Appears in: Proceedings of the 46th IEEE Real-Time Systems Symposium (RTSS 2025)

WATWAOS: A Framework for Worst-Case–Aware Tailoring and Whole-System Analysis of Energy-Constrained Real-Time Systems

Tobias Häberlein, Eva Dengler, Phillip Raffeck, Peter Wägemann
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

***Abstract*—Emerging embedded systems have to increasingly meet energy constraints besides their timing requirements. While frequency-scaling techniques are well explored, existing operating systems for embedded real-time systems have shortcomings in comprehensively exploiting energy-saving features present in modern system-on-chip (SoC) platforms. Existing systems lack operating-system abstractions to exploit the tradeoff between computing performance and energy efficiency. Consequently, whole-system analysis techniques are not applicable to yield optimal configurations tailored to the applications' requirements. Finally, the complexity of modern energy-saving hardware features creates huge search spaces for optimal configurations.**

In this paper, we present WATWAOS, a framework for worst-case–aware tailoring and whole-system analysis of energy-constrained real-time systems. WATWAOS acts as both an analysis/tailoring framework and a (generated) real-time operating system. The approach exploits knowledge acquired during whole-system analysis and applies worst-case–aware tailoring of the system for its runtime. WATWAOS has an awareness of the application's requirements (i.e., deadlines, peripheral devices) and the underlying SoC's energy-saving features. To achieve the tailoring, WATWAOS introduces a concept of *hierarchical abstractions*, which offer fine-grained power-management decisions. These abstractions are designed to enable merging of their states without loss of accuracy. Static analysis based on these abstractions yields worst-case–optimal (i.e., provably energy minimal) solutions with regard to given deadlines. To tackle the enormous search space of our bilevel problem, WATWAOS employs several concepts to exploit advanced features of mathematical optimizing tools. The evaluations of WATWAOS validate our claim of finding worst-case–optimal solutions within acceptable analysis times.

***Index Terms*—real-time operating systems, resource minimization, embedded system-on-chip, whole-system analysis, ILP solving, worst-case execution time, worst-case energy consumption**

I. INTRODUCTION

Time- & Energy-Constrained Embedded Systems: The number of Internet-of-Things (IoT) systems has reached 16 billion [1], and visions predict a route to one trillion systems [2]. Many of these IoT systems are embedded in their environment, making them face both timing and energy constraints, which describe this paper's two central resources of interest. With the increase of rather safety-uncritical consumer electronics, likewise, more safety-critical (e.g., implantable) medical devices and health-monitoring systems have critical timing restrictions while also requiring energy awareness, especially when being battery-powered [3]. For example, battery-powered smart watches [4] or other wearables [5] are increasingly used for medical purposes. Besides battery-operated systems, the *Internet of Batteryless Things* [6], where energy is harvested from the environment, requires meeting both timing and energy constraints [7].

Time-vs-Energy Tradeoff in Modern SoCs: Reducing the energy demand while meeting timeliness has been mainly achieved by dynamic (voltage and) frequency scaling (DVFS) approaches, where we refer to the survey of Bambagini et al. [8]. Frequency scaling still represents the state of the art in embedded systems [9]. However, frequency scaling alone yields minor benefits on modern system-on-chip (SoC) platforms: These SoCs feature a complex intertwined *clock subsystem* with configurable energy-saving features. Starting from generating clock signals, modern SoCs have a plethora of clock sources that differ in their (temperature-dependent) stability, speed, startup time, and energy efficiency. On the opposing side of clock sources in the SoC are power-consuming components, such as (co-)processors, sensors, (analog/digital-to-digital/analog) converters, which are subsequently referred to as generic *devices*. Using these devices, in turn, potentially requires specific configurations of the clock subsystem.

Operating-System Support for Energy-Aware Real-Time Systems: Time- and energy-constrained applications inevitably require a real-time operating system (RTOS) to avoid the error-prone and labor-intensive tuning of bare-metal applications. For example, RIOT [10] is an OS with fixed-priority real-time scheduling. In the context of RIOT's energy savings, Rottleuthner et al. pointed out the necessity of clock-subsystem abstractions and their use for dynamic clock reconfigurations [11]. With the same goal, Chiang et al. introduced the Power Clocks approach to dynamically manage multiple clock sources [12] for Tock OS [13]. Since RIOT and Tock address energy savings with dynamic, feedback-based approaches, they are incapable of giving static runtime guarantees.

Dynamic Reconfiguration vs. Static Guarantees: Contrary to the dynamic approaches, Crêpe [14] and FusionClock [15] build a resource-consumption model of the clock subsystem to reduce the energy demand under real-time constraints. To give runtime guarantees and yield optimal clock configurations, the approaches have a notion of both *worst-case execution time (WCET)* [16] as well as *worst-case energy consumption (WCEC)* [17]. However, these RTOS-agnostic approaches only target periodic task sets or task sets with a single interrupt and software-controlled preemption control,

which do not meet the complexity of realistic task sets. Real-world task sets usually contain multiple asynchronously activated, sporadic tasks and, in most cases, at least timer interrupts for the RTOS' scheduler. Tackling sporadic task sets is especially difficult since this task model comes with the challenge of search-space explosion to find optimal solutions.

Whole-System Analysis & Optimization: This work has the objective of finding optimal configurations (i.e., minimum energy demand) under consideration of real-time constraints. Further, we consider sporadic task sets (i.e., multiple asynchronous interrupts) running on modern SoCs with a multi-sourced clock subsystem. As we will show, this system model leads to enormous search spaces. Asynchronous interrupts (which can, in turn, activate higher-priority tasks) inevitably lead to cycles in the entire system's execution graph. A common approach to handling the analysis and optimization of such directed, cyclic graphs is to map them to a *maximum-cost flow problem*. This type of problem can be – by means of *integer linear programming (ILP)* techniques – solved with powerful mathematical, linear-programming solvers (e.g., Gurobi [18]). Being driven by the relevance of, for example, business decision-making, these tools have been substantially improved over the last decades [19], [20]. Although these solvers are highly optimized and able to exploit the parallelism of multi-core/cluster environments, naïvely formulating ILPs and integrating them with ILP solvers potentially results in poor solving performance. Likewise, choosing abstractions that are too fine-grained (e.g., on the level of single machine-code instructions) leads to practically infeasible results. As part of this paper's optimization-related contributions, we leverage features (e.g., multi-scenario ILPs [21], model-modification callbacks [22]) to significantly reduce optimization times.

Contributions: In this paper, we introduce WATWAOS, a real-time operating-system framework that targets worst-case–aware tailoring and whole-system analysis of energy-constrained real-time applications. Our proposed contributions concern the entire system-software stack, its whole-system analysis, and several optimization techniques to provide practical analysis times:

1) **System & Abstractions**: We propose the open-source WATWAOS real-time operating-system framework [23]. WATWAOS introduces *hierarchical abstractions* that are specifically tailored to meet the complexity of modern embedded SoCs. These abstractions support our graph-merging techniques and keep analysis efforts practical.
2) **Analysis & Tailoring**: Its abstractions allow WATWAOS to conduct fine-grained static analyses for time and energy. As first of its kind, WATWAOS enables system-wide energy minimization, by using modern hardware features, of fixed-priority real-time systems with arbitrary asynchronous interrupts.
3) **ILP-Solving Techniques**: WATWAOS is the first approach for embedded systems that exploits advanced features of ILP solvers to substantially reduce analysis times, as we outline in our evaluation based on a real-world embedded SoC platform.

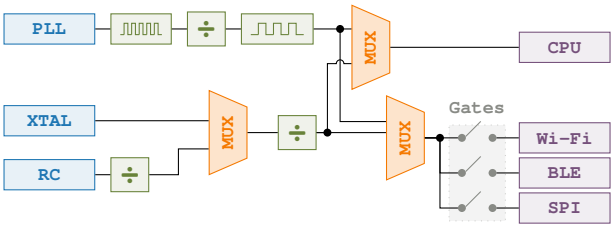

Fig. 1: The clock subsystem (simplified excerpt) is the core component for the time-energy tradeoff on modern SoCs. Multiplexers, dividers, and gates along the signal paths further configure this tradeoff besides individual input clock sources.

II. BACKGROUND & SYSTEM MODEL

We give insights into WATWAOS' background (in Section II-A) and its system model (in Section II-B).

A. Background

Clock Distribution Networks: Our targeted platforms are highly resource-constrained embedded devices. To manage the time-energy tradeoff, these systems feature a mechanism to control the system's components, called the *clock subsystem*, alternatively the *clock tree* or *clock-distribution network*. Figure 1 gives a simplified example of the clock subsystem from the RISC-V–based ESP32-C3 SoC platform. For further insight into the complexity of this subsystem, we refer to its detailed documentation [24]. The chosen SoC features a rather complex clock tree, offering a large configuration space for the time-energy tradeoff. While simpler clock subsystems exist, this SoC serves as a representative platform for this paper, highlighting the complexity of numerous SoCs. The clock subsystem consists of several *input signal nodes* routed to *output devices* such as the CPU or Wi-Fi device via a network of nodes. These nodes are *dividers* that scale the incoming signal by a factor or *multiplexers* that select one of the incoming signals as the output signal. Further, *power gating* can be applied to switch components on or off. This allows the system configuration to be realized in a very fine-grained way. The main benefit of the configurability is the possible, fine-tunable time-energy tradeoff: For example, if a device is not required to be active, the system can deactivate it by turning off the corresponding gate and thereby save energy (i.e., power over time). Another option is to run a device at a lower frequency, resulting in a different power-consumption behavior. For the CPU, this frequency reduction results in a longer execution time but also reduces the energy consumption per time unit. Note that some devices require specific system configurations. For example, the transceiver (necessary for sending Wi-Fi packets) of the ESP32-C3 requires an active high-energy clock (phase-locked loop, PLL) [24]. For more energy efficiency, the clock subsystem can configure the XTAL crystal. Another option is the use of RC oscillators, which are rather temperature-unstable but very energy-efficient.

Power-Consumption Behavior: We subsequently discuss preferable clock-frequency configurations for specific workloads. For our targeted platforms, the power consumption

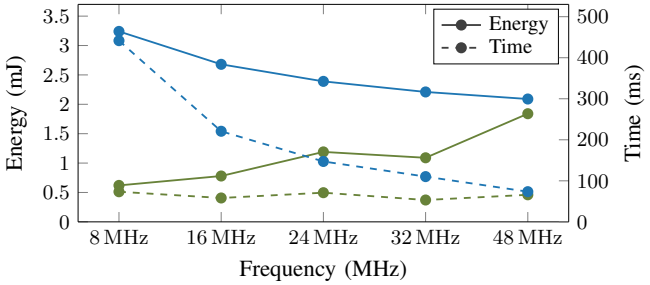

Fig. 2: Our comparison of I/O-bound (green) and CPU-bound (blue) workloads on an embedded SoC platform [25].

consists of a static and a dynamic part [8]. The dynamic part results from the switching activity within the system, while the static part is caused by leakage currents through the transistors. Two main groups of tasks are worth considering: (1) Compute-intensive tasks (also called CPU-bound workloads) should be executed at the highest possible clock frequency. While the dynamic part stays high due to the workload, the task is completed in the shortest possible time. Thereby, the time spent per operation is minimized, resulting in a low static power consumption. Since the internal memory bus is usually clocked with the same speed as the CPU in our targeted embedded SoCs [24], CPU-bound phases do not substantially differ from memory-bound phases in terms of time and power demand. Thus, memory-bound phases are part of the same group. (2) In contrast, the other group consists of I/O-intensive operations (I/O-bound), for example, communicating with external or memory-mapped sensors. Here, the time needed is usually bottlenecked by a relatively slow bus protocol compared to the CPU frequency. Therefore, the goal is to reduce the energy consumption per unit of time. The static power consumption is fixed due to the timing behavior of the I/O operation. Consequently, reducing the dynamic part by choosing a low CPU frequency results in reduced energy consumption. This behavior is shown in Figure 2 for an embedded SoC [25]: The green lines represent a workload where data is read from a connected sensor. While the time needed to perform the transaction stays consistent, the energy requirements rise with increasing CPU frequency. The blue lines, on the other hand, refer to a workload where the retrieved data is processed. Here, both time and energy decrease with an increasing CPU frequency, which motivates our work on provably energy-minimal clock-subsystem configurations.

Static Worst-Case Analysis: A major challenge for any energy-aware real-time system is to guarantee the safe execution of all tasks within both time and energy budgets. Determining suitable values for the WCET and WCEC is a necessary prerequisite for a reasonable execution strategy. For this, numerous approaches to both types of resource analyses exist [16], [17], [26], [27], [28], [29]. A common way of calculating these values is to use a bottom-up approach: The analysis starts by adding up the time of single instructions and then groups these into larger blocks. To achieve this compositionality of resource demands, we assume the absence of timing anomalies [30]. This prevails on our target platform (ESP32-C3), which we configure to bypass caches. The resulting blocks are used to construct a control-flow graph that models all applications and the OS code. Using the *Implicit Path Enumeration Technique (IPET)* [31], [32], the graph is transformed into a mathematical optimization problem. The IPET's core idea is to derive flow constraints, for example, from branches in the program's control flow, and formulate an integer linear program (ILP) from these constraints. By solving the ILP, the IPET *implicitly* derives worst-case estimates instead of explicitly enumerating all possible paths, which is not practical given the myriad of possible execution paths in whole systems. Practicality is a core goal of WATWAOS, for which solving ILP formulations efficiently is essential. While the original IPET [31], [32] operates on single-threaded control-flow graphs, WATWAOS exploits this ILP-based technique for resource minimization across global control flows (i.e., inter-task) with asynchronous activations.

Terminology for Whole-System Analysis: The term *WCET* refers to the execution time of a task in isolation. When accounting for preemptions (e.g., through asynchronous interrupts or higher priority tasks), the *worst-case response time (WCRT)* describes the end-to-end timespan from the beginning to the task's termination, including possible blocking times or interferences through preemptions. For the energy-related problems, the terminology refers to *WCEC* and *worst-case response energy consumption (WCRE)*, respectively.

B. System Model

Sporadic Task Model: WATWAOS features single-core, fixed-priority real-time systems with preemptive, sporadic tasks. Each task τ is activated by an asynchronously occurring interrupt I. The occurrence of sporadic tasks and interrupts are bounded by their *minimum inter-arrival time* $t_{min,I}$. Additionally, each asynchronous interrupt can have a *release jitter* in WATWAOS. In the case of a task with specified deadline requirements, WATWAOS ensures that the respective task τ meets its deadline D_τ. An asynchronously released interrupt leads to the execution of an associated *interrupt service routine (ISR)*. An essential interrupt for most real-time systems is the timer interrupt, which usually triggers the scheduler. WATWAOS handles the timer's ISR as any other potentially occurring interrupt with either a periodic or sporadic arrival time. As a static analysis approach, WATWAOS requires a priori knowledge of the source code, the target platform's clock tree, and the impact of device operations on the power demand. WATWAOS can also incorporate dynamic task startups as long as their code and device use is known in advance. Currently, WATWAOS does not assume shared resources (e.g., mutexes). However, this type of contention-aware analysis has been shown in several works in the context of real-time scheduling [33], [34], [35], [36]. We consider the benefits of integrating these blocking times into our model to be minor in relation to our main objective of selecting provably energy-minimal clock configurations.

Notion of Optimality: WATWAOS aims to give runtime guarantees by means of static analysis, being essential for safety-critical applications. Runtime guarantees describe that no task executes beyond its assigned worst-case execution time or worst-case energy consumption. During the actual runtime, tasks likely consume less than their acquired budget, which is of minor interest for the course of this work. Our objective is to yield provably energy-optimal results under consideration of timeliness. While the guarantees are given during design time, the WATWAOS runtime dynamically reconfigures the system based on the statically acquired knowledge. We refer to execution scenarios as being *worst-case–optimal* when no further energy reduction is possible under the assumption that tasks require their assigned worst-case budget.

III. PROBLEM STATEMENT

In alignment with our contributions, we solve three problems with WATWAOS: the lack of OS abstractions for modern clock subsystems (Section III-A), missing concepts for whole-system analysis and tailoring of energy-constrained real-time systems (Section III-B), and the problem of large search spaces for finding worst-case–optimal solutions (Section III-C).

A. Problem #1: Missing Operating-System Abstractions

Previous approaches make use of the fact that the time-energy tradeoff varies with I/O- and compute-intensive workloads. PowerClocks [12] features an operating system that dynamically tracks I/O operations and adapts the system frequency accordingly. ScaleClock [11], which builds upon RIOT [10], tests the possible configurations in an initialization phase and relies on the best findings during runtime. Although both systems include their clock reconfigurations as part of the operating system, they do not support a fine-grained and adjustable granularity of clock-subsystem reconfigurations. Zephyr [37], an operating system for embedded devices gaining more and more attention, also misses a good abstraction for the clock subsystem: Fischer [38] discussed a potential clock-subsystem integration into Zephyr. However, the proposal for a possible architecture is still a work in progress [39]. Despite the relevance and benefits of this topic, neither an implementation nor a concept on how to handle (provably optimal) energy minimization by exploiting these modern clock subsystems is available for Zephyr.

Approach of WATWAOS to Missing Operating-System Abstractions: WATWAOS leverages the existing concept of power-state–aware blocks [33], [40] and their transitions in a system-wide graph. WATWAOS extends this concept through *hierarchical abstractions*, where information on different device states is embedded into the blocks at a specific layer.

B. Problem #2: Lacking Whole-System Analysis & Tailoring

Looking at the clock subsystems in embedded hardware platforms, two questions arise regarding energy-saving strategies: Which configuration is most suitable for a sequence of instructions, and when should the OS change the configuration between sequences? Existing operating-system-level approaches, such as PowerClocks [12] and ScaleClock [11], address these questions by dynamically applying configurations based on whether the current task is considered to be I/O- or compute-intensive. However, when combining both I/O and compute workloads, clock reconfigurations introduce additional runtime overheads. For example, when switching from a compute-intensive task, which might operate at a high CPU frequency, to an I/O-heavy task, with an optimal low CPU frequency regarding energy savings, additional, potentially substantial reconfiguration penalties are involved. Therefore, constantly changing to the best clock configuration for the isolated workloads can lead to additional energy-consumption penalties when considering the entire schedule. As a consequence, both approaches neglect the impact of the interaction of multiple tasks and, therefore, make suboptimal decisions. To consider the best possible clock configurations together with the resulting reconfiguration penalties, a whole-system analysis and optimization approach is needed to be able to handle all possible options. The challenge is to manage the whole system with all possible states and to determine the energy-optimal clock-configuration settings. The problem regarding analysis optimization intensifies when moving beyond periodic task systems as analyzed in FusionClock [15] or Crêpe [14]. As WATWAOS optimizes whole operating systems with sporadic task sets, which adds more complexity to the analysis, further optimization tailoring is necessary to succeed.

Approach of WATWAOS to Lacking Whole-System Analysis & Tailoring: WATWAOS enables system-wide energy minimization by considering multiple points during runtime at which clock reconfigurations could be beneficial. Our novel approach solves multiple mathematical optimization problems to determine a sequence of reconfigurations that yields the system's worst-case–optimal energy consumption.

C. Problem #3: Search Spaces & Unpractical Analysis Times

Because WATWAOS aims to analyze whole systems, there is a rapidly expanding number of possible nodes. In particular, considering possible system-state reconfigurations adds more complexity. This, in turn, leads to an even larger number of possible configuration options. Therefore, a proper energy-consumption optimization results in large and complex analysis problems. As a consequence, the resulting analysis time substantially increases and may be practically infeasible. The necessity to prune the search space arises: To make the solving of such problems feasible, approaches are necessary that compress all necessary information of the state graphs describing the underlying system in a simpler problem.

Approach of WATWAOS to Search Spaces & Unpractical Analysis Times: In a nutshell, to solve the search-space problem, WATWAOS applies a two-fold approach: (1) *Node-merging techniques* are applied to the resulting graph by adding knowledge about possible device-state changes in the form of a hierarchical formalization. (2) Additionally, WATWAOS exploits advanced features of an ILP solver (i.e., callbacks and multi-scenario ILPs) to further cut down the overall time for finding worst-case–optimal solutions.

IV. The WatwaOS Approach

WatwaOS' objective is to generate tailored OS instances for real-time systems that automatically switch to a clock configuration that is optimal with respect to worst-case assumptions. We leverage techniques of existing approaches and add the additional goal of finding a clock-reconfiguration sequence that minimizes the overall energy consumption. To reduce analysis times, WatwaOS introduces a novel layered graph structure that reduces the number of states to be taken into account and makes use of advanced ILP-solver features. Subsequently, Section IV-A introduces the graph construction to analyze and minimize the WCRE of an OS instance. Sections IV-B and IV-C explain how this graph is transformed into an ILP and how WatwaOS makes use of solver features to speed up the optimization process. We share implementation details in Section IV-D before summarizing in Section IV-E.

A. Graph Construction

PABB Graph: A crucial part of finding a system's worst-case–optimal energy consumption is determining the worst-case energy consumption itself. For this, we adopt an approach similar to that used in existing research approaches [33], [40]. The starting point for WatwaOS' analysis is the control-flow graph of the application and operating system, which is available in the compiler backend. The nodes in this graph, known as basic blocks, represent sets of instructions executed sequentially. WatwaOS merges multiple of these basic blocks into a superstructure called *power-atomic basic blocks (PABBs)*. A PABB can contain multiple connected basic blocks during which the system's power consumption remains the same (since the set of active components is the same). The last instruction of a PABB is usually a system call, such as a clock-subsystem reconfiguration or scheduling operation. This aggregation of basic blocks into PABBs reduces the number of states that need to be considered later in the analysis without sacrificing precision in the solution. Keeping the number of possible states as small as possible is essential for achieving practical optimization times. The final PABB graph subsumes every executable task and interrupt, including the OS code.

Figure 3 (a) displays the PABB graph of an example application. It consists of a single task with four PABBs, which can be categorized into a compute-intensive phase (`filter`, `compute`) and an I/O-intensive phase (`send`). The first node $\mathcal{RP}$ represents a *reconfiguration point* inserted by WatwaOS, which will be discussed further below. The compute blocks of the task can additionally be interrupted by an asynchronous ISR (illustrated with the ϟ sign), which executes two PABBs.

Reconfiguration Points: After constructing the PABB graph, WatwaOS inserts reconfiguration points $\mathcal{RP}$ to reach the objective of energy minimization. To keep the problem analyzable, we adjust the clock configuration only at strategic points where reconfigurations have significant potential. Since I/O system calls are more energy-efficient at lower frequencies (see Section II), we cover these I/O sections with added clock-configuration decisions. WatwaOS adds reconfiguration points at three locations:

(1) Before & after I/O system calls: When a device is queried once, switching to a different clock configuration for that specific interaction can yield energy-optimal configurations.
(2) Before & after loops: Loops with repeated device operations occur, for example, when reading sensor values. Alternating between clock configurations for each iteration may decrease both time and energy efficiency because of switching overheads. Therefore, WatwaOS also takes clock reconfigurations for the entire loop into account.
(3) At beginning & end of a task: A task can execute multiple I/O operations, with short bursts of compute sections in between. Similar to the switching overheads in loops, frequent clock reconfigurations in tasks may result in an overall energy-consumption increase; this is why WatwaOS considers different clock configurations for the entire task.

In the example in Figure 3, WatwaOS added a single reconfiguration point $\mathcal{RP}$ at the start of the task (🏎). The example considers two clock configurations enabling the use of a low or a high clock frequency. This doubles the number of possible states, as the rest of the task can potentially execute at two frequencies (see device-state layer in part (b)). The novel idea of our approach is that each specific sequence of reconfiguration points will later be transformed into its own mathematical optimization problem representing the WCRE (i.e., the WCEC including all possible interferences) of the application. The sequence with the lowest WCRE then corresponds to the worst-case–optimal sequence of reconfiguration points.

State-Transition Graph with Power Information: The next step involves constructing the *power-state–transition graph (PSTG)* of the system, starting from an entry node in a predefined state (e.g., a low-frequency clock configuration). The PSTG consists of nodes that represent all possible states that may occur during system runtime, including transitions to higher-priority tasks or asynchronous ISRs. WatwaOS subdivides a state into three layers, whereby lower layers include the information of the upper layers. Besides the layered structure of multiple nodes in Figure 3, the layered structure of a single PSTG node is further detailed in Figure 4. Subsequently, we give further insights into the individual layers:

Layer (a): PABB Layer. This layer tracks the currently executed PABB of a node. Across the different PSTG layers, multiple nodes can execute the same PABB. Figure 3 shows a scenario where all PABBs except the first one can be executed either at a low (⎍⎍⎍) or a high (⎍⎍⎍) clock frequency.

Layer (b): Device-State Layer. The second layer additionally tracks the state of all system devices. In WatwaOS, a *device* refers to any power-consuming component, such as the CPU, UART controller, or Wi-Fi chip. Each device's state influences the system's overall power consumption. Additionally, some device properties may depend on the state of other devices. For instance, the time it takes to transmit a character via UART depends on the speed of the UART controller, which itself may be controlled by the CPU clock frequency. To account for these dependencies, WatwaOS allows definitions of custom costs per device system call.

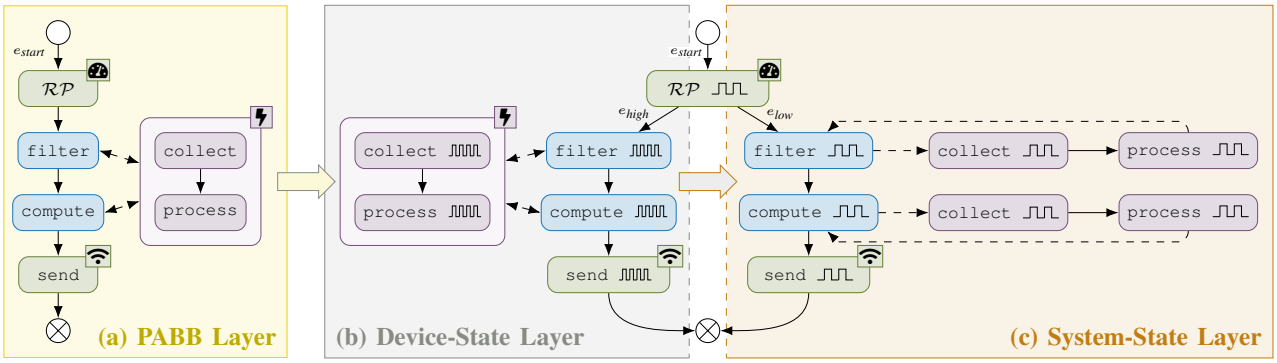

Fig. 3: WATWAOS' hierarchical abstraction structure for an example application with one task and one ISR. (a) The *PABB layer* is an abstraction from the application's control-flow graph. (b) The *device-state layer* extends (a) by adding the device context, such as the clock frequency. (c) The *system-state layer* is fully context-sensitive and extends (b) by scheduling information.

Layer (c): System-State Layer. The third layer additionally includes the context-sensitive (i.e., program-path–sensitive) state of all ISRs and tasks in the system. This information is essential for accurately determining all possible successors of a node. Specifically, the state of each task or ISR comprises both a *static* and a *context-sensitive* part. The *static* part remains unchanged during runtime and is statically configured in the system's description. It includes attributes such as the priority of a task and the minimum inter-arrival time of an interrupt. WATWAOS supports annotating such information directly in the source code using compiler `#pragmas`. The *context-sensitive* state information changes while the system is running and is captured by the PSTG construction's path analysis. With regard to ISRs, this state tracks whether respective interrupts are en-/disabled. Depending on the context-sensitive path constraints, transitions into the ISRs are present as successors. With regard to tasks, the context-sensitive state includes the current process state of a task and whether a task has been interrupted. This information is crucial for determining to which node WATWAOS returns when transitioning from a high-priority task (or ISR) τ_{high} back to a low-priority task τ_{low}: (1) If node n_1 in τ_{low} is interrupted by τ_{high}, then, after τ_{high} finishes, n_1 should be resumed. This creates a cycle in the graph. (2) If τ_{high} is instead started by node n_1 in τ_{low}, then we execute the *successor* of node n_1 in τ_{low} after τ_{high} finishes, which does not create a cycle in the graph. WATWAOS' PSTG construction algorithm starts at an entry node with a known state. The successor of this node is the first PABB of the initial OS task. WATWAOS knows about the system model and the semantics of all available system calls. Consequently, when a low-priority task τ_{low} activates a higher-priority task τ_{high}, then the first PABB of τ_{high} is visited next. When τ_{high} activates τ_{low}, then this information is stored in the state of each node in the PSTG up until the point where a scheduling decision has to be made and τ_{low} is scheduled.

Stateful power-atomic blocks at the device-state layer solve the problem of missing OS abstractions (see Problem #1 in Section III-A). These abstractions include devices and clock configurations and enable configuration switches at runtime.

Hierarchical Abstractions: The PSTG for the example introduced previously is shown in Figure 3 (c). Here, after the entry e_{start}, the $\mathcal{RP}$ node is a reconfiguration point at which either a low-frequency or a high-frequency clock is chosen. The edge e_{low} marks the start of a context with a low and e_{high} with a high clock speed. Thus, at the device-state layer (b), all successor PABBs exist twice. On the system-state layer (c), the number of interrupt nodes increases again (see `collect` and `process` nodes). This is because when an ISR is entered, WATWAOS needs to keep track of the interrupted node so that the correct node is resumed when leaving the ISR.

The presented example shows how the presence of interrupts can significantly inflate the number of states in the PSTG. As these states are later transformed into variables and constraints of an ILP, they directly impact the complexity of the mathematical problem and its solving time. Existing approaches use the most fine-granular version of the PSTG to determine the WCEC [28], [40]. In contrast, WATWAOS' novel understanding of a hierarchical PSTG structure opens up the possibility of notably reducing the number of considered states. This reduction is achieved by using nodes and edges of the device-state layer instead of the system-state layer whenever feasible. This is possible because, when calculating the WCEC, the information of the second layer, specifically the power consumption of each device, is sufficient. To ensure that only valid control-flow paths are considered in the ILP, additional measures are necessary, which will be described in Section IV-B. Compared to existing analysis techniques, WATWAOS effectively merges nodes and edges within the PSTG to minimize the number of states. We subsequently refer to this strategy as the *node-merging mechanism* of WATWAOS.

B. ILP Formulation

After enumerating all system states, WATWAOS transforms the PSTG into multiple ILPs, based on the context-sensitive constraints of the PSTG. Each ILP represents a specific sequence of clock reconfigurations and each ILP is formulated as a maximization problem for determining the WCRE. The PSTG's system-state layer comprises a set of nodes $\mathcal{N}$ and

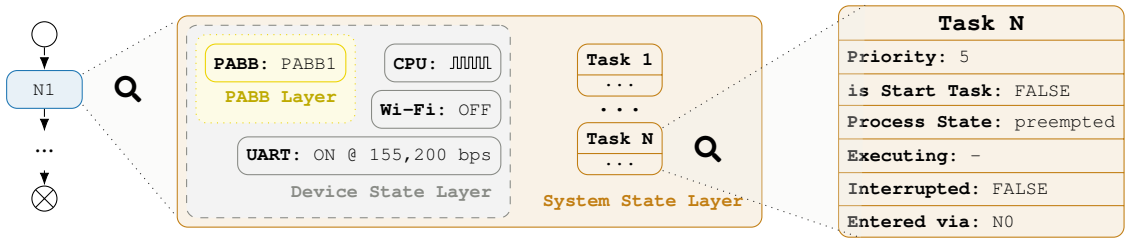

Fig. 4: Contents of a single state in the PSTG, which is subdivided into a PABB layer, a device-state layer, and a system-state layer. Lower layers include all information of their ancestor layers.

edges $\mathcal{E}$. Each node $n \in \mathcal{N}$ gets assigned a variable in the ILP that corresponds to its execution count $f(n)$. The same applies to each edge $e \in \mathcal{E}$ with $f(e)$. In the source code, edges are typically formed by control-flow structures such as if-else statements or loops. The ILP's objective is to maximize the $WCEC$ of all nodes and edges multiplied by their respective execution frequencies:

$$\max(\mathit{WCRE}) = \max\Big(\sum_{n\in\mathcal{N}} WCEC(n)\cdot f(n) + \sum_{e\in\mathcal{E}} WCEC(e)\cdot f(e)\Big)$$

The $WCEC(n)$ of a node n is calculated using its power consumption and its worst-case execution time as $WCEC(n) = P(n)\cdot WCET(n)$. $P(n)$ is derived from the maximum power consumption of all active devices in the PSTG's device-state layer. These values are configured in a configuration file for the target platform. $WCET(n)$ is determined for each node on the device-state layer using the existing worst-case analysis framework Platin [26]. The costs of clock reconfigurations are assigned to the edges (see e_{low} and e_{high} in Figure 3) since they result from hardware reconfiguration penalties (e.g., waiting for a clock to stabilize) rather than from the code itself.

WATWAOS solves multiple ILPs – one for each sequence of clock reconfigurations $\mathcal{RP}$. Among all solutions, the one with the lowest value represents a worst-case–optimal sequence of reconfigurations and, thus, the best energy consumption achievable under worst-case conditions. As a result, our approach consists of the following two interdependent optimization problems, as listed in the following Equation 1:

$$\mathit{WCRE}_{opt} = \underbrace{\min_{\mathcal{RP}} \Big(\underbrace{\max\ (\mathit{WCRE}_{\mathcal{RP}})}_{\textit{inner problem}}\Big)}_{\textit{outer problem}} \qquad (1)$$

The presented type of a hierarchical optimization structure is commonly referred to as a bilevel optimization problem. The *outer problem* identifies the minimal energy consumption across all ILPs, each of which represents an instance of the *inner problem*. The total number of ILPs depends on the number of possible clock configurations $\#CC$ and the number of reconfiguration points $\#\mathcal{RP}$, resulting in $\#CC^{\#\mathcal{RP}}$ scenarios. The example of two clock configurations and 20 $\mathcal{RP}$, resulting in over a million scenarios, underlines the rapid growth of the search space. By finding a solution for WCRE_{opt}, WATWAOS addresses the problem of lacking whole-system analysis and worst-case–optimal tailoring (see Problem #2 in Section III-B). Based on the optimal sequence of clock reconfigurations found by the ILP solver, WATWAOS generates a tailored OS instance that automatically switches its configuration at the determined points.

Basic Constraints: To map valid paths through the PSTG in the ILP, we add several different constraints comparable to the ILP-based IPET for control-flow graphs [31], [32]. The most important of these are structural constraints, which enforce that the count of all outgoing edges $\mathcal{E}_{out}(n)$ from a node n is equal to the count of its incoming edges $\mathcal{E}_{in}(n)$:

$$\forall n \in \mathcal{N} \quad \sum_{e\in\mathcal{E}_{in}(n)} f(e) = \sum_{e\in\mathcal{E}_{out}(n)} f(e)$$

In Figure 3, for example, the basic constraint $f(e_{start}) = f(e_{high}) + f(e_{low})$ holds. Additionally, the count of a node itself must generally be equal to the total sum of its incoming edge frequencies. We exclude the set of edges $\mathcal{E}_{ISR}(n)$ where the system returns to node n from an interrupt, as such edges merely *resume* a previously interrupted execution.

$$f(n) = \sum_{e_{in}\in\mathcal{E}_{in}(n)\setminus\mathcal{E}_{ISR}(n)} f(e_{in})$$

For our ILP, we define an artificial start node s with exactly one outgoing edge $\{s \to *\}$ with a count of 1. Likewise, exactly one of the edges to the end nodes $n \in \mathcal{N}_{end}$ is active:

$$\forall n \in \mathcal{N}_{end} \quad \sum_{e\in\mathcal{E}_{in}(n)} f(e) = 1$$

Control-Flow Loops: Loops pose a major challenge when defining the ILP as they introduce cycles leading to unbounded solutions. In WATWAOS, loops can thus be annotated with an upper bound in the source code, which is transformed into corresponding constraints in the ILP. A loop l consists of a set of nodes $\mathcal{N}_l \subset \mathcal{N}$, one of which is the loop header $n_{hdr} \in \mathcal{N}_l$. The loop has one or more entry edges $\mathcal{E}_{entry,l} = \{n \to n_{hdr} \,|\, n \in \mathcal{N}\setminus\mathcal{N}_l\}$, as well as backlink edges $\mathcal{E}_{backlink,l} = \{n \to n_{hdr} \,|\, n \in \mathcal{N}_l\}$. For each loop l, a constraint is added that limits the backlink count by the upper bound u_l with regard to the count of all entry edges:

$$\sum_{e_b\in\mathcal{E}_{backlink,l}} f(e_b) \le u_l \cdot \sum_{e_e\in\mathcal{E}_{entry,l}} f(e_e)$$

If no entry edge is activated, all backlink-edge frequencies must be 0, resulting in no loop activation. Otherwise, if the loop is activated, u_l bounds the backlink-edge frequencies.

Timing Constraints: WATWAOS allows specifying timing constraints for each task τ by annotating deadlines D_τ via

source code `#pragmas`. In the ILP, we add variables corresponding to each task's WCRT to detect deadline misses:

$$\begin{aligned} WCRT(\tau) \;=\; & \sum_{n\in\mathcal{N}_\tau} WCET(n)\cdot f(n) \\ +\; & \sum_{e\in\mathcal{E}_\tau} WCET(e)\cdot f(e) \end{aligned}$$

The sets $\mathcal{N}_\tau$ and $\mathcal{E}_\tau$ contain all nodes and edges where task τ is potentially runnable, that is, either preempted (e.g., by an interrupt) or actively running (see example of *Process state* in Figure 4). This way, WATWAOS accounts for the task itself and all potential interrupts and activations of higher-priority tasks. Our flow constraints ensure that the ILP solver only selects the nodes and edges belonging to the same (i.e., connected) execution paths of τ. Considering all nodes $\mathcal{N}$ and edges $\mathcal{E}$ yields the total execution time T, which is needed by other constraints to bound the number of interrupts:

$$T = \sum_{n\in\mathcal{N}} WCET(n)\cdot f(n) + \sum_{e\in\mathcal{E}} WCET(e)\cdot f(e)$$

The WCRT variables for each task serve as criteria to determine feasible solutions during ILP solving. Depending on the employed strategy, the solving time can be reduced by terminating early when deadline violations become apparent.

Node Merging: WATWAOS uses the nodes and edges of the device-state layer whenever possible. In the ILP, WATWAOS achieves this by replacing multiple counter variables of nodes of the system-state layer with one counter variable that effectively represents the node at the device-state layer. In the example of Figure 3 (c), the two low-frequency nodes `collect` and `process` at the system-state layer can be replaced with the respective single node of the device-state layer (Figure 3 (b)). By doing so, WATWAOS reduces the ILP's complexity by keeping the number of variables low. Additional constraints ensure that the ILP solver only considers valid control-flow paths. Without such constraints, in the example, the solver might enter the ISR via edge `filter` → `collect`, and incorrectly leave it via edge `process` → `compute`. To maintain accuracy in the ILP solution, WATWAOS adds the following constraint: The number of times a new task (ISR) is entered via a specific edge must be equal to the number of times the same task (ISR) is left. This is where the state information *Entered via:* from the system-state layer (see Figure 4) becomes essential. During state enumeration, WATWAOS tracks the node through which a new task or ISR is entered. When that task or ISR is left, a new constraint is added to the ILP to link the entry edge to the corresponding exit edge. This ensures that control-flow paths remain valid throughout the optimization process.

Interrupts: Interrupts require special handling as they introduce loops in the PSTG. Contrary to control-flow loops, no static bound on the occurrence of interrupts is known. To consider the actual worst-case bound during ILP solving, WATWAOS supports annotating the minimum inter-arrival time $t_{min,I}$ for interrupt I. Combined with the frequencies of its entry edges $\mathcal{E}_{entry,I} = \{\mathcal{N}\setminus\mathcal{N}_I \to \mathcal{N}_I\}$, where $\mathcal{N}_I$ denotes the set of nodes belonging to interrupt I, $t_{min,I}$ relates the occurrence of interrupts to the total execution time T. Release jitter for I is considered by adding one additional slot $t_{min,I}$:

$$t_{min,I}\cdot\sum_{e\in\mathcal{E}_{entry,I}} f(e) \le T + t_{min,I}$$

The occurrence of interrupts increases T, and with increasing T, in turn, more interrupt occurrences are possible. However, this circular dependency does not pose a problem for the ILP solver as long as a minimum inter-arrival time $t_{min,I}$ is given.

Another challenge with interrupts arises when the device state is modified inside an ISR (e.g., by changing to another clock configuration). In such cases, leaving the ISR will not resume the previously interrupted node n_{old} but, instead, return to a new node n_{new} with a different device state. The execution of only one of these two nodes should be counted to keep the ILP solution as small as possible. To ensure a valid upper bound for the WCRE, WATWAOS counts the node that results in higher overall energy consumption. This is accomplished by introducing a new variable $v_{potential}$, which is subtracted from $f(n_{old})$ and added to $f(n_{new})$. Using this variable, the solver has the flexibility to move the execution count of n_{old} to n_{new} if doing so leads to a higher energy consumption. This way of adjusting node frequencies ensures a sound bound for the ILP solution while keeping accuracy.

C. ILP-Solving Techniques

WATWAOS creates and solves multiple ILPs, one for each possible sequence of clock reconfigurations, using the solver Gurobi [18]. Each ILP maps the entire state-transition graph, including all reconfigurations. To allow for only one specific sequence of reconfigurations per ILP, we disable all other possible edges by setting their respective bound to 0. Gurobi offers several optimization techniques for efficiently solving multiple similar ILPs [21], [22], and we exploit five different optimization methods, which are presented below.

Method single-scenario*:* This is a naïve approach that solves all ILPs sequentially, one after another. Gurobi utilizes all available cores to solve an individual ILP.

Method single-scenario+callback*:* Gurobi provides the ability to invoke a callback function whenever a new temporary solution is found. This feature is referred to as *model-modification callbacks* [22]. WATWAOS leverages this callback to prematurely terminate the solving process of an ILP if the temporary solution is higher than any previously computed ILP solution. Early termination is possible since we are only interested in the minimum solution among all ILPs, that is, WATWAOS' *outer problem* (see Equation 1). The callbacks can also be used to terminate early whenever any of the specified task deadlines D_τ are violated.

Method single-scenario+multithreading*:* While Gurobi already parallelizes the ILP-solving process, it only does this for single ILPs. However, since a single ILP can be solved relatively quickly for most of our application scenarios (i.e., in the range of seconds), the scheduling overhead can have an impact on the overall solving time. In WATWAOS' respective implementation, we distribute the ILPs to all available cores in our system, whereas a single ILP is solved on only one core.

Method single-scenario+callback+multithreading*:* This method combines the two previous approaches. The callbacks keep track of the lowest WCRE solution of all threads.

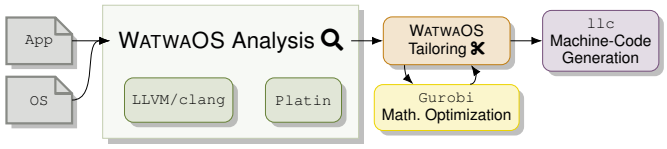

Fig. 5: WatwaOS workflow illustrating involved tools.

Method multi-scenario*:* Gurobi has a special multi-scenario ILP feature [21], which is particularly suited for WatwaOS' approach. When using the multi-scenario feature, a *base ILP* needs to be defined. Each following ILP *scenario* is allowed to modify the constraints of this base ILP. In our approach, the base ILP represents the entire PSTG. Then, for each scenario, we set the upper bound of each disabled clock configuration to 0. Gurobi can use data acquired in previous scenarios to speed up the solving process of the following ones, leading to a decreased overall solving time. Currently, Gurobi's callback functions cannot be used to terminate an individual scenario within a multi-scenario ILP; only the entire model can be terminated. Despite Gurobi's missing support, we will show significant analysis-time reductions with the multi-scenario approach in Section V-B2.

Reconsidering our problem statement, WatwaOS' node-merging technique and use of special solver features address the problem of large search spaces and unpractical analysis times (see Problem #3 in Section III-C). Node merging simplifies the ILPs by reducing their complexity, while leveraging Gurobi's features further accelerates the analysis process.

D. Implementation

WatwaOS leverages information provided by the LLVM compiler framework [41]. For example, it uses the control-flow graph available in LLVM to construct the needed graphs. Moreover, the ILP construction uses information about control-flow loops obtained from LLVM's *LoopAnalysis* pass. Consequently, most of WatwaOS' core functionality is implemented as a pass within the LLVM. Thereby, WatwaOS exploits sophisticated features from the LLVM infrastructure.

Figure 5 gives an overview of the tools involved in the creation of a tailored OS instance. There are two primary components to WatwaOS' implementation: *Analysis* 🔍 and *Tailoring* ✂. The analysis component is primarily integrated into the compiler. It constructs the PSTG and defines all variables and constraints required to formulate the mathematical optimization problem. To calculate the WCEC, WatwaOS needs to know the maximum power consumption of a node and its WCET. The maximum power consumption is configured per device state in a configuration file for the target platform. The (frequency-dependent) WCET, on the other hand, is determined for each block using the Platin analysis framework [26]. WatwaOS' tailoring component is invoked after the analysis phase and uses its output to construct multiple ILPs, which are then solved using Gurobi [18]. WatwaOS modifies the application bitcode by inserting clock reconfigurations at the identified worst-case–optimal points. The modified bitcode is then compiled into machine code using LLVM's `llc` tool.

E. Summary of Approach

The given system is transformed into a PSTG, which includes system states, device information, and reconfiguration options. WatwaOS finds the WCRE for all possible reconfiguration sequences and, therefore, obtains energy-optimal configurations. The use of path constraints encoded in ILPs leads to the fact that WatwaOS inherently finds worst-case–optimal energy-saving configurations. By merging the PSTG before solving, WatwaOS reduces the overall solution time.

V. Evaluation

Subsequently, we present WatwaOS' evaluation: After describing our measurement setup (Section V-A), we focus on the analysis and optimization time of example applications in Section V-B. Section V-C presents the energy reductions achievable in a real-world scenario, and Section V-D evaluates the impact on the schedulability of generated task sets. Finally, Section V-E compares WatwaOS to a DVFS approach.

A. Measurement Setup

WatwaOS' implementation has been evaluated on an ESP32-C3 [24]. It offers a single RISC-V core with a 4-stage in-order pipeline and 400 KiB of single-cycle access SRAM. Especially relevant for WatwaOS is the chip's extensively configurable clock subsystem. Unfortunately, the available documentation does not reveal details about the SoC's energy consumption at different clock configurations. Thus, we based our energy consumption model on worst-observed power measurements. To evaluate the power drawn by the ESP32-C3, we use a Joulescope JS220 energy-measurement unit [42]. It can measure both voltage and current for the device under test at two million samples per second and is thus able to provide precise values even for relatively short tests. The JS220 also supports four general-purpose inputs, one of which has been utilized to determine the start and end of a measurement.

We use a host system with an eight-core AMD Ryzen Pro 7 8840HS and 64 GiB RAM for WatwaOS' analysis and tailoring. For ILP solving, we use the Python API of Gurobi (version `v12.0.1`) [18] with a free academic license. The SoC and the energy-measurement unit are connected to the host system via USB, enabling automated test execution. In the following experiments, we support two clock configurations: a high-frequency configuration for compute-intensive tasks and a low-frequency configuration for I/O-intensive tasks. The high-frequency configuration uses a clock frequency of 160 MHz, which is the highest available option on the ESP32-C3. For the low-frequency configuration, we chose a clock frequency of 40 MHz, which we measured to be most energy-efficient. The source code of all presented experiments is available as part of WatwaOS' evaluation artifact [23].

B. Analysis & Optimization Time

1) Node Merging: WatwaOS applies node-merging techniques to reduce the number of variables and constraints in the PSTG's mathematical formalization. Using these techniques, we are able to merge similar control-flow paths while still

TABLE I: Effect of node merging on the optimization time of OS instances with different numbers of interrupts.

#ϟ	Merging	Var.	Constr.	Solving time (3000 scenarios)
0	Off	98	87	0.10 s
	On	98	87	0.10 s
1	Off	984	663	2.74 s
	On	270	238	2.29 s
2	Off	1854	1229	9.65 s
	On	430	386	5.26 s
3	Off	2724	1795	13.34 s
	On	590	534	6.56 s
4	Off	3594	2361	15.68 s
	On	750	682	9.24 s

obtaining the same solution as without node merging. In this experiment, we evaluate the effects of node merging on the ILP solving time on a system with a varying number of ISRs. All benchmarks comprise one low-priority task τ_{low} and up to four ISRs, each of which activates a high-priority task τ_{high}. Task τ_{low} is divided into a compute-intensive and an I/O-intensive phase, resulting in 13 interruptible PABBs. Task τ_{high} communicates with a connected device via the UART protocol by sending a fixed amount of characters. Table I compares the number of variables, constraints, and the ILP solving time for each benchmark with and without node merging for an increasing number of ISRs (#ϟ). WATWAOS was configured to insert one reconfiguration point $\mathcal{RP}$ before and after tasks and loops with I/O activity. For each benchmark, the solving time for up to the first 3000 ILP scenarios was measured.

Without interrupts, no difference exists in the number of variables and constraints between node merging en-/disabled. This is expected since WATWAOS only merges control-flow paths that exist multiple times in the PSTG. With a single interrupt, the node merging employed by WATWAOS already shows a significant reduction in the number of variables (-72.6 %) and constraints (-64.1 %), as well as a 5.2 % reduction in ILP solving time. WATWAOS achieves these reductions by merging similar states in the PSTG that execute the ISR or the high-priority task. When node merging is disabled, these states exist multiple times: For each interruptible PABB in τ_{low}, there exists a different set of PSTG states corresponding to the ISR and τ_{high}. Our approach consolidates such redundant states, thereby reducing the number of variables and constraints in the ILP and thus the solving time. As the number of interrupts increases to up to four in Table I, the benefit of node merging becomes even more pronounced, leading to a reduction of up to 50.8 %. These results underline the substantial advantages of node merging made possible by WATWAOS' hierarchical abstractions. In particular, real-world systems with interrupts experience a significant decrease in analysis time.

2) ILP-Solving Optimizations: To determine an optimal sequence of reconfigurations, WATWAOS solves multiple ILPs, each of which corresponds to the application's worst-case energy consumption for a particular clock-configuration sequence. In Figure 6, we compare the impact of WATWAOS' ILP optimization techniques (see Section IV-B) on the time to solve the first 3000 ILPs for three applications with varying complexity and, thus, different numbers of ILP variables. Each application comprises two tasks and one ISR with either I/O- or compute-intensive workloads.

TABLE II: ILP solving time for different amounts of tasks and ILP scenarios. Each benchmark also consists of one interrupt.

Tasks	ILP Scenarios 10	100	1000	10000
1	0.1 s	0.1 s	n.a.	n.a.
2	0.1 s	0.2 s	0.7 s	n.a.
3	0.2 s	0.5 s	0.7 s	14.8 s
4	0.4 s	0.9 s	3.3 s	25.3 s
5	1.2 s	2.3 s	6.6 s	54.7 s
6	2.6 s	5.2 s	14.7 s	126.0 s
7	7.9 s	14.2 s	37.7 s	331.8 s

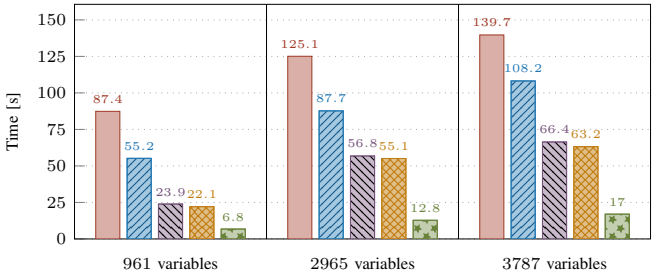

Fig. 6: Effects on the analysis time of different task sets (first 3000 scenarios, mean of 5 iterations) for five ILP-solving methods: single-scenario (▭), single-scenario + callback (▨), single-scenario + multithreading (▨), single-scenario + callback + multithreading (▨) and multi-scenario (▨).

As a baseline, shown as *single-scenario*, all ILP scenarios are solved sequentially after another. This method is the slowest for all tested PSTGs, since no information from previous solutions is leveraged. Introducing a *callback* function allows early termination of the solving process for single ILP and decreased the solving time by up to 37 % in our tests. Using *multithreading* to distribute the ILPs to all available cores enhances the solving performance, up to 70 % faster than *single-scenario*. Combining the callback with multithreading (*callback + multithreading*) further reduces the solving time. Finally, the *multi-scenario* option achieves the most significant solving-time reduction: Compared to *single-scenario*, using the multi-scenario feature reduces the solving time by up to 90 %. While exact solving times depend on the number of tasks and ISRs (determining the number of variables and constraints), *multi-scenario* provides the fastest solution in most observed cases and performs best overall. The results confirm that WATWAOS' use of advanced solver features drastically reduces the overall solving time.

3) Scalability: The duration of WATWAOS' tailoring step is primarily influenced by two factors: the complexity of the application (i.e., the number of tasks / interrupts) and the number of clock reconfiguration points. In this experiment, we compare the ILP solving time for different amounts of tasks and clock reconfiguration sequences, i.e., ILP scenarios.

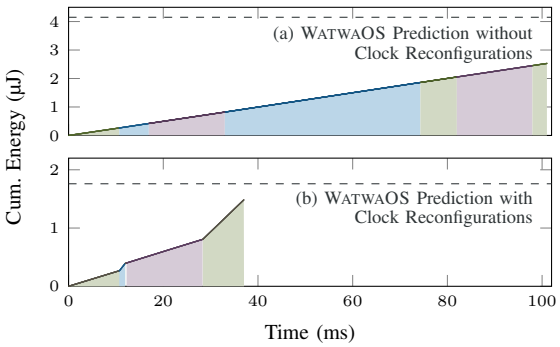

Fig. 7: Energy reduction of WATWAOS: Green areas mark I/O operations, blue areas computations, and purple areas ISR executions. The grey line marks the prediction of WATWAOS.

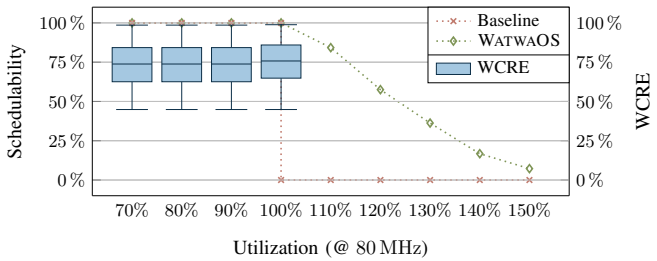

Fig. 8: Schedulability and energy consumption for 1000 task sets. Compared to a baseline system operating at the SoC's default clock fixed at 80 MHz, WATWAOS exploits time-energy tradeoffs by using both slower (energy-saving) and faster (energy-intensive) configurations.

Each benchmark includes one interrupt and up to seven tasks, covering both compute- and I/O-intensive operations.

Table II presents the total solving times for all benchmarks. As expected, the time needed to solve all ILP scenarios increases with both the system's complexity and the number of considered scenarios. Nevertheless, the results indicate that even for larger systems, WATWAOS achieves reasonable solving times in the range of minutes. Opting for coarse-grained clock reconfiguration points results in fewer ILP scenarios and, thus, less overall solving time. Choosing a finer granularity increases solving times but improves the chances of identifying a more energy-efficient clock-reconfiguration sequence.

In summary, WATWAOS provides reasonable solving times depending on the system's complexity and the number of ILP scenarios. By allowing developers to adjust the granularity of clock reconfigurations, we provide the flexibility to balance solving time against potential energy efficiency gains.

C. Energy-Reduction Analysis

In the next experiment, we evaluate the energy reductions achievable with WATWAOS for real-world applications. The test application consists of a single task that simulates a typical sense-compute-send scenario, comprising one compute-intensive and two I/O-intensive phases. Additionally, another device may interrupt the task, triggering an I/O-intensive ISR.

Figure 7 plots the cumulative energy consumption measured on the ESP32-C3 (a) without and (b) with the clock reconfigurations determined by WATWAOS. The green areas represent the I/O-intensive workloads, the blue areas describe the compute-intensive phase. ISR executions happen in the purple areas. In the upper figure, the entire application is executed at a low clock frequency. In contrast, the lower figure shows the system switching to a high clock frequency after the first I/O-intensive phase and back to a low frequency after the compute-intensive phase. Then, before returning from the ISR, the system again switches to a high-frequency clock configuration. This sequence of clock reconfigurations has been identified as worst-case optimal by WATWAOS. The compute-intensive phase takes significantly longer at a low frequency (a) than at a high frequency (b). Switching to a higher frequency for this compute-intensive phase accounts for most of the energy savings achieved by WATWAOS. Overall, the worst-observed energy consumption of the application was reduced by 41 % from 2.52 µJ down to 1.48 µJ.

Note that in the optimized scenario, the second I/O phase is executed at a high clock frequency. For this particular instance, it would have been more energy-efficient to switch to a low clock frequency after the ISR (purple area). However, this ISR may also occur during the compute-intensive phase, where switching to a lower frequency would drastically increase energy consumption. WATWAOS currently supports only non-contextual clock reconfigurations, meaning that it is not possible to switch to different clock configurations depending on the current state. Extending our approach to support contextual clock configurations is a topic of future research on WATWAOS.

In summary, WATWAOS is able to determine worst-case–optimal sequences of clock reconfigurations that result in a significant reduction in energy consumption compared to static clock configurations. As illustrated by a representative scenario for embedded systems, WATWAOS' combination of worst-case analysis (i.e., *inner problem* in Equation 1) and ILP-based tailoring (i.e., *outer problem* in Equation 1) substantially (i.e., 41 %) decreases the worst-observed energy demand.

D. Schedulability Evaluation

In this experiment, we evaluate the impact of WATWAOS on the schedulability and WCRE of 1000 randomly generated task sets consisting of 5–9 tasks with fixed priorities. We assume that device interactions occur in a *sense-compute-send* pattern. To reflect varying device usage patterns, we classify tasks as either I/O-intensive or compute-intensive, with 40 %–80 % of tasks in each set designated as I/O-intensive. The reconfiguration penalty for configuration switching is 1 ms. For each task set, the overall duration is $T = 25$ ms for the baseline at utilization of 100 %. This baseline assumes the SoC is using the PLL clock at 80 MHz [24]: It represents the center in the time-energy tradeoff and the SoC's default for energy-sensitive applications [43], [44]. The task utilizations are distributed using the UUniFast algorithm [45]. All task periods are chosen to be the same, which enables us to achieve total utilizations of 100 % for each baseline task set. For utilizations other

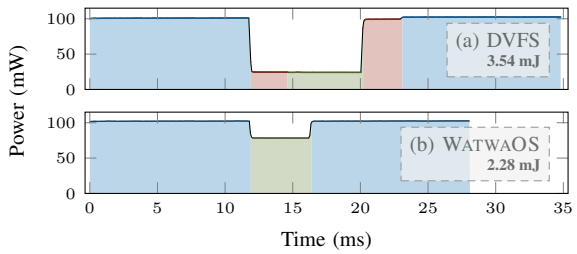

Fig. 9: Energy and Time Reduction of WATWAOS for a *compute→I/O→compute* transition in comparison to DVFS approaches. Green areas mark I/O operations, blue areas computations, and red areas reconfigurations.

than 100 %, the overall duration T is adjusted accordingly while the task parameters stay the same. To achieve variety in the sequences of I/O- and compute-insensitive tasks, this experiment uses randomly assigned priorities. In Figure 8, we compare the performance of WATWAOS against the baseline. In contrast to the 80 MHz-baseline, WATWAOS has awareness of all clocks and their speeds. Of particular relevance here is that clocks can run up to 160 MHz, allowing schedulability beyond the 100 % baseline. The baseline system can schedule all task sets up to 100 % utilization and none beyond (see drop of red line). WATWAOS can schedule more task sets, as it can execute compute-intensive tasks in a shorter amount of time at higher speeds. Notably, WATWAOS can still schedule over half of all task sets at 120 % utilization. For utilizations $\leq 100\,\%$, WATWAOS achieves considerably lower WCREs by switching to more energy-efficient configurations. While some task sets show minor potential for energy optimization (upper whiskers), others see reductions of up to 50 % (lower whiskers). Across all evaluated task sets, the median WCRE is around 75 % of the 80 MHz baseline. In summary, WATWAOS achieves reductions in both time and energy demands. Its awareness of reconfiguration penalties and goal of worst-case optimality ensures that the WCRE never increases.

E. Comparison to DVFS Approaches

Lastly, we compare WATWAOS to a DVFS approach. Figure 9 shows measurements on our hardware platform for a task set representing the common pattern of *compute→I/O→compute*. The upper part *(a)* shows the outcome for a DVFS approach comparable to ScaleClock [11], which uses high clock frequencies for compute-intensive tasks and low frequencies for I/O tasks. Changing the frequency requires clock reconfigurations before and after the I/O task, adding extra time and energy penalties. In comparison, the lower part *(b)* shows WATWAOS' optimal solution. Taking the penalties into account, WATWAOS executes the I/O task at the same frequency as the compute tasks. Here, WATWAOS determines to avoid reconfigurations and thus eliminates penalties. Thereby, WATWAOS reduces the energy consumption and also shortens the execution time. Consequently, our approach can meet task deadlines, while dynamic approaches that lack deadline awareness can violate timing constraints.

VI. Related Work

To our knowledge, WATWAOS advances the state of the art in energy-aware real-time systems with its contributions of (1) hierarchical abstractions for modeling clock subsystems, (2) reconfiguration-aware whole-system analyses, and (3) tighter integration with ILP-solving techniques.This section discusses our contributions based on the state of the art, specifically, for whole-system analysis and ILP solving.

Other approaches, such as Hoeller et al. [46], rely on hierarchical abstractions to optimize the energy consumption of systems. Compared to WATWAOS, existing approaches lack the ability to generate worst-case–optimal solutions. The ARA approach analyzes whole real-time systems without energy constraints [34], [35], [47]. ARA introduces abstraction layers to handle different OSs. Unlike WATWAOS, these layers cannot map energy-related application requirements to SoCs. Our objective is provable energy minimization while meeting real-time guarantees under worst-case assumptions. However, ARA's concepts for multi-core systems are an interesting avenue for WATWAOS.

In ILP-based optimization of multicast traffic [48], constraint reductions like our node-merging techniques reduced the solver's runtime. In contrast to WATWAOS, their problem is not bilevel. Luppold et al. [49] evaluated the performance of ILP solvers in the context of embedded systems. They show that choosing the proper tooling for ILPs can achieve substantial analysis-runtime reductions. In their experiments, Gurobi [18] outperformed CPLEX [50], without using specific solver features. For WATWAOS' target of worst-case optimality, we leverage such features: The use of model-modification callbacks and multi-scenario ILPs drastically reduces the analysis time. Beyond the work of Luppold et al. [49], we are unaware of related works on advanced ILP-solving techniques for worst-case analysis. We argue that further interdisciplinary research on embedded systems and mathematical optimization techniques is necessary to tackle the complexity of resource-constrained and -optimized systems.

VII. Conclusion

The complexity of modern SoCs, expressed by their multitude of clock-subsystem features, requires both comprehensive analysis techniques and operating-system support to enable resource-optimal execution. State-of-the-art operating systems lack abstractions to exploit these features, which hinders them from yielding worst-case–optimal solutions. Further, finding these solutions within acceptable analysis times is challenging.

We presented our WATWAOS approach, featuring both an analysis framework and a real-time operating system. WATWAOS exploits hierarchical abstractions to express energy-saving features. In essence, these abstractions represent fine-grained transitions within the analyzed system. The abstractions are designed to enable node-merging techniques that substantially reduce analysis times. Exploiting several features of ILP solvers further shortens analysis times. As our evaluations show, WATWAOS can reduce the solving time by up to 90 %.

Acknowledgements

We gratefully thank our anonymous shepherd and all reviewers for their valuable and constructive feedback. Further, we acknowledge Robert Burlacu wholeheartedly for sharing his knowledge about discrete optimization and solving bilevel problems. This work is funded by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) under the project number 502947440 (WA 5186/1-1, Watwa: Whole-System Optimality Analysis and Tailoring of Worst-Case–Constrained Applications).

References

[1] IoT Analytics. (2023) State of iot 2023: Number of connected iot devices growing 16%, to 16.0 billion globally. 2023. [Online]. Available: https://web.archive.org/web/20240709200248/https://iot-analytics.com/wp/wp-content/uploads/2023/05/Insights-Release-State-of-IoT-2023-Number-of-connected-IoT-devices-growing-16-to-16.0-billion-globally.pdf

[2] P. Sparks, "White paper: The economics of a trillion connected devices," 2017. [Online]. Available: https://web.archive.org/web/20241013015435/https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-1996-00-00-00-01-30-09/Arm-_2D00_-The-route-to-a-trillion-devices-_2D00_-June-2017.pdf

[3] H. Kim, B. Rigo, G. Wong, Y. J. Lee, and W.-H. Yeo, "Advances in wireless, batteryless, implantable electronics for real-time, continuous physiological monitoring," *Nano-Micro Letters*, vol. 16, no. 1, pp. 52:1–52:49, 2024.

[4] R. S. Al-Maroof, K. Alhumaid, A. Q. Alhamad, A. Aburayya, and S. Salloum, "User acceptance of smart watch for medical purposes: An empirical study," *Future Internet*, vol. 13, no. 5, pp. 127:1–127:29, 2021.

[5] A. K. Yetisen, J. L. Martinez-Hurtado, B. Ünal, A. Khademhosseini, and H. Butt, "Wearables in medicine," *Advanced Materials*, vol. 30, no. 33, pp. 1 706 910:1–1 706 910:26, 2018.

[6] S. Ahmed, B. Islam, K. S. Yildirim, M. Zimmerling, P. Pawełczak, M. H. Alizai, B. Lucia, L. Mottola, J. Sorber, and J. Hester, "The internet of batteryless things," *Communications of the ACM (CACM)*, vol. 67, no. 3, pp. 64–73, 2024.

[7] J. Hester, K. Storer, and J. Sorber, "Timely execution on intermittently powered batteryless sensors," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17)*, 2017, pp. 1–13.

[8] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo, "Energy-aware scheduling for real-time systems: A survey," *ACM Transactions on Embedded Computing Systems (ACM TECS)*, vol. 15, no. 1, pp. 7:1–7:34, 2016.

[9] A. Maioli, K. A. Quinones, S. Ahmed, M. H. Alizai, and L. Mottola, "Dynamic voltage and frequency scaling for intermittent computing," *ACM Transactions on Sensor Networks (ACM TOSN)*, vol. 21, no. 2, pp. 1–34, 2025.

[10] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an os for the internet of things," in *Proceedings of the 32nd IEEE International Conference on Computer Communications (INFOCOM '13)*, 2013, pp. 79–80.

[11] M. Rottleuthner, T. C. Schmidt, and M. Wahlisch, "Dynamic clock reconfiguration for the constrained iot and its application to energy-efficient networking," in *Proceedings of the International Conference On Embedded Wireless Systems And Networks (EWSN '22)*, 2023, pp. 168–179.

[12] H. Chiang, H. Ayers, D. Giffin, A. Levy, and P. Levis, "Power Clocks: Dynamic multi-clock management for embedded systems," in *Proceedings of the International Conference on Embedded Wireless Systems and Networks (EWSN '21)*, 2021, pp. 139–150.

[13] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, S. Leonard, P. Pannuto, P. Dutta, and P. Levis, "The Tock embedded operating system," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17)*, 2017, pp. 45:1–45:2.

[14] E. Dengler and P. Wägemann, "Crêpe: Clock-reconfiguration–aware preemption control in real-time systems with devices," in *Proceedings of the 36th Euromicro Conference on Real-Time Systems (ECRTS '24)*, 2024, pp. 10:1–10:25.

[15] E. Dengler, P. Raffeck, S. Schuster, and P. Wägemann, "FusionClock: Energy-optimal clock-tree reconfigurations for energy-constrained real-time systems," in *Proceedings of the 35th Euromicro Conference on Real-Time Systems (ECRTS '23)*, 2023, pp. 6:1–6:24.

[16] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem – overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (ACM TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[17] R. Jayaseelan, T. Mitra, and X. Li, "Estimating the worst-case energy consumption of embedded software," in *Proceedings of the 12th Real-Time and Embedded Technology and Applications Symposium (RTAS '06)*, 2006, pp. 81–90.

[18] Gurobi Optimization, LLC. (2025) Gurobi optimizer reference manual. [Online]. Available: https://docs.gurobi.com/projects/optimizer/en/current/index.html

[19] R. E. Bixby. (2015) Computational progress in linear and mixed integer programming. [Online]. Available: https://web.archive.org/web/20240702202149/https://people.bath.ac.uk/masjhd/Others/Bixby2015a.pdf

[20] Gurobi Optimization, LLC. (2019) Gurobi 8 performance benchmarks. [Online]. Available: https://web.archive.org/web/20250613005259/https://assets.gurobi.com/pdfs/benchmarks.pdf

[21] ——. (2024) Multiple scenarios. [Online]. Available: https://web.archive.org/web/20241209001548/https://docs.gurobi.com/projects/optimizer/en/current/features/multiscenario.html

[22] ——. (2025) Callbacks. [Online]. Available: https://web.archive.org/web/20250120110927/https://docs.gurobi.com/projects/optimizer/en/current/reference/python/callback.html

[23] T. Häberlein, E. Dengler, P. Raffeck, and P. Wägemann. WatwaOS source code. 2025. [Online]. Available: https://gitos.rrze.fau.de/i4/openaccess/watwaos

[24] Espressif Systems Co., Ltd., *ESP32-C3 Technical Reference Manual*, 2022, version 1.3. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-c3_technical_reference_manual_en.pdf

[25] STMicroelectronics. (2025) Ultra-low-power with FPU Arm Cortex-M4 MCU 80 MHz with 1 Mbyte of Flash memory, LCD, USB OTG, DFSDM. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32l476rg.pdf

[26] E. J. Maroun, E. Dengler, C. Dietrich, S. Hepp, H. Herzog, B. Huber, J. Knoop, D. Wiltsche-Prokesch, P. Puschner, P. Raffeck, M. Schoeberl, S. Schuster, and P. Wägemann, "The platin multi-target worst-case analysis tool," in *Proceedings of the 22nd International Workshop on Worst-Case Execution Time Analysis (WCET '24)*, 2024, pp. 2:1–2:14.

[27] J. Morse, S. Kerrison, and K. Eder, "On the limitations of analyzing worst-case dynamic energy of processing," *ACM Transactions on Embedded Computing Systems (ACM TECS)*, vol. 17, no. 3, pp. 59:1–59:22, 2018.

[28] P. Raffeck, J. Maier, and P. Wägemann, "WoCA: Avoiding intermittent execution in embedded systems by worst-case analyses with device states," in *Proceedings of the 25th ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '24)*, 2024, pp. 83–94.

[29] S. Wegener, K. K. Nikov, J. Nunez-Yanez, and K. Eder, "EnergyAnalyzer: Using static wcet analysis techniques to estimate the energy consumption of embedded applications," in *Proceedings of the 21th International Workshop on Worst-Case Execution Time Analysis (WCET '23)*, 2023, pp. 9:1–9:14.

[30] S. Hahn, J. Reineke, and R. Wilhelm, "Towards compositionality in execution time analysis: Definition and challenges," *ACM SIGBED Review*, vol. 12, no. 1, pp. 28–36, 2015.

[31] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proceedings of the 32nd Design Automation Conference (DAC '95)*, 1995, pp. 456–461.

[32] P. Puschner and A. Schedl, "Computing maximum task execution times: A graph-based approach," *Real-Time Systems*, vol. 13, pp. 67–91, 1997.

[33] C. Dietrich, P. Wägemann, P. Ulbrich, and D. Lohmann, "SysWCET: Whole-system response-time analysis for fixed-priority real-time systems," in *Proceedings of the 23rd Real-Time and Embedded Technology and Applications Symposium (RTAS '17)*, 2017, pp. 37–48.

[34] G. Entrup, B. Fiedler, and D. Lohmann, "MultiSSE: Static syscall elision and specialization for event-triggered multi-core rtos," in *Proceedings*

of the 29th Real-Time and Embedded Technology and Applications Symposium (RTAS '23), 2023, pp. 262–275.

[35] G. Entrup, A. Kässens, B. Fiedler, and D. Lohmann, "Applied static analysis and specialization of cross-core syscalls for multi-core autosar os," *Real-Time Systems*, vol. 60, no. 3, pp. 491–533, 2024.

[36] A. Miné, "Static analysis of embedded real-time concurrent software with dynamic priorities," *Electronic Notes in Theoretical Computer Science*, vol. 331, pp. 3–39, 2017.

[37] Zephyr Community. (2025) Zephyr Project. 2025. [Online]. Available: https://www.zephyrproject.org/

[38] M. Fischer, "Reworking the zephyr clock control subsystem," 2023, talk at Embedded Open Source Summit (EOSS 2023). [Online]. Available: https://static.sched.com/hosted_files/eoss2023/b4/Reworking%20the%20Zephyr%20Clock%20System.pdf

[39] Zephyr Community. (2025) [RFC] Proposal for a Precision Clock Subsystem Architecture. 2025. [Online]. Available: https://web.archive.org/web/20250214074722/https://github.com/zephyrproject-rtos/zephyr/issues/76335

[40] P. Wägemann, C. Dietrich, T. Distler, P. Ulbrich, and W. Schröder-Preikschat, "Whole-system worst-case energy-consumption analysis for energy-constrained real-time systems," in *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS '18)*, 2018, pp. 24:1–24:25.

[41] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)*, 2004, pp. 75–86.

[42] Jetperch LLC, *Joulescope JS220 User's Guide Precision DC Energy Analyzer*, 2022, revision 1.3. [Online]. Available: https://download.joulescope.com/products/JS220/JS220-K000/users_guide/

[43] Espressif Systems Co., Ltd. (2025) Esp32-c3 – light sleep example. [Online]. Available: https://github.com/espressif/esp-idf/blob/4e036983a751e4667ade94c8f6f6bf1e7f78eff0/examples/system/light_sleep/sdkconfig.defaults#L1

[44] RIOT Operating System. (2025) Riot os default clock configuration. [Online]. Available: https://github.com/RIOT-OS/RIOT/blob/5136d2379e0327fb9864a090be3a4535373d9d49/cpu/esp32/include/sdkconfig_esp32c3.h#L56

[45] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, pp. 129–154, 2005.

[46] A. S. Hoeller, L. F. Wanner, and A. A. Fröhlich, "A hierarchical approach for power management on mobile embedded systems," in *Proceedings of the IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES '06)*, 2006, pp. 265–274.

[47] G. Entrup, J. Neugebauer, and D. Lohmann, "RTOS-independent interaction analysis in ARA," in *Proceedings of the 15th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '22)*, 2022, pp. 7–13.

[48] E. Schweissguth, D. Timmermann, H. Parzyjegla, P. Danielis, and G. Mühl, "Ilp-based routing and scheduling of multicast realtime traffic in time-sensitive networks," in *Proceedings of the 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '20)*, 2020, pp. 1–11.

[49] A. Luppold, D. Oehlert, and H. Falk, "Evaluating the performance of solvers for integer-linear programming," Technische Universität Hamburg, Tech. Rep., 2018.

[50] IBM. (2025) ILOG CPLEX Optimization Studio. [Online]. Available: https://www.ibm.com/products/ilog-cplex-optimization-studio