that the WATWA-OS Analyzer starts the PSTG construction by enumerating all *scheduling-related* paths. This creates a graph of all valid execution paths through the system. Then, in a second step, all *device syscalls* are evaluated using the abovementioned mechanism.

After enumerating all possible system states, WATWA-OS tries to merge interrupt nodes (cf., Section 3.3.2). This merging is done by finding identical interrupt subgraphs in the PSTG and keeping only one of them. All edges to the deleted subgraphs are then adjusted to point to the retained interrupt subgraph.

4.3.4 PML Output

After the PSTG is built, LLVM dumps it into a Program Metainfo Language (PML) file, which is recognized by the `platin` analysis toolkit. A PML file is a collection of one or multiple YAML documents that store information about the program structure and other meta-information (e.g., flow facts). In addition to the PSTG, more information is written to the file which can later be used by `platin` to further tighten the WCRE bounds. This includes the bitcode and machine basic blocks for each function and their instructions. The bitcode basic blocks are used in a node of the PSTG. The corresponding machine basic blocks and their instructions are used to calculate the WCET bounds

in platin. However, a 1:1 mapping between bitcode and machine code basic blocks does not necessarily exist. The compiler may, for instance, create additional machine code blocks or rearrange code for optimization purposes. Nevertheless, finding the corresponding machine code instructions for a bitcode basic block is essential for calculating its WCET. To relate the basic blocks of these two layers, platin uses so-called Control-Flow Relation Graphs (CFRGs) [10]. CFRGs consist of different node types corresponding to machine or bitcode basic blocks or both. A CFRG node that contains blocks of both layers is called a *progress node*. As its name suggests, it synchronizes the progress of the bitcode layer with the machine code layer, thus allowing a bitcode basic block to be associated with one or more machine code basic blocks. The WATWA-OS Analyzer uses the previous work of the T-CREST project [10], which creates CFRGs in LLVM that are then written to the PML file.

Finally, the PML file also contains information about flow facts that help to bound the WCRE of the application. This includes the minimum Inter-Arrival Times of interrupts, which can be specified with a pragma, as explained in Section 4.2. In addition, the application developer can annotate loops with a special loopbound pragma to specify their minimum and maximum number of iterations. platin already includes mechanisms that transform these flow facts from the bitcode layer to the machine code layer to further tighten the WCRE bounds.

4.4 ILP Construction

platin is a toolkit for analyzing the worst-case behavior of a system written in Ruby. As a basis, the WATWA-OS Analyzer uses a modified version of platin from the SysWCEC project, which already provides the ability to create ILPs that maximize the WCRE.

platin takes several PML files as input, providing not only the necessary information about the program to be analyzed (i.e., the PSTG, basic blocks, and flow facts), but also information about the target platform. WATWA-OS adds the following information in order to compute the WCRE:

- **Clock Configurations:** WATWA-OS currently supports two clock configurations. For each configuration, the clock frequency, as well as the maximum power consumption of the system at that configuration, must be specified in a PML file. In addition, the maximum transitioning costs (i.e., energy consumption and duration) from one clock configuration to another are required.
- **OS Syscalls:** The maximum duration (in number of clock cycles) of each OS syscall is needed. In theory, platin could calculate this information by analyzing the corresponding OS functions. However, in its current state, WATWA-OS' scheduling syscalls all depend on the number of tasks in the system. This means that the more tasks an OS instance has, the longer it will take to execute one of these syscalls. In order to calculate upper bounds depending on the number of tasks, additional flow facts would need to be introduced to platin. WATWA-OS currently uses a different approach and annotates syscall nodes directly in the PSTG with their maximum cost in LLVM. This is because, at that point, the number of tasks is known, and thus, the maximum number of cycles per syscall can be calculated.
- **Device Syscalls:** For device syscalls, the maximum energy consumption and duration (in seconds) per clock configuration must be specified. The maximum number of clock cycles alone, as for OS syscalls, is not sufficient because device syscalls may consume different amounts of time and energy depending on the current clock configuration. For instance, a call to uart_write() at a high clock frequency may be only slightly faster than at a low frequency due to slow transfer speeds and thus consume significantly more energy.

Having all the necessary information, `platin` is able to use the Implicit Path Enumeration Technique (IPET) [16, 18] to construct all variables and constraints needed for an ILP that determines an upper bound on the energy consumption of the system. As a first step, the maximum duration (unit: number of clock cycles) of each PABB is calculated by analyzing the executed bitcode and machine code basic blocks. To achieve this calculation, `platin` utilizes the CFRG and any applicable flow facts, e.g., loop bounds, which have been annotated in the source code. It then sums up the maximum costs of each executed instruction in the worst-case execution path. These costs per instruction have to be determined beforehand and are specified in `platin` for each supported hardware platform.

After determining the maximum duration for each PABB, `platin` then constructs all variables and constraints for the final ILP. One of these constructed variables is the global timing variable, which corresponds to the total execution time of the system. Previously, this timing variable represented a number of clock cycles. However, using the number of clock cycles to represent execution times is no longer possible in WATWA-OS. Instead, the actual execution time in seconds must be used. This is due to the fact that from now on, the clock configuration of the system can change several times during operation, which can lead to different execution times per PABB. WATWA-OS, therefore, determines the real execution time per combination of PABB and clock frequency.

The resulting ILP constructed by `platin` contains all variables and constraints for the entire PSTG, including the clock configuration decision nodes. Solving this model yields the worst WCRE among all possible clock configuration alternatives. To find the *best* WCRE, multiple scenarios are generated from the base ILP returned by `platin`, which are then separately solved. This is done in the last step by the WATWA-OS Optimizer.

4.5 OS Optimization

As mentioned in the previous section, `platin` constructs an ILP containing all the necessary variables and constraints. It then dumps all information into a JSON file, which is read and parsed in the last step by the WATWA-OS Optimizer.

The WATWA-OS Optimizer creates a multi-scenario ILP, where each scenario corresponds to a different sequence of activated clock configurations. For this, the Python interface of the commercial solver Gurobi is used, which provides the ability to create and solve multi-scenario ILPs. The model is created as follows:

1. First, all variables and constraints are added to a new ILP model. This corresponds to all nodes and edges of the PSTG, including all clock configuration nodes. The resulting model serves as the basis for all following scenarios.

2. Second, for each permutation of clock configuration alternatives, a new scenario is added to the existing ILP model. A single clock configuration alternative describes the possibility of choosing between two edges in the PSTG: either the clock configuration is changed or not. For a specific permutation of alternatives, only one of these two edges can be taken, not both. Consequently, one of the two edges must be deactivated. This is achieved by setting the upper bound of the corresponding edge variable in the scenario to 0.

The resulting multi-scenario model is solved by Gurobi. Afterwards, the WATWA-OS Optimizer searches the scenario with the lowest WCRE bound among all solutions. It extracts the variable frequencies determined by the solver and translates them into active and inactive nodes and edges in the PSTG. From this, the optimizer is able to determine at which points in the code a clock-configuration change is necessary.

Finally, the optimizer injects the LLVM IR code of the OS instance with actual calls to the methods for changing the clock configuration. To reiterate, the basic blocks responsible for changing the clock configuration were empty until now. That is, they did not contain any real logic. In the PSTG, however, these basic blocks were treated as if they actually changed the clock configuration. Therefore, in the last step, the logic has to be added to all basic blocks that are included in the best scenario. The injected LLVM IR file can then be compiled and linked to obtain the final executable image.

4.6 Summary

This chapter discussed the implementation of WATWA-OS' two main components, the Analyzer and the Optimizer. The Analyzer implementation has been integrated into the LLVM compiler framework and the `platin` analysis tool kit. The main benefit of this separation is that already existing software features can be reused. For example, LLVM already provides a CFG construction mechanism, which is needed to compile the source code. `platin`, on the other hand, is capable of applying the IPET required to formulate a mathematical optimization problem.

The WATWA-OS Optimizer takes care of specifying a multi-scenario ILP and then calls the mathematical solver to obtain a solution for each scenario. Finally, it adds the actual clock-configuration calls to the application at the determined points and outputs the optimized application as LLVM IR code.

5 EVALUATION

The goal of WATWA-OS is to minimize the Worst-Case Response Energy Consumption (WCRE) of an OS instance by switching between different clock configurations at runtime. This chapter evaluates the quality of WATWA-OS' predictions as well as the runtime of the optimization toolkit on a real hardware platform. Sections 5.1 to 5.3 describe the hardware platform and measurement setup and determine as yet unknown energy and timing values required for the WATWA-OS Analyzer. Then, in Section 5.4, the quality of WATWA-OS' predictions is evaluated using an example application scenario. Sections 5.5 and 5.6 deal with the offline properties of WATWA-OS, i.e., the analysis and optimization time and the effect of node merging and multi-scenario ILPs. Finally, the evaluation results are summarized in Section 5.7.

5.1 Hardware Platform

The implementation of WATWA-OS has been evaluated on an ESP32-C3 by Espressif Systems [6], a small, low-cost MCU based on the RISC-V architecture. The ESP32-C3 offers a single RISC-V core with a 4-stage in-order pipeline and 400 kB of single-cycle access SRAM. This makes it a suitable hardware platform for WATWA-OS since the effects of caches or complex pipelining are not currently taken into account. The ESP32-C3 is a popular and widely deployed MCU with a rich set of peripherals, most notably Wi-Fi and Bluetooth support.

OS instances of WATWA-OS currently support up to three periodic timer interrupts as well as the UART and GPIO peripherals of the ESP32-C3. These OS instances run completely bare-metal and do not depend on other software such as the ESP-IDF framework[2]. The boot code of WATWA-OS takes care of initializing the hardware platform. It performs the following actions on the ESP32-C3:

1. **Disabling Watchdogs:** The ESP32-C3 features multiple watchdog timers that must be periodically fed by the operating system to prevent the chip from being reset. The idea behind these timers is to be able to detect and handle erroneous software behavior, such as when the system is stuck in an endless loop. Because their functionality is not required for WATWA-OS, the boot code disables these timers. This results in better analyzability of the whole system, as interrupts can significantly increase the number of system states that need to be considered.

2. **Initialization of Timer Interrupts:** The OS initializes all periodic timer interrupts, which have been specified in the application source code. WATWA-OS, therefore, utilizes the SYSTIMER component of the ESP32-C3 [7], which offers three different timer comparators that can be set independently of each other.

[2] https://docs.espressif.com/projects/esp-idf/en/v5.3.1/esp32c3/get-started/index.html

3. **Initial Clock Configuration:** The boot code selects an initial clock configuration for the CPU. There are several ways to influence the clock frequency on the ESP32-C3. This is explored in more detail in Section 5.3.
4. **Peripheral Initialization:** The boot code initializes the UART controller with a given baud rate. This involves calculating necessary divider values for the clock signal that is used to generate the baud rate.
5. **Scheduler Initialization:** The final step is the initialization of the OS scheduler. WATWA-OS sets up an initial stack for every task and marks the startup task as runnable. This initial task will then be dispatched.

5.2 Test and Measurement Setup

This section presents the test and measurement setup used to evaluate WATWA-OS, and shows how accurate measurements were achieved.

5.2.1 Specifications

The following evaluation encompasses multiple hardware and software components, ranging from the host system on which the OS instances are compiled and optimized to the final hardware platform on which measurements are taken. All relevant components of the measurement setup and their specifications are listed in Table 5.1.

5.2.2 Time and Energy Measurements

In order to determine tight WCRE bounds, accurate measurements of the hardware platform's timing- and energy-related behavior are necessary. As explained in Section 4.4, WATWA-OS requires several predetermined energy and timing values to find optimal clock configurations, e.g., for device syscalls. For the evaluation of WATWA-OS, energy, and timing values are measured using the Joulescope JS220 Precision DC Energy Analyzer [13]. It measures both voltage and current for the device under test at 2 million samples per second and is thus able to provide accurate values even for relatively short tests.

The JS220 also supports four General Purpose Inputs (GPIs), one of which is utilized to determine the start and end of a measurement. Therefore, GPIO pin IO1 of the ESP32-C3 is connected to pin

Table 5.1 – Overview of all relevant test setup components

Compiler	OS Code: clang 14.0.6 (-Og) Application Code: modified version of clang 7 (-O0)
Host machine	CPU: Intel i7 4790 @ 3.60 GHz (4 cores / 8 threads) RAM: 16 GB (DDR3) OS: Debian 12 (Bookworm)
Target platform	Espressif ESP32-C3-WROOM-02 Single Core RISC-V, 400 kB SRAM
Measurement	Joulescope JS220 2 MHz sampling frequency

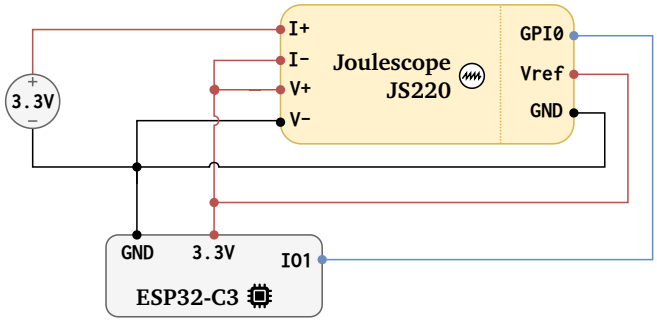


Figure 5.1 – Initial measurement setup

`IN0` on the JS220. In this setup, setting `IO1` to a logical 0 starts the measurement, while setting it to a logical 1 indicates the end of the measurement. The `Vref` and `GND` pins of the JS220 have been connected to the `3.3V` and `GND` pins of the ESP32-C3. Figure 5.1 shows the initial measurement setup with all connections. It is important to note that the Joulescope's sensor side is electrically isolated from the connected host computer, thus eliminating possible ground noise [13].

The setup presented in Figure 5.1 had been initially used. However, on closer inspection of the measurement results, it became clear that there were large fluctuations in the measured current values. These fluctuations can be seen in Figure 5.2, where simple multiplications were performed in a loop on the target device. The current scatters around 8 mA with an amplitude of up to 2 mA, which was not expected. After some experimentation, it became clear that the GPIO circuitry of the measurement setup caused these significant fluctuations. However, this part cannot simply be removed as it is necessary to accurately determine the start and end of a test.

To achieve influence-free measurements, an optocoupler has been added between the I/O output of the ESP32-C3 and the JS220. An optocoupler is a component that transfers electrical signals between two isolated circuits by using light. Inside the optocoupler is an LED that is turned on and off by the GPIO pin of the ESP32-C3. On the other end, there is a light detector (phototransistor) that works similar to a bipolar junction transistor. Since the LED which is toggled by the ESP32-C3 requires a considerable amount of current (several mA), it is important that it is always switched off during measurements to reduce the amount of noise in the system. However, even then, the power consumption of the LED does not drop to zero immediately. The measurements, therefore, still include some amount of energy not consumed by the MCU alone. To reduce these effects, an

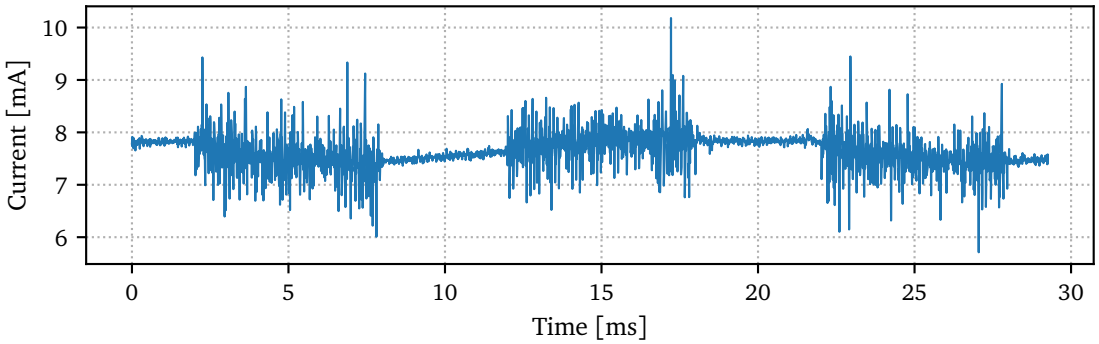


Figure 5.2 – Initial measurement setup: Current Fluctuations

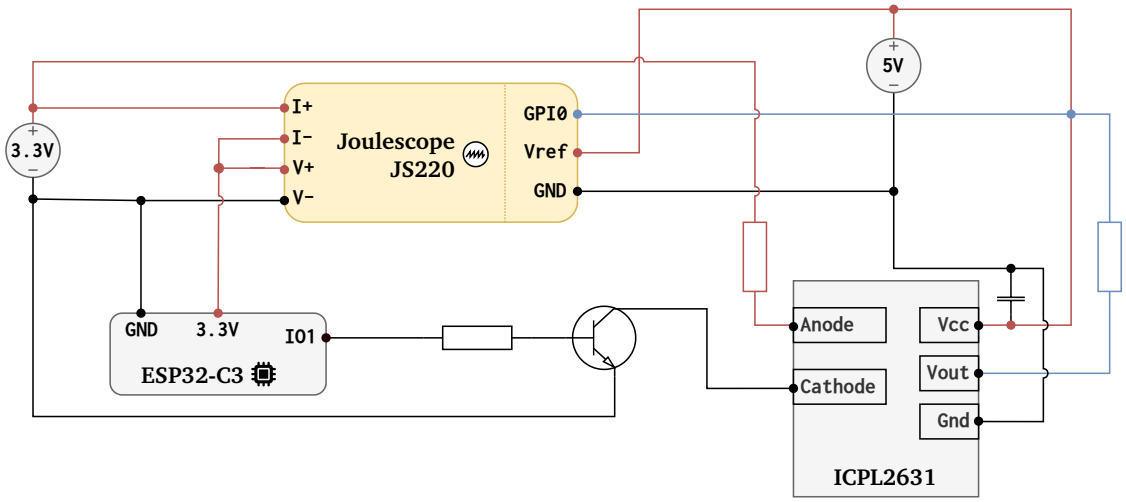


Figure 5.3 – Final measurement setup, including the optocoupler and transistor.

NPN-transistor that is controlled by the GPIO pin of the ESP32-C3 has been added. The collector and emitter pins of the transistor are directly connected to the measurement power supply. As a result, the energy analyzer no longer registers the collector-emitter current that drives the LED of the optocoupler. Instead, only a relatively small amount of current (i.e., µA) is measured, which the MCU needs to enable the transistor.

The final measurement setup is depicted in Figure 5.3. It uses an ICPL2631 optocoupler [11], optimized for high transfer rates between the two isolated circuits (10 Mbit/s). Note that this optocoupler operates at 5 V, which is provided by a second power supply. As the datasheet [11] recommends, a 0.1 µF capacitor has been added between the `Vcc` and `GND` pin of the optocoupler. This final setup provides stable current and voltage measurements with significantly reduced current fluctuations and no more external influence.

5.3 Finding Optimal Clock Configurations

Before WATWA-OS is able to generate optimized OS instances that can be evaluated on the MCU, some configuration parameters need to be determined. This section measures and examines these parameters, most importantly the clock configurations and their power demands.

WATWA-OS currently supports two clock configurations: one for compute-intensive tasks and one for I/O-intensive tasks. An essential component of a clock configuration is the CPU clock frequency. In general, low clock frequencies are more energy efficient for I/O-intensive tasks, while high clock frequencies are more energy efficient for compute-intensive tasks [2, 26]. However, simply selecting the lowest possible clock frequency for I/O operations may not be ideal, as the evaluation results in Section 5.3.3 will show. Moreover, the selection of a clock frequency is in itself a complex operation that may require the (re-)configuration of multiple system components.

The following subsections deal with finding optimal CPU clock frequencies on the ESP32-C3 depending on the operations being performed. Section 5.3.1 presents the clock tree of the target hardware platform and all of its frequency control knobs. Sections 5.3.2 and 5.3.3 evaluate the effect of different clock configurations on I/O- and compute-intensive tasks. Finally, all configuration parameters needed for subsequent analysis steps are presented in Section 5.3.4.

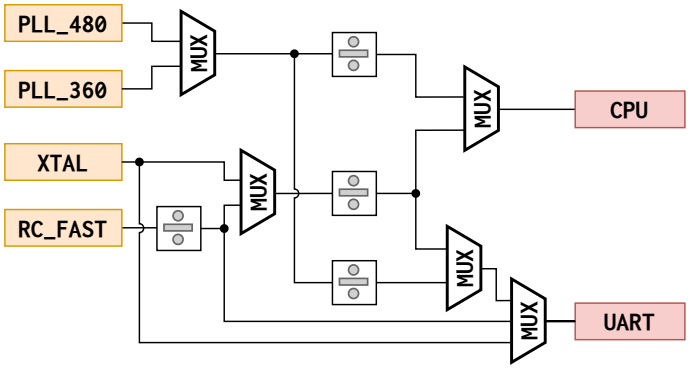


Figure 5.4 – Clock Tree of the ESP32-C3, only including the CPU and UART Controller

5.3.1 ESP32-C3 Clock Tree

Section 2.1 provided an overview of how different components, such as frequency scalers and multiplexers, are combined in modern MCUs to generate a clock frequency. The totality of all clock-related components forms a system's clock tree. The interdependencies between different components and configuration possibilities can become complex. Figure 5.4 shows the clock tree of the ESP32-C3 with respect to the components controlling the CPU frequency and the UART controller. Four clock sources can drive the CPU: Two PLLs (`PLL_480` and `PLL_360`), an external oscillator (`XTAL`), and an internal RC oscillator (`RC_FAST`). Each of these sources comes with one or multiple dividers, some of which can be configured in software. The `PLL_480` and `XTAL` oscillators can be configured to generate a wide range of clock frequencies. These two options are, therefore, used in the following to find optimal clock configurations. The UART controller is configured to always depend on the CPU clock frequency. This means that changing the clock frequency requires a reconfiguration of the UART controller, as the baud rate is derived from the UART controller frequency.

5.3.2 Compute-Intensive Operations

To find the optimal clock frequency for compute-intensive operations on the ESP32-C3, an application that performs 10,000 multiplications is evaluated. Several clock frequencies are tested, ranging from 1 MHz to 160 MHz. The results are shown in Figure 5.5. As expected, the higher the clock frequency,

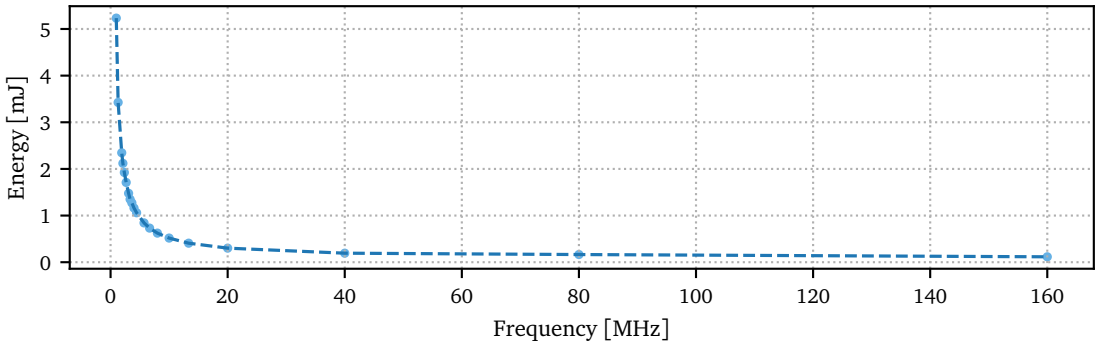


Figure 5.5 – 10,000 multiplications on different clock frequencies.

the less energy is consumed for the same number of multiplications. Thus, a clock frequency of 160 MHz is chosen for the clock configuration for compute-intensive workloads. While the power consumption of the MCU is very high at this frequency, the energy consumption per instruction is low because the time it takes to complete all multiplications is very short.

5.3.3 I/O-intensive Operations

Finding an energy-efficient clock frequency for I/O-intensive operations is not as straightforward as with compute-intensive workloads. This is because the optimal frequency depends on the duration of the executed I/O operation. An example of this can be seen in Figure 5.6. To find a suitable clock frequency for I/O operations on the ESP32-C3, 100 characters are transmitted over the UART protocol at different clock frequencies and baud rates. Energy-optimal frequencies per baud rate are marked with an ×. Generally, in the example, the lower the clock frequency, the less energy is consumed. However, for each baud rate, there is a lower limit to the clock frequency, after which the energy consumption increases again. This is due to the fact that if the clock frequency is too low, the time it takes to complete an I/O operation increases, thus causing a higher energy consumption. At higher baud rates, the optimal clock frequency is higher than at lower baud rates. In the conducted test, the optimal clock frequency varies between 2 and 6 MHz. As a result, a frequency of 4 MHz is chosen for the I/O clock configuration for WATWA-OS.

5.3.4 Clock Configuration Parameters

Now that two clock configurations have been selected, it is necessary to determine the energy and timing values needed for the analysis and optimization steps of WATWA-OS. This includes (1) the maximum power consumption at each clock configuration, (2) the maximum energy and timing costs of switching from one clock configuration to another, and (3) the maximum energy and timing costs for each device syscall at each clock configuration.

This data is determined by evaluating different scenarios and using the worst observed power, timing, and energy values. Finally, the determined values are added to a PML configuration file that `platin` can read and use for the ILP construction. Table 5.2 displays all determined values, and all previous findings become apparent.

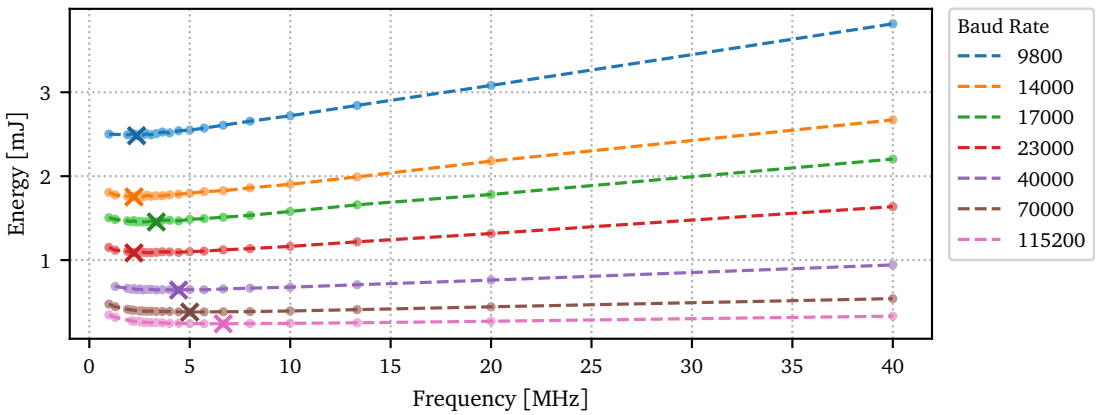


Figure 5.6 – UART transmission at different clock frequencies and baud rates.

Table 5.2 – Worst observed measurements

Frequency	Max. Power	`to_low_freq()`	`to_high_freq()`	`uart_write()`
low (4 MHz)	26.87 mW	0.449 µJ 18 µs	0.559 µJ 22 µs	2.85 µJ 117 µs
high (160 MHz)	101.951 mW	5 µJ 103 µs	0.205 µJ 2 µs	7.95 µJ 95 µs

Before `platin` is able to analyze the WCRE of an application, it still needs to know the maximum duration of each OS scheduling syscall in clock cycles. Note that `platin` can derive the actual energy and time values from this data with knowledge of the clock frequency and maximum power consumption. WATWA-OS currently supports three scheduling syscalls, whose execution time depends on the number of tasks in the system. Consequently, the execution time of each of these syscalls can be split into a part that must run every time and a part that linearly depends on the number of tasks. This results in the following formula per syscall:

$$\#cycles(\text{syscall}) = \#baselineCycles(\text{syscall}) + (\#loopCycles(\text{syscall}) \cdot \text{numTasks})$$

This formula has been integrated into WATWA-OS for the relevant scheduling syscalls. The missing parameters *#baselineCycles* and *#loopCycles* were determined by executing the syscalls on multiple OS instances with different task sizes. To determine the number of cycles per syscall invocation, the performance counter registers of the ESP32-C3 were used (i.e., `mpcer`, `mpcmr`, and `mpccr`), which can be configured to count the number of clock cycles [7].

The measured parameters were the last missing piece to properly evaluate WATWA-OS on the target hardware platform. The following section evaluates an example application scenario on the ESP32-C3 and tests the quality of WATWA-OS' clock configuration predictions.

5.4 Quality of WATWA-OS' Optimization Results

Now that all necessary values have been determined for the ESP32-C3, WATWA-OS can analyze and optimize OS instances. The first tested application consists of one task that includes a compute-intensive phase and an I/O phase where the results are transferred via the UART protocol. The source code of the application is presented in Listing 5.1. First, the `computation()` method is called in a loop

```
1  #pragma os task "start=true,priority=1,id=1"
2  void task1() {
3      #pragma loopbound min COMPUTATION_LOOPS max COMPUTATION_LOOPS
4      for (int i = 0; i < COMPUTATION_LOOPS; i++) {
5          computation();
6      }
7
8      #pragma loopbound min IO_LOOPS max IO_LOOPS
9      for (int i = 0; i < IO_LOOPS; i++) {
10         uart_write();
11     }
12 }
```

Listing 5.1 – Tested application

that is executed COMPUTATION_LOOPS times. In the tested application, this computation corresponds to one multiplication. Then, the `uart_write()` method, which transfers one character at a baud rate of 115,200 bit/s, is called as many times as set in IO_LOOPS. The loops are annotated with the `loopbound` pragma, which `platin` uses to add upper bound constraints for the corresponding variables in the ILP. The task is annotated with the `os task` pragma. LLVM can parse this pragma to automatically create suitable header files used in the creation of the OS instance.

5.4.1 Configuration Changes at Task Granularity

WATWA-OS can consider clock-configuration changes at different granularity levels. At the coarsest level, configuration changes are considered per task. In the example above, this means that WATWA-OS may decide that it is more energy efficient to run `task1` at a high frequency or at a low frequency. If it decides that `task1` should be run at a high frequency, then no calls are added, as every OS instance starts at the high clock frequency configuration. If, on the other hand, WATWA-OS decides that the task is more energy efficient at a low clock frequency, then it will add a call to the `to_low_freq()` method at the start of the task.

Figure 5.7 shows the maximum energy consumption determined by WATWA-OS compared to the actual energy consumption measured on the ESP32-C3. The number of I/O-loop iterations is fixed at 20, while the number of compute-loop iterations varies. The lines refer to different scenarios in the multi-scenario ILP generated by WATWA-OS. The dotted blue line corresponds to the scenario where the frequency is not changed, i.e., the task is executed at a high frequency. The solid red line, on the other hand, corresponds to the scenario where the clock configuration is changed to a low clock frequency at the beginning of the task.

The first observation that can be made is that before a certain number of iterations of the compute loop, it is more energy efficient to execute the task at a low clock frequency. This is because the calls to `uart_write()` are far more energy efficient at lower clock frequencies. Only after a considerable amount of multiplications is it more energy efficient to stay at the initial high clock frequency. WATWA-OS predicts that before about 350 multiplications, it is worth switching to a low clock frequency at the beginning of the task. After that, it is more energy efficient to stay at the initial high clock frequency. On the test system, this intersection occurs at a later point, at ∼680 multiplications. This is due to the power and energy values that WATWA-OS uses. Since the goal of WATWA-OS is to optimize the WCRE of the system, it assumes the worst power consumption for every executed instruction. Therefore, the slope for each line in Figure 5.7 is steeper for WATWA-OS

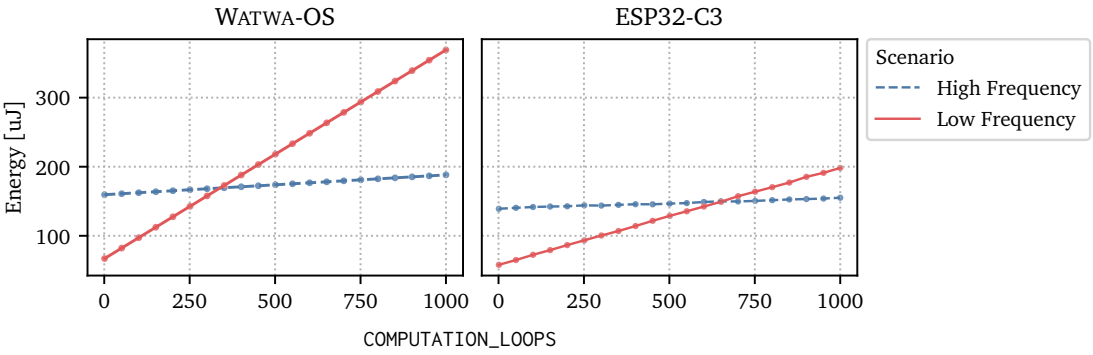


Figure 5.7 – Example application with different values of COMPUTATION_LOOPS (IO_LOOPS = 20).

than for the ESP32-C3 readings since WATWA-OS predicts that each computation consumes more energy than it actually does. Still, for a low enough number of multiplications, WATWA-OS is able to achieve considerable energy savings by switching to a lower clock frequency while ensuring that the consumed energy of the system remains below the calculated WCRE.

When using the configuration alternatives proposed by WATWA-OS, i.e., executing the task at a low frequency below 350 multiplications, then the overestimations of the energy consumption are as follows: Between 0 and 350 iterations of the compute loop, the energy predictions by WATWA-OS are 13.6 % - 16.3 % higher than the actual measured results. When `COMPUTATION_LOOPS` is greater than 350, WATWA-OS executes the task at a high clock frequency. In this case, the overestimations range from 68.9 % to 89.8 %, increasing with the number of multiplications in the compute loop.

The presented results show that when only considering task-level clock-configuration changes, WATWA-OS is capable of minimizing the system's energy demand in many cases. The following section shows how these results can be further improved by considering more clock-configuration decision points.

5.4.2 Configuration Changes Before and After Loops

WATWA-OS can also consider clock-configuration changes before and after loops that contain a device syscall. This can be particularly effective for the presented example application since the I/O loop and the compute loop would be able to run at different clock frequencies.

In this test, WATWA-OS is configured to only consider clock-configuration changes before and after syscall loops. There are two relevant scenarios: One where the I/O loop is executed at a high frequency, and thus the entire task is executed at a high frequency, and one where the system switches to a low clock frequency before executing the I/O loop. Figure 5.8 shows the predicted and measured energy consumption for the two relevant scenarios. The x-axis indicates the number of compute-loop iterations. In the upper half, only one character is transmitted via UART, while in the lower half, the I/O loop is executed twice.

For the transmission of only one character, WATWA-OS predicts that it would be slightly more energy efficient to remain at a high clock frequency. This is confirmed by measurements on the

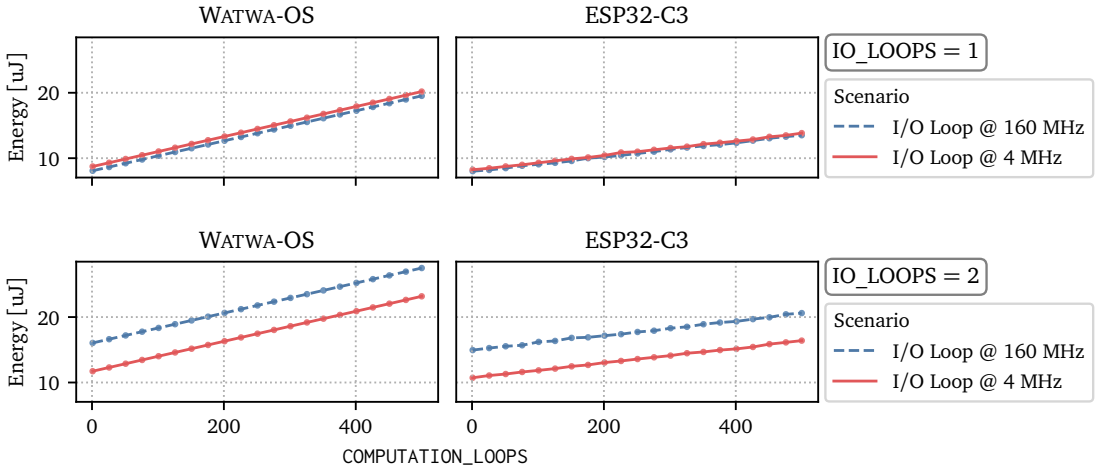


Figure 5.8 – Example application with different values of `COMPUTATION_LOOPS` at 1 and 2 I/O Loop iterations. The I/O Loop is either executed at a high or at a low frequency.

actual test system, where transmitting a single character over UART is slightly more energy efficient at a low clock frequency. At two or more iterations of the I/O loop, the difference between the low-frequency and high-frequency clock configuration become more apparent. In this case, the energy savings of switching to a low-frequency clock configuration are far more significant.

As in the previous test, WATWA-OS can predict that at a certain number of characters transmitted via UART, it is beneficial to switch to a low-clock configuration. It is able to make these predictions while also guaranteeing upper bounds on the WCRE. The overestimations in this test range from 11 % to 30 %. As stated in the previous section, the overestimation increases with the number of iterations of the compute loop, as WATWA-OS assumes that the system consumes more power than it actually does. This test only considered clock-configuration changes before and after loops. When also taking into account possible changes at the task level, then, for the presented application scenario, it is still always more efficient to only switch to a low clock configuration before the I/O loop.

5.4.3 Effect of Interrupts

Considering the effect of interrupts during WCRE analysis can get very complex. On the one hand, the analysis toolkit must be aware that at every interruptible node in the PSTG an interrupt may occur. On the other hand, overestimating how often an interrupt occurs in the worst case should be avoided to ensure tight WCRE bounds and, thus, good clock-configuration predictions.

In Figure 5.9, the previously presented example application has been extended by a periodic timer interrupt that occurs every 5 ms. When this interrupt is triggered, an ISR is called that performs a variable amount of multiplications. The parameters `IO_LOOPS` and `COMPUTATION_LOOPS` of `task1` have both been set to 100. As in the previous section, WATWA-OS considers two scenarios: One where the entire application is executed at a high frequency, and one where the system switches to a low frequency before executing the I/O loop.

Figure 5.9 compares the results of WATWA-OS to the ones measured on the ESP32-C3. The measurements show that there is a number of multiplications in the ISR at which it is more energy efficient to execute the entire application at a high clock frequency. This is due to the fact that the more multiplications are performed in the ISR, the longer it takes to execute `task1`. In turn, this makes it possible for more timer interrupts to occur, further prolonging the execution time of the application. This effect can be observed in the figure both for WATWA-OS and the ESP32-C3. In both cases, there are visible steps in the graph, which occur every time an additional interrupt occurs.

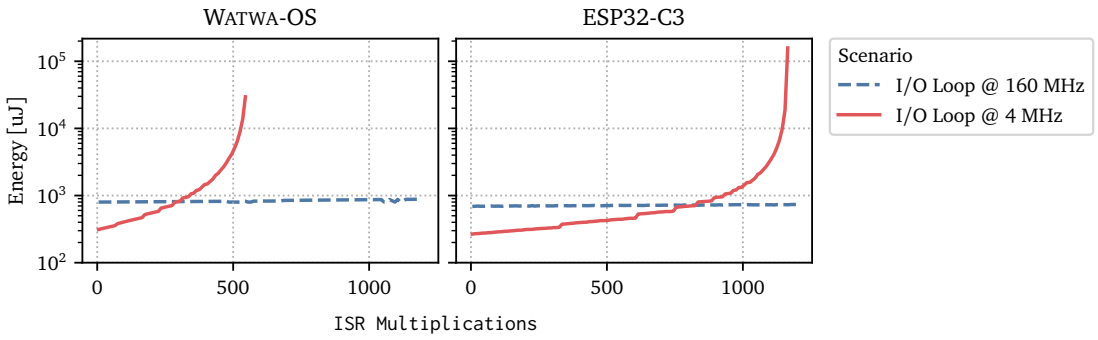


Figure 5.9 – Energy consumption (logarithmic scale) of the example application with an ISR that performs multiplications in a loop (`ISR Multiplications`). Each curve is fitted over the measured data points (one data point every 10 multiplications). The parameters `IO_LOOPS` and `COMPUTATION_LOOPS` of `task1` are both set to 100. The interrupt has an IAT of 5 ms.

Moreover, in both cases, there is a maximum number of possible multiplications in the ISR, after which the execution time of the interrupt is longer than the IAT, and thus, the application never terminates.

On the ESP32-C3, switching to a low-frequency clock configuration is beneficial up to ∼830 multiplications. After that, it is more energy efficient to stay at the initial clock frequency of 160 MHz. WATWA-OS predicts that this point happens much sooner, at ∼310 multiplications. Again, this is because WATWA-OS assumes worst-case energy values, resulting in overestimations of the WCRE. A possible solution to improve WATWA-OS predictions in this case is to reduce the complexity of the ISR by moving computations into special, high-priority tasks. WATWA-OS would then be able to consider a clock-configuration change at the start and end of these tasks.

5.4.4 Summary

This section compared the optimal clock configurations as predicted by WATWA-OS to the measured results on the hardware platform. The results show that WATWA-OS is capable of determining clock configurations that decrease the energy consumption of the system. The quality of the results varies depending on the granularity used to create clock-configuration decision points (i.e., task-level or loop-level granularity). Especially when looking at mixed-workload tasks, i.e., tasks that consist of I/O- *and* compute-intensive workloads, it is better to consider clock reconfigurations at a finer granularity. This has been shown in the tested application scenario. When only considering clock-configuration changes at the task-level, WATWA-OS does not always provide the best possible results (cf., intersection points in Figure 5.7). However, if clock reconfigurations before and after I/O loops are also allowed, WATWA-OS' predictions get significantly better, as shown in Section 5.4.2. When the analyzed system contains one or more interrupts, it is best to move complex logic into separate, high-priority tasks (see Section 5.4.3) to obtain better clock-reconfiguration predictions.

WATWA-OS is a static approach that considers all possible system states. Its analysis and optimization mechanism is split up into several steps, spanning multiple software components. The following section focuses on the execution time of these steps.

5.5 Analysis and Optimization Speed

The analysis and optimization speed of WATWA-OS depends on several factors, most notably the number of scenarios in the multi-scenario ILP that is solved by Gurobi. However, the creation of the PSTG and the construction of the resulting ILP by `platin` can also take a significant amount of time, depending on the application to be analyzed. This section discusses the analysis and optimization duration for multiple test applications.

All tested applications consist of a start task *t1* similar to the one shown in Listing 5.1, i.e., it is split up into a compute- and an I/O-intensive phase. All applications also have a second task *t2* that either does some simple multiplications ($t2_{simple}$) or that communicates via UART ($t2_{complex}$). Finally, the tested applications can have one or multiple periodic ISRs, which activate the second task $t2_{simple}$ or $t2_{complex}$.

Table 5.3 – `platin` analysis time for different applications

Application	PSTG Nodes	Total Time	PABB WCET Calculation	Node Constraints	Loops
$t2_{simple}$, 1 ISR	199	2.947 s	0.918 s	0.348 s	1.096 s
$t2_{simple}$, 2 ISRs	299	25.731 s	0.906 s	4.27 s	18.17 s

Table 5.4 – Compile Time for different applications

Application	PSTG Nodes	Total Time	Scheduling States	Device States	Loop Detection
$t2_{simple}$, 1 ISR	199	0.692 s	0.15 s	0.15 s	0.14 s
$t2_{simple}$, 2 ISRs	299	1.949 s	0.35 s	0.31 s	1.05 s
$t2_{complex}$, 1 ISR	664	30.810 s	1.80 s	1.56 s	26.82 s
$t2_{complex}$, 2 ISRs	1104	1100 s	4.74 s	3.82 s	1054 s

5.5.1 LLVM

For WATWA-OS, LLVM has been modified to be able to construct the PSTG needed for further analysis. It transforms the initial control-flow graph of the source code file into a graph containing PABBs, which is then used to enumerate all possible system states. The PSTG construction mechanism can be divided into three main time-consuming steps: (1) Scheduling State Enumeration, (2) Device State Enumeration, and (3) Loop Detection. The first two steps are necessary to construct a complete PSTG that covers all possible system states. The loop detection is necessary to ensure that each loop can only be activated by valid entry edges (cf., Section 3.1.1).

Table 5.4 shows the compilation time for applications of varying complexity. All applications have been compiled with flags to create clock-configuration alternatives before and after syscalls and at the start and end of a task. The total compilation time is broken down into the three major time-consuming steps mentioned above. As expected, the compilation time increases with the number of nodes included in the final PSTG. In almost all cases, the loop-detection mechanism makes up the majority of the compilation time. This is due to the fact that adding interrupts can significantly increase the number of loops in the PSTG that have to be considered. In the case of the most complex application ($t2_{complex}$, 2 ISRs), WATWA-OS detects over 1.5 million loops, which results in a final PML file size of around 1 GB.

For most of the tested applications, the compilation time stays below 1 minute. For more complex applications, this time increases with the number of interrupts in the system. Still, the compilation time of 1100 s for the most complex tested application is acceptable, as the compilation only needs to be performed once before system runtime. Currently, the loop-detection mechanism is implemented using the algorithm presented by Johnson [12], whose runtime depends on the number of loops in the graph. Possible improvements to the current implementation are discussed in Chapter 6.

5.5.2 `platin`

After creating the PSTG and storing all information in a PML file, `platin` is called to construct all variables and constraints needed for the final ILP. As with LLVM, multiple time-consuming steps are involved when constructing the final ILP. This includes (1) the calculation of WCET values for each PABB, (2) the construction of structural node and edge constraints, and (3) the creation of loop constraints. Table 5.3 lists the analysis time of the `platin` toolkit for two of the applications that were examined before.

Just as with the execution of LLVM, handling loop constraints makes up the majority of `platin`'s runtime. This is because `platin` has to check for every entry edge into a loop L if that entry edge is actually reachable from outside the loop. If so, it is considered a real entry edge into loop L. Otherwise, i.e., if the edge is only reachable from a node inside loop L, it is not considered a real entry edge. One example of this is an interrupt that interrupts a node inside a loop L_1. This interrupt creates another loop L_2 with the interrupted node. However, the edge from the interrupt node to the interrupted node must not be considered as entry edge to loop L_1.

Table 5.5 – Optimization Time for different numbers of scenarios

ILP Scenarios	ILP Variables	Optimization Time
4	178	0.02 s
64	915	2.50 s
256	933	6.87 s
1024	951	102.91 s

Table 5.6 – Optimization Time for different numbers of variables

ILP Scenarios	ILP Variables	Optimization Time
64	915	2.50 s
64	1472	0.85 s
64	2156	3.15 s
64	2614	5.59 s

The number of nodes and loops included in the PSTG plays a major role in the execution time of `platin`. Further details on how to possibly reduce this number and speed up the loop-detection process are discussed later in Chapter 6.

5.5.3 WATWA-OS Optimizer

After `platin` has constructed all necessary variables and constraints for the ILP, the WATWA-OS Optimizer is called. It creates a multi-scenario ILP with all possible clock-configuration scenarios using the Python interface of the Gurobi solver [8]. After all scenarios have been solved, the optimizer determines the scenario with the best, i.e., lowest, WCRE bound. The active clock-configuration nodes and edges in the corresponding PSTG of the best solution represent the worst-case optimal clock-configuration decisions of the analyzed system.

The total solving time of the multi-scenario ILP depends on multiple factors, as seen in Tables 5.5 and 5.6. In Table 5.5, the same application has been compiled using multiple flags, which results in different amounts of clock-configuration alternatives. The total optimization time in this example increases with the number of scenarios in the ILP. Note that each new clock-configuration alternative quadruples the number of scenarios in the ILP. For instance, when considering a clock-configuration change before and after a device syscall, then four new scenarios must be considered:

1. Switch to the low-frequency clock configuration before the device syscall and do not switch back afterwards.
2. Switch to the low-frequency clock configuration before the device syscall and switch back to the high-frequency configuration afterwards.
3. Do not change the clock configuration before the device syscall, but switch to the high-frequency configuration afterwards.
4. Do not change the clock configuration at all.

It is not sufficient to view each configuration alternative in the ILP in isolation since previous decisions may affect the timing and energy behavior of the following nodes in the PSTG. Consequently, it is necessary to have a system-wide view that considers all possible system states and all possible permutations of clock configurations. However, as shown in Table 5.5, this quickly leads to a

large number of scenarios that need to be solved and thus significantly increases the optimization time. Therefore, several compiler flags allow the application developer to reduce the number of clock-configuration decision points considered. For example, if the number of scenarios and thus the solving time becomes too large, the scenario count can be reduced by considering only clock-configuration changes at the task level or only before and after loops.

For the tested application scenarios in Table 5.5, the optimization time stays well below one minute for 256 or fewer ILP scenarios. At 1024 ILP scenarios, the optimization takes about 1.5 minutes, which is still acceptable. With this number of ILP scenarios, it is possible to consider a clock-configuration change at the task level for a system with five tasks.

Table 5.6 shows the effect of increasing numbers of variables on the solving time of the ILP. In this case, multiple applications have been evaluated, each of which has the same number of clock-configuration alternatives, but differs in the number of variables and constraints in the final ILP. Generally, the solving time increases with the number of variables in the ILP. However, this is not always the case, as shown in the second row. Here, the solving time is relatively fast at 0.85 s compared to an ILP with fewer variables that takes 2.50 s. There are multiple reasons for these fluctuations in the solving time. For instance, the solver may find infeasible values and paths more quickly for some ILPs due to more restrictive constraints. Furthermore, Gurobi uses a random seed for every invocation of the solver, which typically leads to different solution paths. Because of this, there can be significant solving-time fluctuations, even for the same ILP. Finally, there is a multitude of configurable parameters that influence the execution of Gurobi. For all tested ILPs of WATWA-OS, the best solving times were achieved when setting Gurobi's `MIPFocus` parameter to 1. This high-level parameter instructs Gurobi to find feasible solutions more quickly.

The results in Tables 5.5 and 5.6 show that the optimization time of WATWA-OS depends on the number of variables and scenarios in the mathematical optimization problem. For systems with limited complexity, i.e., only a few ILP scenarios and variables, this time stays below 1 minute. For more complex systems with larger task sets, the number of ILP scenarios and variables increases, and so does the optimization time of WATWA-OS. Still, the optimization time is acceptable, as this step has to be performed only once before system runtime.

5.5.4 Summary

This section showed that the analysis and optimization time of WATWA-OS depends on multiple factors. An important factor is the number of scenarios in the multi-scenario ILP. To achieve reasonable solving times, this number should be kept as low as possible. Therefore, the application developer has the possibility to restrict the granularity at which WATWA-OS considers a clock-configuration change via compiler flags. Another important factor is the number of nodes in the PSTG, which also contributes to the solving time, as seen in Section 5.5.3.

The following section evaluates two mechanisms employed by WATWA-OS to decrease the number of PSTG nodes and to reduce the total ILP solving time.

5.6 Reducing Optimization Time

WATWA-OS implements two approaches that reduce the optimization time of an OS instance. First, it tries to reduce the number of nodes in the PSTG by merging together multiple sub-graphs that belong to the same interrupt. Second, it uses a special feature of the commercial Gurobi solver, called multi-scenario ILPs, to reduce the overhead of respecifying all variables and constraints for each clock configuration scenario. In the following, the effect of these two approaches is evaluated.

Table 5.7 – Effect of node merging on variable and constraint count as well as the solving time

Tested Application	Type	Variables	Constraints	Solving Time
$t2_{simple}$, 1 ISRs	normal	143	134	0.008 s
	merged	119 (−16.8 %)	112 (−16.4 %)	0.012 s (+50 %)
$t2_{simple}$, 2 ISRs	normal	178	170	0.025 s
	merged	136 (−23.6 %)	138 (−18.8 %)	0.005 s (−80 %)
$t2_{complex}$, 1 ISR	normal	612	608	0.374 s
	merged	512 (−16.3 %)	518 (−14.8 %)	0.123 s (−67.1 %)
$t2_{complex}$, 2 ISRs	normal	855	924	1.019 s
	merged	655 (−23.4 %)	744 (−19.5 %)	0.043 s (−95.8 %)
$t2_{complex}$, 3 ISRs	normal	1098	1280	0.860 s
	merged	798 (−27.3 %)	1010 (−21.1 %)	0.081 s (−90.6 %)

5.6.1 Node Merging

WATWA-OS offers the option to merge similar interrupt-induced sub-graphs in the PSTG to reduce the number of nodes and constraints in the final ILP, as detailed in Section 3.3.2. This merging mechanism can be applied to all applications with at least one interrupt, as only interrupt sub-graphs are merged together. The impact of node merging on the number of variables and the solving time is shown in Table 5.7 for multiple tested applications.

As expected, the number of variables and constraints decreases in all tested applications when the merging mechanism is used. However, the number of constraints does not decrease as much as the number of variables because some original constraints are still needed to maintain the accuracy of the ILP solution. For almost all tested application scenarios, a significant decrease in solving time of up to 95.8 % has been measured when using the merging mechanism. This is not only due to the reduction in the number of variables. Instead, the number of constraints per variable increases when the merging mechanism is used, thus limiting the number of solution paths in the ILP. This is because, for every interrupt in the PSTG, there are now multiple edges leading from an interrupted node to the ISR entry node. Without node merging, this is not the case since every interrupted node has its own ISR entry node. The same applies to the interrupt exit nodes.

The presented node-merging mechanism adds an overhead of 10 % - 20 % to the PSTG construction time, depending on the complexity of the graph. However, this additional overhead is well worth it, especially when many ILP scenarios have to be solved, as the time spent per ILP can be significantly reduced. The results show that the solving time can be reduced by up to ∼90 % per ILP, underlining the effectiveness of WATWA-OS' node-merging mechanism.

5.6.2 Multi-Scenario ILPs

Instead of solving multiple single ILPs, WATWA-OS uses one multi-scenario ILP to find the lowest WCRE among multiple clock configuration possibilities. First, a base ILP model containing all variables and constraints that have been constructed by `platin` is defined. Then, for every scenario, certain edges are deactivated by modifying the upper bound of their variables in the ILP. In this way, only a few modifications to the base model need to be made for each scenario. On the other hand, when using multiple single-scenario ILPs, variables and constraints of the PSTG must be added to the model in each instance.

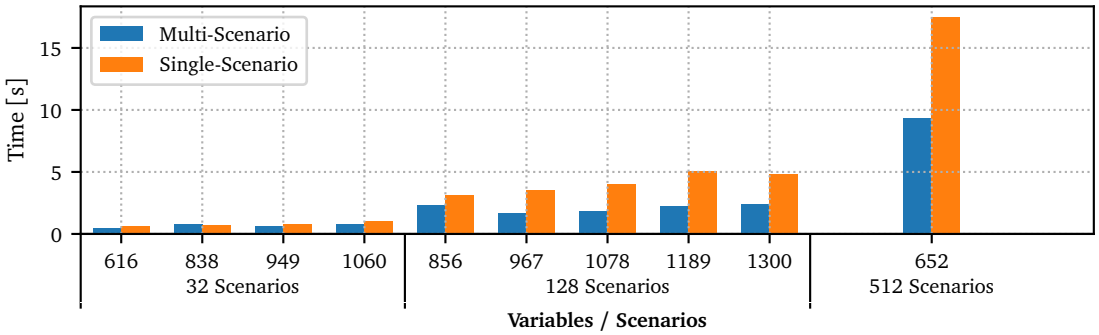


Figure 5.10 – Solving Time: Single- vs. Multi-Scenario ILPs

The benefit of multi-scenario ILPs in comparison to single-scenario ILPs is shown in Figure 5.10. In this test, several applications with different numbers of scenarios and variables were tested, and the total execution time of the WATWA-OS Optimizer was measured. The Optimizer has been extended to support the construction of multiple single-scenario ILPs. The results show that the execution time of the Optimizer is generally faster when using a multi-scenario model than when using multiple single-scenario ILPs. Note, however, that this increase is mainly due to the additional overhead of adding the same variables and constraints for each ILP. The actual solving times, i.e., the time to solve one multi-scenario ILP and the cumulative solving time of all single-scenario ILPs, are very similar in both cases. The execution time of the Optimizer when using single-scenario ILPs is up to 2.2× the execution time of the multi-scenario variant. In most tested scenarios, the execution time has been significantly reduced when using multi-scenario ILPs.

5.7 Conclusion

WATWA-OS is able to optimize the energy consumption of a system by changing the clock configuration at certain points during its execution while also guaranteeing upper bounds on the WCRE. This evaluation showed that it comes close to real determined optimal clock-configuration changes. However, since WATWA-OS uses worst-case energy values, it does not predict the exact points at which a clock-configuration change actually saves the most energy.

The main drawback of WATWA-OS in its current form is the analysis and optimization time, which increases with the number of nodes and clock configuration alternatives in the PSTG. WATWA-OS provides the ability to reduce the number of clock-configuration decision points considered in the graph, e.g., by limiting the granularity at which a clock configuration alternative is inserted. While this may lead to less optimal solutions, the optimization speed can be significantly increased, as shown in Section 5.5.3. Another solution to the problem of long optimization times is the presented node-merging mechanism and the use of multi-scenario ILPs. The node-merging mechanism is especially useful when the application consists of multiple interrupts. Multi-scenario ILPs, on the other hand, show their strengths when many clock-configuration alternatives are considered, as they significantly reduce the overhead of respecifying ILP variables and constraints. Further ideas on how to reduce the optimization time in the future are discussed in the next chapter.

6 DISCUSSION AND FUTURE WORK

WATWA-OS seeks to find optimal clock configuration points during system execution while guaranteeing upper WCRE bounds. The evaluation showed that there are still some areas in which the optimization results and speed can be improved. This chapter discusses some of the drawbacks of WATWA-OS and explores how these problems might be addressed in future work.

Loop Handling

WATWA-OS currently spends most of its time finding loops in the PSTG of a system and determining valid entry edges into these loops. The problem is that two loops that share one or more nodes can activate each other in the solution found by the solver without there being an actual path to those loops. This means that the solver might take one path through the graph but also take two loops that are located somewhere completely different. The current solution to this problem is to add constraints to the ILP, which state that each loop can only be activated by an entry edge that is reachable from outside the loop (see Section 3.1.1). This is currently implemented in LLVM and `platin`. LLVM takes care of determining all loops in the PSTG, while `platin` finds all valid entry edges into the loop and creates the appropriate ILP constraints.

Finding all loops in the PSTG is implemented using the algorithm of Johnson [12], which finds all elementary circuits in a directed graph. The main drawback of using this approach is that it does not understand the semantics of the underlying graph. A more efficient implementation could make use of the already computed loop information provided by other LLVM passes. In addition, the semantics of the Operating System can be used. For example, an interrupt generally creates a loop at the interrupted node. However, while this approach is more specific, it also has to account for more edge cases. For instance, an interrupt does not necessarily always return to the interrupted node since the interrupt may have changed the system state. Similarly, valid entry edges can be found by considering the semantics of the PSTG.

Improving Clock Configuration Decisions

As shown in the evaluation, WATWA-OS does not necessarily detect the optimal points at which a frequency change is most energy efficient. This is because its goal is to find worst-case optimal solutions. Hence, WATWA-OS internally uses the worst-observed energy and timing values. The quality of the clock-configuration changes proposed by WATWA-OS depends heavily on these configurable worst-case parameters.

However, when worst-case energy bounds are not a concern, the efficiency gains of WATWA-OS can be improved by using the average energy consumption as a parameter for clock configurations. In the example in Figure 5.7, this would reduce the slope angle of the low-frequency scenario,

thus moving the point at which WATWA-OS switches to a lower clock frequency closer to the actual measured point.

ILP Scenarios

Another time-consuming step of WATWA-OS is solving the multi-scenario ILP. The number of scenarios grows exponentially with the number of considered clock configuration alternatives. To keep this number as small as possible, it is currently advisable to only consider frequency changes at the task level or only before and after loops with device syscalls.

The reason why multiple ILPs need to be solved is that the node and edge frequencies may change depending on the clock configuration. Since the clock configuration can now change at various points in the system, these node frequencies must be recalculated for every permutation of clock configuration alternatives.

However, it may still be possible to get good results without solving all possible scenarios. A new approach could calculate the WCRE for only one base ILP, e.g., for an ILP that does not change the clock frequency at all. The solution of this ILP contains all node frequencies for this specific case. Afterwards, a new ILP could be formulated where the node frequencies are already pre-filled with the results of the first ILP. In this new ILP, all clock configurations are possible, i.e., no constraints disable a particular clock configuration alternative. This ILP can then have a minimization objective to find the lowest energy consumption. As a result, this ILP finds out which of the clock configuration alternatives should be taken. While these alternatives may not reflect the true worst-case optimal solution, they may be close enough to provide a good approximation. This new approach would ultimately trade worst-case optimality for optimization speed and is a topic for future research.

Node Merging

It has been shown that the node merging mechanism employed by WATWA-OS can significantly reduce the number of variables and constraints in the resulting ILP and thus decrease the solving time. This mechanism benefits not only WATWA-OS but also the original SysWCEC approach that WATWA-OS adapts.

Currently, WATWA-OS only merges subgraphs that are induced by interrupts and the nodes that belong to them. This prevents the PSTG from including many similar nodes that differ only in their scheduling state, i.e., from which node they were called. This merging mechanism could be extended in the future to also merge different task activations together. For instance, if a high-priority task is activated at different places in the source code, then all of these activations could point to the same starting node of the task. Currently, all nodes of the high-priority task would be duplicated for each activation instead.

Operating System

WATWA-OS offers a simple operating system with a fixed-priority preemptive scheduler and support for interrupts. Tasks can be activated and terminated via syscalls, but there are currently no synchronization mechanisms available. The previous SysWCEC approach showed that more complex systems are possible and that synchronization mechanisms and messaging support can be integrated without significant changes in the analysis approach. For WATWA-OS, the main way to add OS functionality would be to extend the system state enumeration methods in the LLVM compiler framework, as these methods have knowledge of the OS semantics and use them to create the final PSTG.

Finally, to support industry standards such as OSEK-OS [17], the current OS configuration with pragmas can be extended to support more parameters. Another approach would be to introduce a new configuration file format, as recommended by OSEK-OS, where all necessary OS objects and parameters are statically defined.

Hardware Platform

WATWA-OS currently supports small single-core systems with in-order pipelines and no caches. However, the approach is not limited to these systems and can be extended to support more complex hardware platforms. For example, there are already approaches that are able to employ system state enumeration on multicore systems while keeping the number of states to a minimum [5]. Still, in environments where giving an upper bound on the worst-case response energy consumption is required, the hardware constraints imposed by WATWA-OS are realistic, as small single-core microcontrollers are predictable and energy efficient.

7 CONCLUSION

The goal of this work was to create a framework for static WCRE analysis and automatic clock configuration optimization for embedded real-time systems. WATWA-OS achieves this by extending an already existing approach for whole-system energy analysis called SysWCEC. Essentially, WATWA-OS considers different scenarios of clock-configuration changes that can happen at certain points during system runtime. It then analyzes the Worst-Case Response Energy Consumption (WCRE) of each scenario and chooses the clock configurations of the scenario with the lowest WCRE bound.

WATWA-OS is able to create instances of a small operating system that uses a fixed-priority preemptive scheduler and that has support for interrupts. The OS instances can be configured using special source code annotations. WATWA-OS can automatically analyze these OS instances and inject clock-configuration changes at points that result in an optimized energy consumption of the system. While only a few scheduling system calls are supported at this moment, the presented approach is also applicable to more complex systems as long as the OS semantics are deterministic. This determinism is usually required by industry standards for embedded operating systems anyway.

In comparison to dynamic optimization approaches, WATWA-OS has some considerable advantages. First, it adds no timing and energy overhead at system runtime since all clock-configuration changes are determined statically. In addition, WATWA-OS considers the transition costs of a clock-configuration change from one configuration to another. These costs are often overlooked and can cause significant overhead due to frequency stabilization phases or driver reinitialization. Most importantly, the static approach presented provides a whole-system view that ultimately helps determine whether or not a clock-configuration change actually saves energy.

Among all considered scenarios, WATWA-OS finds the worst-case optimal clock configuration points in the system. The evaluation shows that these results come close to the actual observed optimal points and that they can be further improved by tightening the WCRE bounds. The main drawback of WATWA-OS in its current form is the time it takes to analyze and optimize the system. While some employed techniques, such as the node merging mechanism, can result in a significant analysis time reduction, other parts of the analysis still take a long time. The previous chapter gave an outlook on how these challenges may be tackled in future research, e.g., by better utilizing the semantics of the underlying data structures.

In summary, it has been shown that static whole-system analysis techniques are not limited to determining upper bounds on the execution time and energy consumption of a real-time system. By taking advantage of statically available knowledge, these techniques can also be used to optimize the energy consumption of a system. In the future, the presented approach can be combined with other whole-system techniques that try to obtain more information about the control flow and data dependencies in order to further improve the optimization results.

LIST OF ACRONYMS

MCU Microcontroller Unit
DFS Dynamic Frequency Scaling
RTOS Real-Time Operating System
UART Universal Asynchronous Receiver Transmitter
GPIO General Purpose Input/Output
GPI General Purpose Input
I^2C Inter-Integrated Circuit
IoT Internet of Things
OS Operating System
PLL Phase-Locked Loop
WCET Worst-Case Execution Time
WCRT Worst-Case Response Time
WCEC Worst-Case Energy Consumption
WCRE Worst-Case Response Energy Consumption
ISR Interrupt Service Routine
IPET Implicit Path Enumeration Technique
ILP Integer Linear Program
CFG Control-Flow Graph
PABB Power Atomic Basic Block
PSTG Power-State-Transition Graph
IAT Inter-Arrival Time
CCDP Clock-Configuration Decision Point
PML Program Metainfo Language
CFRG Control-Flow Relation Graph

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

REFERENCES

[1] Mario Bambagini et al. “Energy-Aware Scheduling for Real-Time Systems: A Survey.” In: *ACM Trans. Embed. Comput. Syst.* (2016). DOI: 10.1145/2808231. URL: https://dl.acm.org/doi/10.1145/2808231.

[2] Holly Chiang et al. “Power Clocks: Dynamic Multi-Clock Management for Embedded Systems.” In: EWSN ’21. Junction Publishing, 2021. DOI: 10.5555/3451271.3451284. URL: https://dl.acm.org/doi/10.5555/3451271.3451284.

[3] Eva Dengler and Peter Wägemann. “Crêpe: Clock-Reconfiguration-Aware Preemption Control in Real-Time Systems with Devices.” In: Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. DOI: 10.4230/LIPIcs.ECRTS.2024.10. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2024.10.

[4] Christian Dietrich et al. “SysWCET: Whole-System Response-Time Analysis for Fixed-Priority Real-Time Systems (Outstanding Paper).” In: RTAS 2017. IEEE, 2017. DOI: 10.1109/rtas.2017.37. URL: http://dx.doi.org/10.1109/RTAS.2017.37.

[5] Gerion Entrup, Björn Fiedler, and Daniel Lohmann. “MultiSSE: Static Syscall Elision and Specialization for Event-Triggered Multi-Core RTOS.” In: RTAS 2023. IEEE, 2023. DOI: 10.1109/rtas58335.2023.00028. URL: http://dx.doi.org/10.1109/RTAS58335.2023.00028.

[6] *ESP32-C3 Series: Datasheet*. Version 1.7. Espressif Systems. 2024. URL: https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf.

[7] *ESP32-C3: Technical Reference Manual*. Version 1.1. Espressif Systems. 2024. URL: https://www.espressif.com/sites/default/files/documentation/esp32-c3_technical_reference_manual_en.pdf.

[8] LLC Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2024. URL: https://www.gurobi.com.

[9] Stefan Hepp et al. “The platin tool kit-the T-CREST approach for compiler and WCET integration.” In: 2015.

[10] Benedikt Huber, Daniel Prokesch, and Peter Puschner. “Combined WCET analysis of bitcode and machine code using control-flow relation graphs.” In: LCTES ’13. Association for Computing Machinery, 2018. DOI: 10.1145/2465554.2465567. URL: https://doi.org/10.1145/2465554.2465567.

[11] *ICPL2630 / ICPL2631 Datasheet*. DC93164. ISOCOM COMPONENTS. 2012. URL: http://isocom.com/wp-content/uploads/2017/08/dc93164.pdf.

[12] Donald B. Johnson. “Finding All the Elementary Circuits of a Directed Graph.” In: *SIAM Journal on Computing* 4 (1975), pp. 77–84. DOI: 10.1137/0204007. URL: https://doi.org/10.1137/0204007.

[13] *Joulescope JS220 User’s Guide Precision DC Energy Analyzer*. Revision 1.8. Jetperch LLC. 2024. URL: https://download.joulescope.com/products/JS220/JS220-K000/users_guide/.

[14] David H.K. Kim, Connor Imes, and Henry Hoffmann. “Racing and Pacing to Idle: Theoretical and Empirical Analysis of Energy Optimization Heuristics.” In: *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*. 2015, pp. 78–85. DOI: 10.1109/CPSNA.2015.23. URL: https://doi.org/10.1109/CPSNA.2015.23.

[15] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.” In: CGO ’04. IEEE, 2004. DOI: 10.1109/CGO.2004.1281665. URL: https://doi.org/10.1109/CGO.2004.1281665.

[16] Yau-Tsun Steven Li and Sharad Malik. “Performance analysis of embedded software using implicit path enumeration.” In: DAC ’95. Association for Computing Machinery, 1995. DOI: 10.1145/217474.217570. URL: https://doi.org/10.1145/217474.217570.

[17] *Operating System Specification 2.2.3*. OSEK/VDX Group. 2005. URL: https://www.irisa.fr/alf/downloads/puaut/TPNXT/images/os223.pdf.

[18] Peter P. Puschner and Anton V. Schedl. “Computing Maximum Task Execution Times - A Graph-Based Approach.” In: (1997). DOI: 10.1023/a:1007905003094. URL: http://dx.doi.org/10.1023/A:1007905003094.

[19] Phillip Raffeck, Johannes Maier, and Peter Wägemann. “WoCA: Avoiding Intermittent Execution in Embedded Systems by Worst-Case Analyses with Device States.” In: LCTES 2024. Association for Computing Machinery, 2024. DOI: 10.1145/3652032.3657569. URL: https://doi.org/10.1145/3652032.3657569.

[20] Tifenn Rault, Abdelmadjid Bouabdallah, and Yacine Challal. “Energy efficiency in wireless sensor networks: A top-down survey.” In: *Computer Networks* 67 (2014), pp. 104–122. DOI: https://doi.org/10.1016/j.comnet.2014.03.027. URL: https://www.sciencedirect.com/science/article/pii/S1389128614001418.

[21] Michel Rottleuthner, Thomas C Schmidt, and Matthias Wahlisch. “Dynamic Clock Reconfiguration for the Constrained IoT and its Application to Energy-efficient Networking.” In: EWSN ’22. Association for Computing Machinery, 2023. DOI: 10.5555/3578948.3578964. URL: https://dl.acm.org/doi/10.5555/3578948.3578964.

[22] *STM32L476xx Datasheet*. DS10198. Rev. 11. STMicroelectronics. 2024. URL: https://www.st.com/resource/en/datasheet/stm32l476rg.pdf.

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

[24] *User Manual: STM32 Nucleo-64 boards (MB1136)*. UM1724. 14.0. STMicroelectronics. 2020. URL: https://www.st.com/resource/en/user_manual/um1724-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf.

[25] Peter Wägemann et al. “Whole-System Worst-Case Energy-Consumption Analysis for Energy-Constrained Real-Time Systems.” In: ECRTS 2018. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. DOI: 10.4230/LIPICS.ECRTS.2018.24. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2018.24.

[26] Andreas Weissel and Frank Bellosa. “Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management.” In: *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. 2002, pp. 238–246. DOI: 10.1145/581630.581668. URL: https://doi.org/10.1145/581630.581668.