

CHRONOS: Compensating Hardware Related Overheads with Native Multi Timer Support for Real-Time Operating Systems

1st Kay Heider

TU Dortmund University
Dortmund, Germany

kay.heider@tu-dortmund.de

2nd Christian Hakert

TU Dortmund University
Dortmund, Germany

christian.hakert@tu-dortmund.de

3rd Kuan-Hsun Chen

University of Twente
Enschede, the Netherlands

k.h.chen@utwente.nl

4th Jian-Jia Chen

TU Dortmund University
Dortmund, Germany

jian-jia.chen@cs.tu-dortmund.de

Abstract—The management of timing constraints in a real-time operating system (RTOS) is usually realized through a global tick counter. This counter acts as the foundational time unit for all tasks in the systems. In order to establish a connection between a tick and an amount of elapsed time in the real world, often this tick counter is periodically incremented by a hardware timer. At a fixed interval, this timer generates an interrupt that increments the counter. In an RTOS, jobs can only become ready upon a timer tick. That means, during a tick interrupt, the tick counter will be incremented, jobs will be released, and potentially, a scheduling decision will be conducted to select a new job to be run. As this process naturally uses some processing time, it is beneficial regarding the system utilization to minimize the time spent in tick interrupts. In modern microcontrollers, multiple hardware timers are often available. To utilize multiple timers to reduce the overhead caused by tick interrupts, multiple methods are introduced in this paper. Generally, the task dispatching process is distributed over multiple timers, where each timer manages a subset of the task set. The number of interrupts that are triggered by these timers can then be reduced by mapping tasks to timers in such a manner that the greatest common divisor (GCD) of all task periods in a subset is maximized, and the GCD is adopted as the interrupt interval of the timer. To find an optimal mapping of tasks to timers, an MIQCP-model is presented that minimizes the overall number of tick interrupts that occur in a system, while ensuring a correct task release behavior. The presented methods are implemented in FreeRTOS and evaluated on an embedded system. The evaluation of the methods show, that compared to the baseline implementation in FreeRTOS that uses a single timer with a fixed period, the presented methods can provide a significant reduction in overhead of up to $\approx 10\times$ in peak and up to $\approx 6\times$ in average.

Index Terms—real-time operating system, task handling, timer, job release, miqcp

I. INTRODUCTION

In a real-time operating system (RTOS), the notion of time is usually abstracted by a global system tick counter. Often, this tick counter is periodically incremented by an underlying hardware timer that counts at a fixed frequency. This system timer establishes the connection between the tick counter and

This work has received funding from the DFG Priority Program “Disruptive Memory Technologies” (SPP 2377) as part of the project “ARTS-NVM” (502308721). It was further supported by the DFG Project “One-Memory” (405422836).

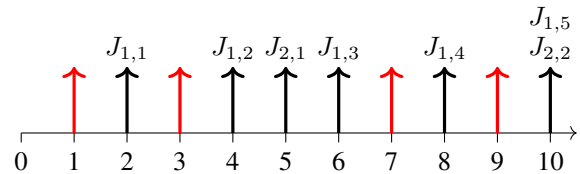


Fig. 1. Example of tick interrupts with a single timer and the release times of jobs from two tasks τ_1 and τ_2 . An upward arrow indicates a tick interrupt.

the elapsed wall time. Usually, the process of handling the timer tick is realized by an interrupt.

In addition to keeping the time in an RTOS, the tick counter is used to schedule and release jobs. A job can only ever become ready upon a timer tick, and depending on the scheduling policy, that newly released job can be selected to be executed after the tick handling. To correctly release jobs in a system with strictly periodic tasks, the tick interrupt needs to occur at a period, which is a common divisor of all task periods. Otherwise, a job would miss the time it should become ready. As a result, there are potentially tick interrupts where no job becomes ready, and only the tick counter is incremented. For example, in a system where one task releases jobs every two time units and another task releases jobs every five time units, the system timer needs to tick at every time unit as both task periods are co-prime. Suppose that both tasks release their first job at time zero. Then, the system execution is only affected by the interrupt that dispatches new jobs that at times 2, 4, 5, 6, 8, 10, ... and so forth. The tick interrupt, however, occurs at every time unit. In Figure 1, the tick interrupts and release times of the jobs from these two tasks are depicted for a system with a single timer. During the ten time units depicted, at four out of ten interrupts, no jobs become ready. The process of incrementing the system tick counter, potentially releasing new jobs, and performing a scheduling decision, naturally causes an overhead for each handling of a timer tick. To reduce this overhead, it is desirable to reduce the number of tick interrupts where no job becomes active, while maintaining a correct job release behavior.

Modern microprocessors often possess multiple hardware

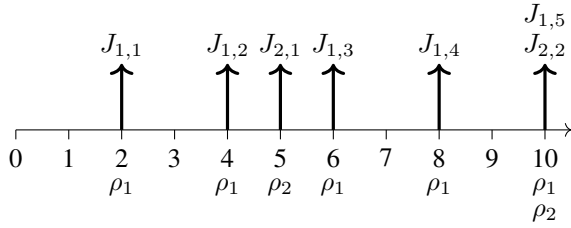


Fig. 2. Example of tick interrupts with two timers ρ_1, ρ_2 and the release times of jobs from two tasks τ_1, τ_2 . An upward arrow indicates a tick interrupt.

timers that can be configured to independently generate interrupts at different intervals. For example, the ESP32 series of microprocessors contain four independent hardware timers [17]. An automotive microprocessor like the Infineon TC1796, that is, for example, used by the SLOTH on time, contains 256 hardware timers [1]. Utilizing multiple timers to manage a set of periodic real-time tasks can reduce the number of tick interrupts and thus reduce the overall overhead caused by tick interrupts. This can be achieved by partitioning the task set into subsets where the greatest common divisor (GCD) of the task periods is maximized. A separate hardware timer can then be assigned to manage only that partition of the task set, and thus, the tick interrupt interval can be set to the GCD of the task periods. Considering the example from before, a single timer must tick at every time unit to ensure the correct dispatching of jobs. For a system with at least two hardware timers, one timer could manage the task with period two, and another timer could manage the task with period five. Thus, the tick interrupt intervals could also be set to two and five time units, respectively. As a result, tick interrupts only occur when jobs are released, minimizing the overall number of tick interrupts.

In Figure 2, the tick interrupts and release times of the two jobs are depicted for a system with two timers ρ_1, ρ_2 . During the depicted ten time units, a job becomes ready at every tick interrupt. Since the timers are configured such that they trigger an interrupt only at the time when a job will be released, there are no interrupts that cause unnecessary overhead. It should be noted that at time ten, two interrupts are triggered, one from each timer.

In this paper, we examine strategies to reduce the task handling overhead incurred by housekeeping operations of an RTOS for strictly periodic task sets. As shown by a recent empirical industry study, in 82% of the investigated systems, task activations are periodic [2]. Towards optimizing the task handling process, we present an MIQCP that is used to partition a given task set and configure separate hardware timers to handle those partitions to minimize the overall number of tick interrupts. Moreover, we examine different strategies to manage the set of delayed tasks which has an impact on the efficiency of delaying tasks and releasing jobs.

II. RELATED WORK

Many methods to improve the system efficiency of real-time operating systems (RTOSes) have been proposed before. Hofer et al. propose a method where, for every task, multiple hardware timer cells are used, among other things, to control the activation of the task and to monitor the deadline of a task [1]. Their approach achieves a very low task dispatching latency, resulting in an improvement of up to $171\times$. To improve the efficiency of managing timers in a system, Varghese and Lauck discuss several implementation schemes and propose the use of a timing wheel which effectively realizes a bucket sorting technique for systems where there is enough memory available [3]. Their approach can perform management operations of timers in constant time. Another approach to realize efficient timers is given by Aron and Druschel [4]. They propose soft timers which are triggered at certain times during the execution of the system where the overhead of invoking a timer handler is very low. In order to more efficiently realize typical operations in an RTOS, such as task scheduling and time management, Kohout et al. propose a hardware module that realizes these operations [5]. Their approach decreases the processing time for these RTOS operations by up to 90%. Hagens and Chen investigate the efficiency of using different data structures as a basis for the task dispatcher in FreeRTOS, an open-source RTOS that will also be used in this paper [6]. They evaluate the usage of Lists, Binary Search Trees, Red-Black Trees, and Heaps. Moreover, Balas and Benini also consider FreeRTOS, but on a RISC-V-based microcontroller [7]. They utilize specific instructions of the RISC-V instruction set to improve the handling of interrupts and to reduce the time needed for context switches.

III. SYSTEM MODEL

We consider an RTOS with a periodic timer tick interrupt that invokes a *system tick handler*. This interrupt handler is responsible for central housekeeping operations of the RTOS, namely incrementing the global tick counter, releasing new jobs, and conducting a scheduling decision whenever a new job is released. Therefore, new jobs are solely released during the system tick interrupt. Generally, the system tick handler can be invoked by a timer that generates interrupts at a fixed interval, i.e., every P time units, the interrupt will occur, or by a *one-shot* timer whose alarm interval is reconfigured after every occurrence. In this paper, we consider RTOSes where the system tick is backed by a timer interrupt with a fixed interrupt interval during runtime.

Moreover, we consider task sets that are comprised of strictly periodic tasks that release their first jobs synchronously at time 0. Hence, any job is always released at time points that are integer multiples of their corresponding task's period. We denote $\mathbb{T} = \{\tau_1, \dots, \tau_n\}$ as the task set of n tasks that should be executed on the system. A periodic task is denoted by $\tau_i = (C_i, T_i, D_i)$ with C_i as the worst-case execution time (WCET), T_i the period of the task, and D_i the relative deadline. Every task releases infinitely many jobs, the set of jobs released by task τ_i are denoted as \mathbb{J}_i , and the k -th released

job of τ_i is denoted as $J_{i,k}$. Since every task releases their first job at time 0, the release time of the k -th job of task i is given by $k \cdot T_i$. Additionally, we assume that the task set, along with the number of timers and the mapping of tasks to timers, are given. The number of timers and the mapping are also assumed to be fixed during runtime.

Regarding the hardware side, we assume a CPU with multiple hardware timers, whose interrupt intervals can be independently chosen. The interrupt intervals are assumed to be configurable as an integer multiple of a base time interval that is specified by the hardware. After a timer has triggered an interrupt, the internal counter of the hardware timer should be reset to zero. Alternatively, a timer could trigger interrupts whenever the internal counter has reached an integer multiple of a given time interval. The set of configurable timers is denoted as $\mathbb{P} = \{\rho_1, \dots, \rho_m\}$, with timer ρ_j triggering an interrupt every P_j time units.

IV. MULTI-TIMER TASK DISPATCHING

Generally, for a synchronous task set that contains only periodic tasks, the release times of the tasks are always at tick counts that are multiples of the task periods. To ensure a correct task release behavior, the tick interrupt needs to occur at an interval that is a divisor of all task periods. Consequently, if any two task periods are co-prime, the tick interrupt needs to occur at every time unit. However, if the GCD of all task periods is greater than one, it is possible to trigger an interrupt at an interval given by the GCD of the task periods. The ticks that are required to ensure a correct task release behavior, i.e., ticks where tasks are released, are called *required* ticks. A tick where no task can be released is called a *not-required* tick. It is desirable, regarding the improvement of the system efficiency, that the number of *not-required* ticks is reduced, as every interrupt causes a certain overhead. The notion of *required* and *not-required* ticks can be explained by Figure 1 using the example from the introduction. Here, a red upward arrow indicates a *not-required* tick and a black upward arrow indicates a *required* tick. As the GCD of the task periods is one, the timer period P_j must also be set to one. In this scenario, the ticks 1, 3, 7, and 9 are *not-required* as, there, no task becomes ready.

When a system can harness multiple hardware timers, it is possible to distribute the task handling process over multiple timers. Instead of using a single timer to manage the releases of all tasks, every hardware timer can manage a subset of tasks where the GCD of the task periods is potentially greater. The GCD in that subset can then be adopted as the tick interrupt interval of the associated hardware timer. The idea is shown in Figure 3 and described in more detail in the following:

For a task set \mathbb{T} that contains n periodic tasks, there are at most n different task periods. When employing multiple hardware timers ρ_1, \dots, ρ_m to distribute the task handling process over, every timer ρ_j can be assigned a subset $\mathbb{T}_j \subseteq \mathbb{T}$. In the extreme case, every hardware timer could be assigned a subset in which the task period is uniform, i.e., the system would need at most n hardware timers such that no tick interrupts

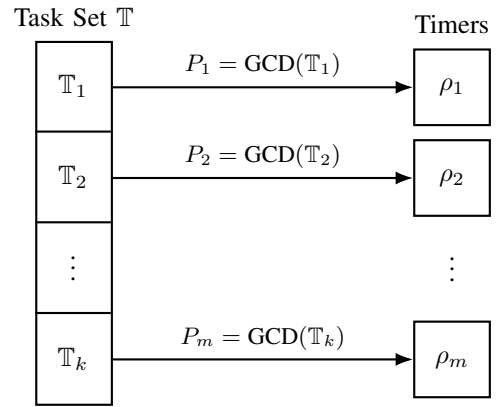


Fig. 3. Mapping of k partitions of a task set \mathbb{T} to m timers

occur for *not-required* ticks. However, this is unrealistic, as in most systems, there are significantly fewer hardware timers than tasks with different periods. A more sensible approach is to partition the task set into k subsets $\mathbb{T}_1, \dots, \mathbb{T}_k$ that cover multiple tasks. A timer ρ_j is then configured to maintain the tasks of the set \mathbb{T}_j , for $1 \leq j \leq k$. Intuitively, to reduce the number of tick interrupts, the task set should be partitioned such that the GCD of the task periods of each partition is maximized. However, more attention needs to be paid to the number of timers employed and their configured periods:

Suppose that the system contains m hardware timers ρ_1, \dots, ρ_m that trigger periodic interrupts at P_1, \dots, P_m time units. The expected number of tick interrupts that occur for a fixed time interval in the multi-timer system can be calculated as:

$$\sum_{j=1}^m \frac{1}{P_j} \quad (1)$$

As a result, the overall number of tick interrupts is generally reduced when the timer periods are increased. However, special attention has to be given to situations where it is not always beneficial to utilize multiple timers even though they tick at longer intervals than it would be possible with just a single timer. For example, suppose that the task set consists of three tasks with periods $T_1 = 2$, $T_2 = 3$, and $T_3 = 5$. In the system with a single timer, a tick interrupt needs to occur at every time unit as the task periods are co-prime, while for a system with three hardware timers ρ_1, ρ_2, ρ_3 their interrupt intervals could be set to $P_1 = 2$, $P_2 = 3$, and $P_3 = 5$ time units. The system with multiple timers then experiences $\frac{1}{2} + \frac{1}{3} + \frac{1}{5} > 1$ tick interrupts under the observation from Equation (1), i.e., more interrupts will be triggered compared to the single timer system, and thus more overhead is caused.

A further observation is that there is naturally no benefit in having multiple timers whose interrupt intervals are integer multiples of each other. Specifically, if the GCD in any subset was equal to one, the associated timer would need to interrupt at every time unit and there is no benefit to employ any additional timers.

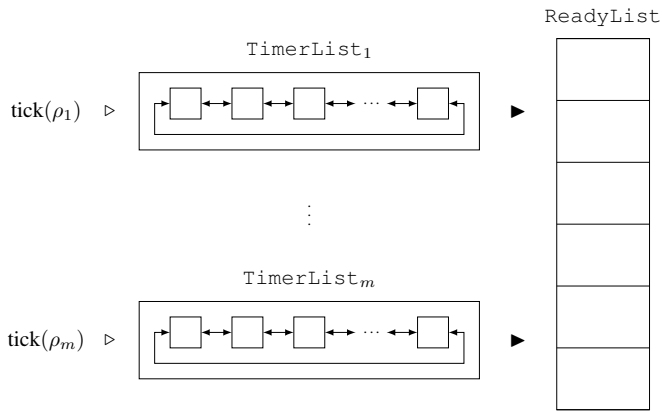


Fig. 4. Dispatcher architecture with multiple timers

Naturally, it is not always possible to find a partition that assigns multiple tasks to each timer where overall the number of tick interrupts are reduced. Nonetheless, if the system contains a sufficient number of hardware timers for a task set, the overall overhead can potentially be reduced. A method to find an optimal partition of the task set that minimizes the number of tick interrupts is presented in Section VI.

V. CHRONOS

In order to exploit the observations made in Section IV, the task handling process has to be distributed over multiple timers. We present *Chronos*, a method that uses multiple timers with different interrupt intervals to release jobs from a given partition of a task set. The general idea is to create multiple tick counters and multiple lists to store delayed tasks that are each managed by a separate hardware timer. Each timer then maintains the task release process for a partition of the task set. We assume that the set of delayed tasks is kept sorted.

a) Architecture: *Chronos* naturally extends the concept of a single timer that manages a global tick and a global list for delayed tasks to multiple timer sources. Given a system that contains m hardware timers, the mapping of tasks to timers, that is supplied as an input, induces a partition of the task set into at most m subsets. Each subset \mathbb{T}_j contains the tasks that shall be managed by timer ρ_j . The period P_j of timer ρ_j is configured to the GCD of the periods of all tasks in \mathbb{T}_j . If all available m timers are used, m tick counters and m lists for delayed tasks will be kept. Each timer ρ_j independently manages a tick counter $\text{tick}(\rho_j)$ and a list TimerList_j that is used to store delayed tasks. A schematic for the architecture can be found in Figure 4.

Instead of a global tick count that is used by all tasks as a time reference, an independent tick counter is created for each hardware timer. Each of these counters is incremented whenever the associated timer triggers an interrupt. To still have a unified length for a tick over all timers, the tick counters are incremented by the period of the timer. It is assumed that all timers are configured such that the basic time unit for all timers is the same, e.g., all timers generate interrupts at multiples of 1 ms such that the time unit of task and timer

Algorithm 1: *Chronos* Interrupt Routine

Input: Timer number j

Require: $P_j > 0$, TimerList_j sorted ascending by tasks' nextRelease

```

1  $\text{tick}(\rho_j) += P_j$ 
2 if  $\text{tick}(\rho_j) \geq \text{timerNextRelease}_j$  then
3   loop
4     if  $\text{TimerList}_j$  is empty then
5        $\text{timerNextRelease}_j =$ 
6          $\text{MAX\_VALUE}$ 
7       break
8     end
9      $\tau \leftarrow \text{head}(\text{TimerList}_j)$ 
10    if  $\tau.\text{nextRelease} > \text{tick}(\rho_j)$  then
11       $\text{timerNextRelease}_j =$ 
12         $\tau.\text{nextRelease}$ 
13      break
14    end
15    remove  $\tau$  from  $\text{TimerList}_j$ 
16    add  $\tau$  to  $\text{ReadyList}$ 
17  end
18 end

```

periods is also given in ms. Tasks that are assigned to a specific timer then use the timer's tick counter as a reference.

Alongside a tick counter $\text{tick}(\rho_j)$, a list TimerList_j is created for each hardware timer ρ_j . This TimerList_j is used to store delayed tasks that are assigned to the timer ρ_j . We assume that the tasks are inserted into the TimerList_j sorted by their next release time in ascending order. Compared to a system with one global timer, the number of tasks that are managed by a single timer in *Chronos* is usually lower as the tasks are distributed over multiple timers. This is beneficial in terms of the overhead for inserting tasks.

Furthermore, a function $\text{delay}()$ is introduced delays a periodic task in a multi-timer setup. This function computes the next time when a job of the task should be released and inserts the delayed task to the TimerList_j of the associated timer ρ_j in a sorted manner. The list is sorted by the next release times in ascending order. Also, a $\text{timerNextRelease}_j$ value is kept for each timer ρ_j that is always set to the next tick count when a task assigned to ρ_j should become ready.

b) Interrupt Routine: As the number of timers that will be used by *Chronos* is variable, the interrupt routine for the hardware timers is not specifically designed for each timer but bases its functionality on the timer that executes it. Algorithm 1 shows pseudocode of the interrupt routine.

The *Chronos* tick interrupt routine is executed by all hardware timers and takes as input an identifier of the timer that triggers the input, i.e., if timer ρ_j triggers the interrupt, the index j will be used to determine which tick counter and which TimerList_j has to be updated. Generally, it is assumed that the timer period P_j is greater than zero and that the associated

TimerList_j is sorted by the task's next release time in ascending order.

First, the tick counter $\text{tick}(\rho_j)$ is incremented by the timers period P_j . After this, it is checked whether a task can become ready by comparing the updated tick counter $\text{tick}(\rho_j)$ with the $\text{timerNextRelease}_j$ value. If it is not time to release a task yet, the interrupt exits early. Conversely, if the tick count has reached the time when a task can be released, the associated TimerList_j is iterated. In the case that the TimerList_j becomes empty because all tasks were released, the $\text{timerNextRelease}_j$ is set to a maximal value, such that if multiple interrupts are triggered without any task being added to the list in between, the interrupt routine can always exit early. Otherwise, the task that is at the head of the TimerList_j will be inspected. If the next release time of the inspected task τ has not been reached yet, the $\text{timerNextRelease}_j$ is set to the next release time of that task, and the routine exits. If the task τ should be released, it is removed from the TimerList_j and added to the global ReadyList . This is done until a task is encountered that cannot be released yet or the TimerList_j is empty.

A. Chronos-const

As *Chronos* aims to reduce the number of tick interrupts, it can be explored whether it is beneficial to reduce the overhead of inserting tasks into a TimerList_j in a sorted manner at the cost of increasing the workload in each tick interrupt. The method called *Chronos-const* realizes a constant cost for inserting tasks into a TimerList_j with the trade-off that a tick interrupt now always causes a linear overhead regarding the number of tasks contained in the TimerList_j .

Generally, *Chronos-const* uses the same architecture as *Chronos*; however, the way each TimerList_j is managed changes. To achieve a constant cost for inserting tasks into a TimerList_j , tasks will now be appended to the list. That means there is no order that can be exploited in the tick interrupt. To determine which tasks should be released, the whole TimerList_j has to be inspected during an interrupt of timer ρ_j . For every task, its next release time will be examined, and the task will be released accordingly. This causes a linear overhead regarding the number of delayed tasks contained in the TimerList_j at the point when the interrupt is triggered. The tick interrupt routine is stated in pseudocode in Algorithm 2.

Like the *Chronos* tick interrupt routine, the *Chronos-const* tick interrupt routine is executed by all hardware timers and determines which timer structures should be updated on the basis of the supplied timer number. Here, it is also assumed that timer periods are strictly greater than zero, but the TimerList_j does not need to be sorted.

The first step in the interrupt routine is to increment the tick counter $\text{tick}(\rho_j)$ by the timer's period P_j . If the tick count has not reached the time when a task can be released, the routine exits early. Otherwise, initially, the $\text{timerNextRelease}_j$ will be set to a maximal value. Over the course of the interrupt, the $\text{timerNextRelease}_j$ will always be updated if a task

Algorithm 2: Chronos-const Interrupt Routine

Input: Timer number j

Require: $P_j > 0$

```

1 tick( $\rho_j$ ) +=  $P_j$ 
2 if tick( $\rho_j$ )  $\geq$  timerNextRelease $_j$  then
3   timerNextRelease $_j$  = MAX_VALUE
4    $\tau \leftarrow \text{head}(\text{TimerList}_j)$ 
5   while  $\tau \neq \text{tail}(\text{TimerList}_j)$  do
6     if  $\tau.\text{nextRelease} > \text{tick}(\rho_j)$  then
7       if  $\tau.\text{nextRelease} <$ 
8         timerNextRelease $_j$  then
9         timerNextRelease $_j$  =
10           $\tau.\text{nextRelease}$ 
11       end
12        $\tau \leftarrow \text{next}(\text{TimerList}_j)$ 
13     else
14        $\tau' \leftarrow \text{next}(\text{TimerList}_j)$ 
15       remove  $\tau$  from  $\text{TimerList}_j$ 
16       add  $\tau$  to  $\text{ReadyList}$ 
17        $\tau \leftarrow \tau'$ 
18     end
19   end
20 end
```

is encountered whose next release time is lower than the current $\text{timerNextRelease}_j$. That means, at the end of the routine, the $\text{timerNextRelease}_j$ will be set to the minimal next release time of any contained task. The TimerList_j will always be iterated completely, i.e., the routine exists only after the tail of the TimerList_j is reached. The tail of a list is assumed to be defined to be a special item that marks the end of the list. If the currently examined task cannot be released, the $\text{timerNextRelease}_j$ will be adjusted accordingly, and the iteration continues with the next task. However, if the task can be released, first, the task's successor in the list will be stored in τ' . This is because if τ is removed from the list, the current position in the list would be lost. Afterwards, τ is removed from the TimerList_j and inserted into the global ReadyList . Finally, the iteration continues by setting τ to τ' , restoring the position in the list and effectively advancing by one item.

B. Chronos-harmonic

The first two presented methods are applicable to any task set comprised of only periodic tasks and can result in a reduced number of tick interrupts if a suitable partition is found. For harmonic task sets, additional optimizations can be made. *Chronos-harmonic* exploits the fact that the order in which tasks from a harmonic task set are released always stays the same. In fact, the release pattern is given by the periods of the tasks. Generally, *Chronos-harmonic* can be applied to harmonic task sets where all tasks are periodic and released at time zero. Additionally, *Chronos-harmonic* can be applied to task sets that are not harmonic as a whole, but where a

harmonic base exists that contains at most as many elements as there are available hardware timers. Then, the tasks with periods of each harmonic chain can be managed by a separate hardware timer. With *Chronos-harmonic*, the cost of delaying a task is constant without increasing the overhead of the tick interrupt.

Since this method is specifically designed for harmonic task sets, it should be discussed whether harmonic task sets are in fact encountered in real systems. For example, in many avionic systems, the task periods are harmonic, as described by Easwaran et al. [8]. Harmonic task sets are also considered for applications in the field of robotics [9], while in automotive systems or submarine systems, many tasks also have harmonic periods [10], [11]. Therefore, much research also focuses on harmonic task sets [12]–[14].

a) *Architecture*: The architecture of *Chronos-harmonic* is still similar to *Chronos* in terms of the multi-timer capabilities, as it distributes the task dispatching process over multiple hardware timers by introducing local structures for every timer. That means, for each timer ρ_j , a `TimerListj` and a tick counter `tick(ρ_j)` is kept. However, here, the `TimerListj` is implemented as an array of fixed-size that contains all tasks a timer manages, sorted after their period in ascending order. In more detail, for a timer ρ_j and the associated harmonic task set $\mathbb{T}_j = \{\tau_1, \dots, \tau_k\} \subseteq \mathbb{T}$ with $1 \leq k \leq |\mathbb{T}|$, the `TimerListj` is an array of k elements that stores the tasks τ_1, \dots, τ_k sorted ascending by their periods T_1, \dots, T_k . As the task set is given and no tasks are allowed to be created during runtime, the task set \mathbb{T}_j can be sorted offline before the scheduler is started. Since \mathbb{T}_j is harmonic, for any two task $\tau_1, \tau_2 \in \mathbb{T}_j$ with $T_1 \geq T_2$, T_1 is an integer multiple of T_2 . Consequently, whenever T_1 is released, T_2 and any other task in \mathbb{T}_j with a smaller period than T_1 will also be released. In *Chronos-harmonic*, the period P_j of the associated timer ρ_j is configured to the smallest period of any task in \mathbb{T}_j , i.e., $P_j = \min\{T_i \mid \tau_i \in \mathbb{T}_j\}$. Therefore, every time ρ_j triggers an interrupt, at least one task will be released, i.e., the task with period P_j . Also, if `tick(ρ_j) mod $T_i = 0$` , for any task period $T_i \in \{T_i \mid 1 \leq i \leq k\}$, all tasks in \mathbb{T}_j with periods that are smaller than T_i have to be released. That means the order in which tasks are released is always fixed and determined by the tasks' periods. As a result, there is no need to dynamically insert and sort the `TimerListj`.

b) *Interrupt Routine*: In *Chronos-harmonic*, this observation is exploited by the fixed-sized array that contains the tasks in combination with an array that contains the task periods in the same order. That means, for a task set with n tasks, the index i , with $1 \leq i \leq n$, selects the task τ_i in the `TimerListj`, and the tasks period T_i in the `TaskPeriodsj` array. Generally, whenever a task τ_i is released, it will be removed from the `TimerListj` by setting the field `TimerListj[i]` to NULL. This marks that the task τ_i is currently ready and can be used to detect a potential deadline miss if the task is not in the `TimerListj` when an interrupt is triggered where τ_i should be released.

Now, in every interrupt for timer ρ_j , the `TaskPeriodsj`

array is iterated from the start until a task period T_i is encountered where `tick(ρ_j) mod $T_i \neq 0$` or the end of the array has been reached. As the periods are sorted in ascending order, once a task has been encountered that cannot be released, further tasks will not be released either. When a task $\tau_i \in \mathbb{T}_j$ should be released, τ_i has to be in the `TimerListj`, i.e., the entry `TimerListj[i]` cannot be NULL. If the entry is NULL, the task is skipped, and the next task is examined. When a task τ_i is released, the entry `TimerListj[i]` is set to NULL, and τ_i is inserted into the global ready queue. The described tick interrupt routine is given as pseudocode in Algorithm 3.

Algorithm 3: *Chronos-harmonic* Interrupt Routine

Input: Timer number j

Require: $P_j > 0$, `TimerListj` sorted ascending by task periods

```

1 tick( $\rho_j$ ) +=  $P_j$ 
2 for  $i \leftarrow 1$  to  $|\mathbb{T}_j|$  do
3   if tick( $\rho_j$ ) mod  $T_i \neq 0$  then
4     break
5   end
6   if TimerListj[i] = NULL then
7     continue
8   end
9    $\tau_i \leftarrow$  TimerListj[i]
10  TimerListj[i] = NULL
11  add  $\tau_i$  to ReadyList
12 end
```

VI. MAPPING TASKS TO TIMERS

In order to utilize the methods described in Section V, a mapping of tasks to timers is needed. This mapping is critical to the efficiency of the methods, as it directly influences the number of tick interrupts that occur. For this reason, it is desirable to find an optimal mapping such that the overall number of tick interrupts is minimized.

A. Problem Definition

As described in Section IV, the number of tick interrupts that occur for not-required ticks should be minimized. This works by mapping each task to a timer, inducing the partition of task set into m subsets, for m timers. Concretely, every task in the same subset of the partition $\Gamma = \{\mathbb{T}_1, \dots, \mathbb{T}_m\}$ of the task set $\mathbb{T} = \{\tau_1, \dots, \tau_n\}$ will be assigned to the same timer ρ_j for $1 \leq j \leq m$. The goal is then to minimize the expected number of tick interrupts. This objective can be written as:

$$\min \sum_{j=1}^m \frac{1}{P_j} \quad (2)$$

for m timers with timer periods P_1, \dots, P_m . It should be noted that the system could possess more than m timers, as it is not always optimal to utilize all available timers. For a

timer ρ_j to correctly release the tasks of its associated task set \mathbb{T}_j , the timer period P_j must be an integer divisor of all task periods in $\mathcal{T}_j = \{T_i \mid \tau_i \in \mathbb{T}_j\}$. It is assumed, without loss of generality, that any two tasks $\tau_1, \tau_2 \in \mathbb{T}$ have different task periods T_1, T_2 , since tasks with the same period can be assigned to the same timer ρ_j without impacting P_j . In order to minimize $\frac{1}{P_j}$ for any timer ρ_j , the timer period P_j has to be maximized, i.e., P_j should be set to the GCD of \mathcal{T}_j .

B. Minimizing the Number of Tick Interrupts

The problem described in Section VI-A can intuitively be described as an MIQCP model.

Let $\mathbb{T} = \{\tau_1, \dots, \tau_n\}$ be set of tasks with periods $\mathcal{T} = \{T_i \mid 1 \leq i \leq n\}$. Additionally, let $m \in \mathbb{N}_{\geq 1}$ be the number of timers that are available in the system. Then, the problem can be formulated as the following MIQCP model:

$$\min \sum_{j=1}^m f_j \cdot u_j \quad (\text{VI.1})$$

$$\text{s.t. } f_j \cdot P_j = 1 \quad \text{for } j = 1, \dots, m \quad (\text{VI.2})$$

$$\sum_{j=1}^m m_{i,j} = 1 \quad \text{for } i = 1, \dots, n \quad (\text{VI.3})$$

$$d_i \cdot \sum_{j=1}^m P_j \cdot m_{i,j} = T_i \quad \text{for } i = 1, \dots, n \quad (\text{VI.4})$$

$$\bigvee_{i=1}^n m_{i,j} = u_j \quad \text{for } j = 1, \dots, m \quad (\text{VI.5})$$

$$d_i \in \mathbb{Z}, 1 \leq d_i \leq T_i \quad \text{for } i = 1, \dots, n \quad (\text{VI.6})$$

$$P_j \in \mathbb{Z}, 1 \leq P_j \leq \max \mathcal{T} \quad \text{for } j = 1, \dots, m \quad (\text{VI.7})$$

$$f_j \in \mathbb{Q}, \frac{1}{\max \mathcal{T}} \leq f_j \leq 1 \quad \text{for } j = 1, \dots, m \quad (\text{VI.8})$$

$$m_{i,j} \in \{0, 1\}, u_j \in \{0, 1\} \quad \text{for } i = 1, \dots, n, j = 1, \dots, m \quad (\text{VI.9})$$

It should be noted that the task periods T_1, \dots, T_n and the maximal task period $\max \mathcal{T}$ are constants in the model. Furthermore, an inequality written as $x^L \leq x \leq x^U$ is a shorthand for two separate constraints $x^L \leq x$ and $x \leq x^U$. The model introduces the following variables:

- For every timer ρ_j with $1 \leq j \leq m$, P_j denotes the period of the timer ρ_j and f_j denotes the expected number of interrupts that the timer causes. The timer period P_j is an integer and can assume values between one and the maximal task period of \mathbb{T} . This is because if a timer is configured, it must have an interrupt interval of at least one time unit. Also, an interrupt interval is naturally bounded by the greatest task period. In the extreme case, the timer period can be equal to the maximal task period. These bounds are stated in Equation (VI.7).

As the expected number of interrupts caused by a timer ρ_j is calculated as $\frac{1}{P_j}$, the range of f_j is directly given by the bounds of P_j . The bounds of f_j are defined in Equation (VI.8). It should be noted that because f_j is effectively calculated as $\frac{1}{P_j}$, f_j is rational.

Additionally, a binary variable u_j is introduced for each timer ρ_j , which indicates whether ρ_j should be used in the system. If $u_j = 1$, timer ρ_j will be used.

- The binary variables $m_{i,j}$ are defined for every task τ_1, \dots, τ_n and for every timer ρ_1, \dots, ρ_m . A value of $m_{i,j} = 1$ indicates that the task τ_i is assigned to timer ρ_j .
- For every task $\tau_i \in \mathbb{T}$, an integer variable d_i is introduced. If a task τ_i is assigned to timer ρ_j , the timer period P_j must be an integer divisor of the task period T_i , i.e., a number $d_i \in \mathbb{Z}$ must exist such that $T_i = d_i \cdot P_j$ holds. The bounds of d_i can directly be derived from the bounds of P_j . Each variable d_i has a lower bound of one and an upper bound of T_i . This is stated in Equation (VI.6).

The presented MIQCP model provides a realization of the problem in the following manner:

The objective function given in Equation (VI.1) minimizes the total expected number of tick interrupts. Equation (VI.1) realizes the remark about Equation (2) that not necessarily every timer should be used by integrating the binary variables u_j into the objective function. Conveniently, this means that for $u_j = 0$, the number of interrupts caused by timer ρ_j is also zero, i.e., the timer ρ_j will not be used.

The constraint given by Equation (VI.2) effectively assigns f_j the number of interrupts caused by timer ρ_j , as $f_j \cdot P_j = 1 \Leftrightarrow f_j = \frac{1}{P_j}$. Because Equation (VI.2) is a quadratic equality constraint, the model becomes non-convex [15].

Next, any task τ_i must be assigned to exactly one timer ρ_j . Equation (VI.3) ensures that this requirement is satisfied by enforcing that the sum of all variables $m_{i,j}$ for a particular task τ_i is one. Thus, exactly one $m_{i,j} \in \{m_{i,j} \mid 1 \leq j \leq m\}$ is allowed to be set to one.

The critical part of this model is the constraint that for each task τ_i that is assigned to a timer ρ_j , the timer period P_j must be an integer divisor of the task period T_i . This condition is given in Equation (VI.4) for every task $\tau_i \in \mathbb{T}$. Naturally, P_j only needs to be a divisor of T_i if τ_i is assigned to ρ_j . As the assignment of a task τ_i to a timer ρ_j is indicated by the binary variable $m_{i,j}$ and Equation (VI.3) states that a task is always assigned to exactly one timer, the timer period that has to be a divisor of T_i is selected by $\sum_{j=1}^m P_j \cdot m_{i,j}$. The result of the sum is the period of the timer that task τ_i is assigned to. If an integer d_i exists that satisfies the equality constraint, P_j is an integer divisor of T_j . Here, it should be noted that multilinear terms like $x \cdot y \cdot z$ are often not allowed. However, they can be rewritten by introducing auxiliary variables. Hence, for the readability of the model, the term is left in its trilinear form.

Finally, the constraint given in Equation (VI.5) ensures that any binary variable u_j is set to one if any task is assigned to timer ρ_j , i.e., if any $m_{i,j} = 1$ for all tasks $\tau_i \in \mathbb{T}$.

Notably, the presented non-convex MIQCP model can be transformed into an ILP model by linearizing the quadratic constraints and objective function and reformulating the objective to remove the rational variables. That is possible because all variables have a defined upper and lower bound.

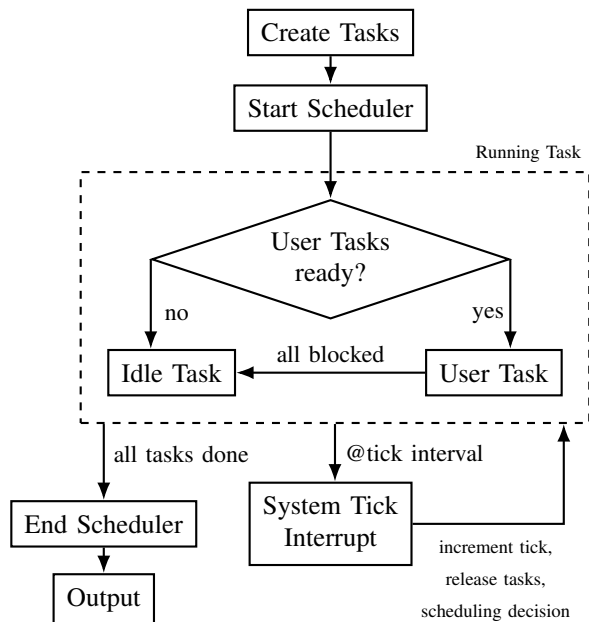


Fig. 5. Structure of the FreeRTOS program

VII. EVALUATION

We evaluate the methods presented in Section IV on a real-world embedded system that contains four hardware timers. In order to evaluate the efficiency of the methods, the cumulative time spent in tick interrupts and the cumulative time that is spent on delaying tasks will be measured for every method. The time spent on the insertion of delayed tasks is also recorded to evaluate the efficiency of *Chronos-const* and *Chronos-harmonic* as both methods aim to reduce the overhead caused by inserting the tasks in a sorted manner. These measurements will then be compared to the baseline implementation in FreeRTOS [16], which uses a single timer.

a) Evaluation Setup: The embedded system that was used to perform the measurements is an ESP32-S3-DevKitC that contains an ESP32-S3 chip [17] with additionally 8 MB of externally connected PSRAM. The ESP32-S3 is a microcontroller that contains two XTensa LX7 CPU cores that can be configured to run at up to 240 MHz. Internally, the chip can access 512 kB of SRAM. Additionally, the chip contains four hardware timers that can be configured to trigger interrupts at periodic intervals independently of each other. In the following, the four hardware timers will be described as ρ_1, ρ_2, ρ_3 , and ρ_4 and their periods will be described as P_1, P_2, P_3 , and P_4 .

The methods presented in Section IV were implemented in the ESP32 FreeRTOS port using the uncore version of FreeRTOS 10.5.1 included in the ESP-IDF 5.2.1 development framework. Every test was performed with enabled preemption and time slicing. The included `GPTimer` library was used to configure the hardware timers.

For all tests, the CPU frequency was set to 240 MHz without dynamic frequency scaling. For the baseline implementation

in FreeRTOS, the system timer period was always set to 1 ms. The additional hardware timers were configured such that one tick in the multi-timer setup also corresponds to 1 ms. For this, the frequency at which the internal counters of the hardware timers operate was configured to 1 MHz. Then, for a timer ρ_j with a period of P_j ticks, the timer was configured to generate an interrupt when the internal counter reaches a value of $P_j \cdot 1000$, i.e., an interrupt is generated after $P_j \cdot 1$ ms. When an interrupt is triggered, the counter is always reset to zero and starts anew.

b) Structure of the FreeRTOS program: In the following, the structure of the program that is used to evaluate the methods is outlined. A visualization of the structure of the program can be found in Figure 5. Generally, in FreeRTOS, it is allowed to create tasks after the scheduler was started, i.e., the task set can grow dynamically. However, as the task set needs to be known in advance to create the mapping of tasks to timers, the program takes a predefined task set as an input, and all tasks are created before the FreeRTOS scheduler is started. After the scheduler is started, the execution of the created tasks starts. Every task is configured with a fixed workload and a fixed number of releases. When all tasks have finished their workload, the scheduler is ended and the collected information, such as the cumulative time spent in tick interrupts, total runtime and number of deadline misses, is produced as an output.

c) Measurement Methodology: In order to evaluate the overhead caused by the different methods, the overall time spent in tick interrupts has to be measured. For this, probe points are added at the start and the end of any interrupt to record the time spent in that interrupt.

To differentiate tick interrupts from other interrupts, such as yield calls that are also implemented via interrupts in the ESP32 port of FreeRTOS, the source of the interrupt is inspected before any measurements are taken. As the ESP32-S3 is based on the XTensa architecture [18], there are 32 interrupt sources that can be allocated to different interrupt handlers. Every interrupt is identified in the system by a number between 0 and 31, which is assigned to that interrupt when it is allocated. When an interrupt occurs, the `INTERRUPT` register can be read to check which interrupt was triggered.

The `INTERRUPT` register is a 32-bit register where the i -th bit represents the interrupt that was allocated the number i . That means, if bit k of the `INTERRUPT` register is one when an interrupt occurs, the interrupt source that was assigned the number k triggered the interrupt. In order to only measure the time spent in tick interrupts, the mapping of interrupt sources to bits of the `INTERRUPT` register was fixed. Then, at the start of each interrupt, it is checked whether the bit that represents a tick interrupt source is set to one. If that is the case, a timestamp in the form of the current clock cycle is recorded. For this, the `CCOUNT` register can be read. At the end of an interrupt, another timestamp is recorded, if a timestamp was also recorded at the start of the interrupt. Additionally, to measure the time that is spent on delaying tasks, probe points are added to the `xTaskDelayUntil()`

and `delay()` methods.

d) Test Configuration: To evaluate the different methods, several test scenarios were formed. First, *Chronos* and *Chronos-const* were evaluated against the baseline FreeRTOS implementation using a non-harmonic task set. For this, two different workload scenarios were evaluated where the amount of work that each task has to perform differs. In the *low* single-job workload scenario, each task has to perform 1000 additions before it will be delayed again. For the *high* single-job workload scenario, each task performs 10000 additions during its execution. This was done to explore how the overhead of the tick interrupts changes when tasks are more likely not to be delayed already when the tick interrupt occurs next. A higher single-job workload means that each task needs more time to execute and thus spends more time in the `ReadyList`. Thus, the number of tasks that need to be processed during a tick interrupt tends to be less, resulting in a shorter interrupt duration.

Generally, each task is configured to be released five times before it is considered to be finished. After five releases, the task will be blocked immediately after it has been released. Furthermore, the methods are also evaluated for harmonic task sets, such that *Chronos-harmonic* can also be applied. Here, the same low and high single-job workloads are employed, albeit for tasks with different periods, as the task set is harmonic.

Additionally, a scenario called *harmonic single* is evaluated in which the baseline implementation is compared to a single timer setup using the techniques from *Chronos-const* and *Chronos-harmonic* with a fixed timer period of 1 ms. This scenario compares the efficiency of the different methods without reducing the number of interrupts. However, in this scenario the number of additions each task has to perform was reduced to 100, such that no deadline misses occur.

Since the ESP32-S3-DevKitC is an embedded system with only moderate amounts of memory and computational power, it cannot support very large task sets without experiencing deadline misses. For this reason, both the non-harmonic and harmonic task sets contain 100 tasks. For the multi-timer methods, the MIQCP model from Section VI was used to distribute the tasks to the four timers and to configure the periods of the timers. The result is a partition $\Gamma = \{\mathbb{T}_1, \mathbb{T}_2, \mathbb{T}_3, \mathbb{T}_4\}$ of the task set \mathbb{T} where timer ρ_j is assigned the tasks of the set \mathbb{T}_j , for $1 \leq j \leq 4$. For both the non-harmonic and the harmonic task sets, the tasks were generated with periods that are multiples of 3 ms, 5 ms, 7 ms, or 11 ms. Therefore, the four timers were also configured to generate interrupts at 3 ms, 5 ms, 7 ms, or 11 ms. For the *harmonic single* test, only a single timer with a period of 1 ms was used. In the non-harmonic task set, for every task a base period $b \in \{3 \text{ ms}, 5 \text{ ms}, 7 \text{ ms}, 11 \text{ ms}\}$ was chosen and then multiplied with a random factor $r \in \{x \in \mathbb{N} \mid 1 \leq x \leq 10\}$, such that for a timer ρ_j , all its associated tasks have a period of $T_i = P_j \cdot r$, for all $\tau_i \in \mathbb{T}_j$ and $P_j \in \{3 \text{ ms}, 5 \text{ ms}, 7 \text{ ms}, 11 \text{ ms}\}$. For the harmonic task sets, the same approach was used, but the random factor r was always selected from the set $\{2^i \mid i \in \{0, 1, 2, 3, 4\}\}$ to

Workload	Number of additions	Timer periods
low	1000	3, 5, 7, 11
high	10000	3, 5, 7, 11
harmonic single	100	1
harmonic low	1000	3, 5, 7, 11
harmonic high	10000	3, 5, 7, 11

TABLE I
OVERVIEW OF ALL TEST CONFIGURATIONS

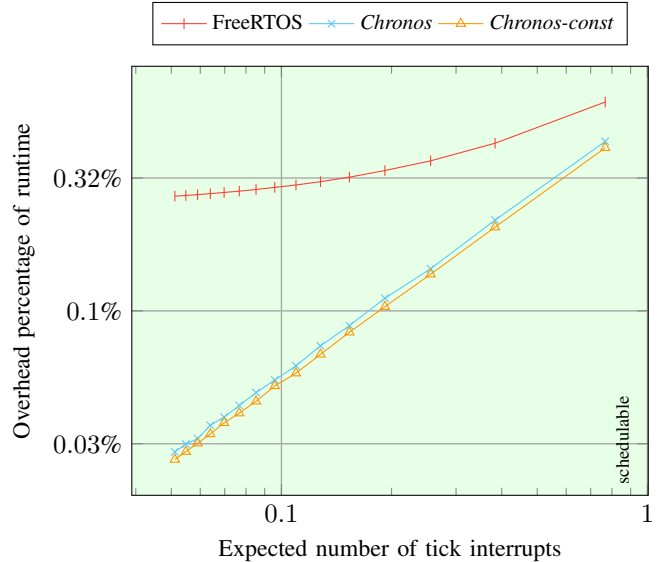


Fig. 6. Non-harmonic task sets with a low single-job workload

only create tasks with harmonic periods. The random factor r was always sampled from a uniform distribution. We used Gurobi [19] to implement and solve the MIQCP.

The generated tasks have periods that are multiples of the timer periods to be more representative of realistic test scenarios. If all tasks had the same period as their associated timer, its `TimerList` would always contain all tasks of that partition when the tick interrupt occurs, assuming no task misses its deadline. Consequently, all tasks would always need to be released then as well. A summary of the different test configurations can be found in Table I.

Finally, to evaluate the impact of the different methods for task sets with increasing task periods, a common *period factor* $p \in \mathbb{N}$ was introduced that is used to uniformly scale the timer and task periods. For a given period factor p , all timer periods and all task periods were multiplied by p . Every scenario was evaluated for period factors between 1 and 15, as testing has shown that the trends continue in the same manner for larger period factors.

A. Non-Harmonic Task Sets

In the following, *Chronos* and *Chronos-const* are evaluated against the baseline FreeRTOS implementation that uses a single timer.

In every plot, the period factors for which the task set experiences deadline misses regardless of the task dispatching

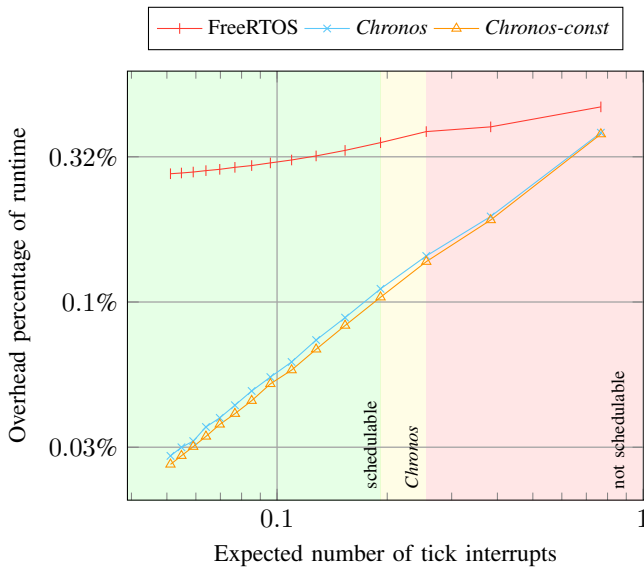


Fig. 7. Non-harmonic task sets with a high single-job workload

method is colored with a red background and labeled *not schedulable*. For period factors where no deadline misses occur with any method, the background is colored green and labeled *schedulable*. In general, a task set is called *schedulable* here, if under any method, the task set can be executed without any deadline misses. If, for a particular period factor, deadline misses occur when using the baseline implementation but not when using *Chronos* or *Chronos-const*, the region is colored with a yellow background and labeled *Chronos*. Lastly, if all methods other than *Chronos-harmonic* experience deadline misses, the region is colored with a blue background and labeled *harmonic*. It should be noted that the range of the y-axis differs between plots of different scenarios as the range of the measured data also differs vastly.

a) *Low Workload Scenario*: First, the overhead measurements for non-harmonic task sets with a low single-job workload are presented in Figure 6.

The y-axis shows the percentage of the total runtime that is spent on delaying tasks and handling tick interrupts. On the x-axis, the expected number of tick interrupts when using *Chronos* or *Chronos-const* is shown for period factors $p \in [1, 15]$. This is normalized to the expected number of tick interrupts when using the baseline implementation and can be calculated as $\frac{1}{P_1} + \frac{1}{P_2} + \frac{1}{P_3} + \frac{1}{P_4}$. A value of 0.5 as the expected number of tick interrupts means that the multi-timer methods cause half the number of tick interrupts compared to the baseline in a fixed time interval. It should be noted that both axes of the overhead measurements are logarithmically scaled since the values on both axes are fractions.

At first glance, it can be seen that the task sets for all period factors are schedulable under all task dispatching methods. The measurements show that *Chronos* and *Chronos-const* provide significant reductions in overhead compared to the baseline implementation in FreeRTOS that only uses a single timer.

Furthermore, for increasing period factors, the percentage of the runtime that is spent on blocking tasks and handling tick interrupts constantly decreases. The overhead for the baseline implementation in FreeRTOS also decreases, albeit considerably slower. This can be explained by the number of tick interrupts that occur when using one of the multi-timer methods compared to the baseline implementation. For *Chronos* and *Chronos-const*, the number of interrupts that occur does not change for longer period factors. This is because all task periods are uniformly scaled with the period factor, resulting in the GCD of every partition also being scaled in the same manner. Also, for longer running programs, the percentage of the overall runtime that is spent on delaying tasks and handling tick interrupts also naturally decreases as only a small portion of the runtime is overhead.

Another observation can be made about the difference in overhead caused by *Chronos* and *Chronos-const*. While both methods cause a similar amount of overhead, *Chronos-const* always causes slightly less overhead compared to *Chronos*. In this scenario, the trade-off made in *Chronos-const* is worthwhile with regard to the overhead. All in all, for this task set, the multi-timer methods provide a considerable reduction in overhead compared to the baseline implementation for all period factors.

b) *High Workload Scenario*: In Figure 7, the measurements for non-harmonic task sets with a high single-job workload are presented. At first glance, it can be seen that for period factors of one to three, deadline misses occur regardless of the employed task dispatching method. Also, for a period factor greater than three, only when using *Chronos* or *Chronos-const* no deadline misses occur. That means, by changing the dispatching method, it is possible to make a task set schedulable that was previously not. Finally, for period factors greater than four, the task set becomes schedulable under all methods.

The trends of the overhead measurements match those of the low workload scenario. However, it can be observed that for a period factor of one, the overhead caused by *Chronos* is only slightly higher than the overhead caused by *Chronos-const*, meaning that for task sets where many deadline misses occur, *Chronos* can provide a similar system efficiency to *Chronos-const*. Nevertheless, *Chronos-const* still performs slightly better for all period factors.

B. Harmonic Task Sets

We further conducted experiments where the examined task sets are always harmonic. Therefore, it is possible to also evaluate *Chronos-harmonic* alongside *Chronos* and *Chronos-const* against the baseline implementation.

a) *Single Timer Tests*: First, in Figure 8, the impact on the overhead by using the techniques from *Chronos-const* and *Chronos-harmonic* without using multiple timers or adapting the timer period to the GCD of the task set are presented. Concretely, the data points for *const* show the effect of sacrificing an ordered `TimerList` for a faster task insertion. The data points for *harmonic* show the effects of using the

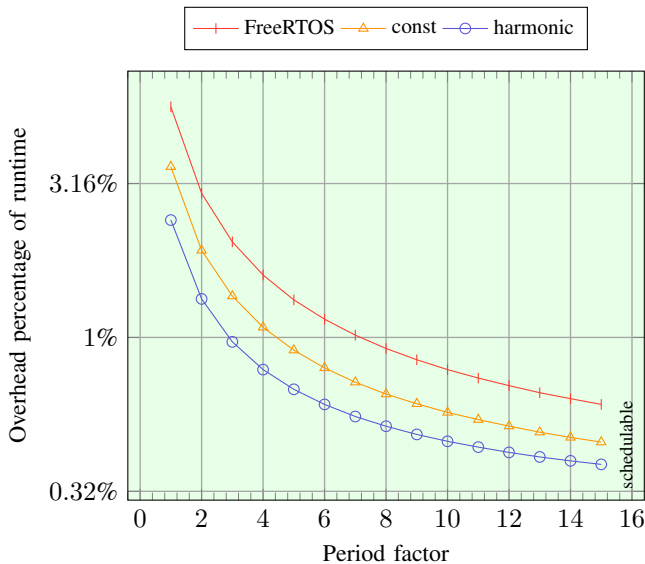


Fig. 8. Harmonic task sets with a single timer

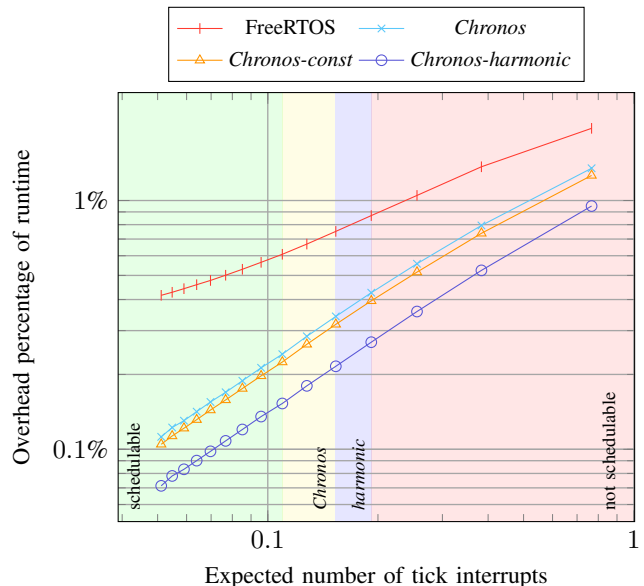


Fig. 10. Harmonic task sets with a high single-job workload

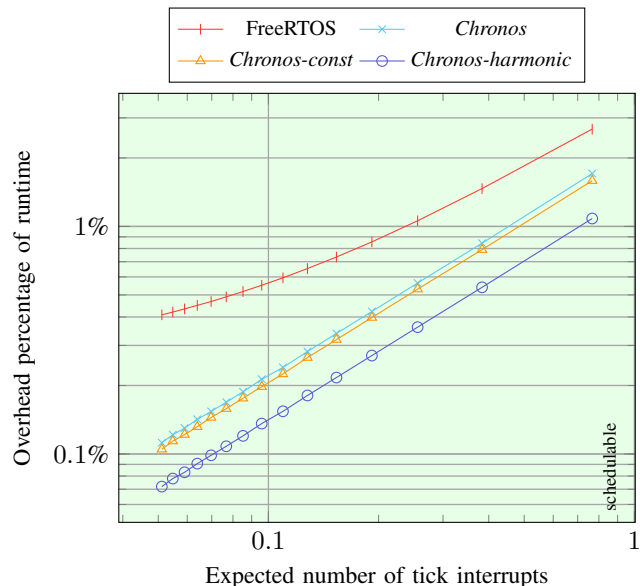


Fig. 9. Harmonic task sets with a low single-job workload

fixed-array-based dispatching method from Section V-B. For this test, all timer periods were fixed at 1 ms, and the number of additions a task has to perform was set to 100. It should be noted that, here, the period factor is shown on the x-axis as all methods use a single timer that has a fixed period.

At first glance, it can be seen that regardless of the period factor, all task sets are schedulable. While both methods improve the system efficiency in comparison to the baseline, the *harmonic* method further reduces the overhead compared to the *const* method. For this task set and test configuration, it is beneficial to utilize the *const* or *harmonic* methods even when only a single timer can be used whose period cannot be extended.

b) *Low Workload Scenario*: In Figure 9, the measurements for harmonic task sets with a low single-job workload are shown. Generally, all multi-timer methods provide a significant reduction of the overhead compared to the baseline. While *Chronos* and *Chronos-const* show similar results, using *Chronos-harmonic* can further reduce the overhead. In this scenario, the overhead decreases faster for increasing period factors for the multi-timer methods compared to the baseline implementation.

c) *High Workload Scenario*: Next, the measurements for the high single workload scenario for harmonic task sets are given in Figure 10. This plot shows that using *Chronos-harmonic* can enable more task sets to be schedulable compared to using *Chronos* or *Chronos-const*. In more detail, for period factors up to four, the task sets are not schedulable under any task dispatching method, while for period factors greater than four, task sets are schedulable when using *Chronos-harmonic*. Notably, only for period factors greater than five, task sets become schedulable under *Chronos* and *Chronos-const*. When using the baseline implementation, task sets become schedulable for period factors greater than seven. As a result, here, using *Chronos-harmonic* instead of *Chronos* or *Chronos-const* can allow further task sets to be schedulable.

C. Discussion

In this section, the methods presented in Section IV were evaluated on an embedded system. For all tested configurations, it can be observed that utilizing multiple hardware timers can reduce the overall time overhead. Also, for scenarios where a task set is not schedulable under the default FreeRTOS implementation with a single timer, it is possible that when employing one of the multi-timer methods, that task set will be schedulable. Additionally, for harmonic task sets, it can be beneficial to exploit the way harmonic tasks are released to

Scenario	Peak <i>chronos</i>	Mean <i>chronos</i>	Peak <i>const</i>	Mean <i>const</i>	Peak <i>harmonic</i>	Mean <i>harmonic</i>
Non-harmonic						
low	9.16×	4.64×	9.79×	4.94×	—	—
high	9.36×	5.98×	10.01×	6.37×	—	—
Harmonic						
single	—	—	1.56×	1.41×	2.33×	1.83×
low	3.64×	2.52×	3.89×	2.69×	5.68×	3.94×
high	3.7×	3.07×	3.97×	3.3×	5.83×	4.82×

TABLE II
REDUCTION OF OVERHEAD COMPARED TO FREERTOS

further reduce the overhead. Moreover, for scenarios where only a single timer can be used whose period cannot be extended, the *const* and *harmonic* methods can still reduce the overhead compared to the baseline implementation.

The evaluation results are summarized in Table II. For every tested scenario, the peak reduction of the overhead when using one of the presented methods compared to the baseline implementation is listed. In addition, the geometric mean of the overhead reduction over all task sets that are schedulable under any method is also listed for every presented method. The results for *Chronos* are given in the columns labeled *chronos*, and the results for *Chronos-const* are shown in the columns labeled *const*. For the harmonic task sets, the results for *Chronos-harmonic* are listed in the columns labeled *harmonic*.

For the tests with non-harmonic task sets, *Chronos-const* achieves a reduction of the overhead of 9.79×–10.01× in peak and 4.94×–6.37× on average. *Chronos* is slightly less efficient and reduces the overhead by 9.16×–9.36× in peak and 4.64×–5.98× on average. For the harmonic task sets with only a single timer, the *const* method reduces the overhead by 1.56× in peak and 1.41× on average. The *harmonic* method is more efficient and reduces the overhead by 2.33× in peak and 1.83× on average. Next, for scenarios with multiple timers, *Chronos* reduces the overhead by 3.64×–3.7× in peak and 2.52×–3.07× on average, while *Chronos-const* achieves an overhead reduction of 3.89×–3.97× in peak and 2.69×–3.3× on average. *Chronos-harmonic* is more efficient here as it reduces the overhead by 5.68×–5.83× in peak and 3.94×–4.82× on average.

VIII. CONCLUSION

In this paper, we examined how to utilize multiple hardware timers to reduce the overhead of the task handling process in an RTOS. We formulated an MIQCP that partitions a given task set and assigns each partition to a different hardware timer, such that the overall number of tick interrupts is minimized. Additionally, we also investigated how to further reduce the overhead by changing the way the set of delayed tasks is organized. The first method, *Chronos*, distributes the task set over multiple timers and keeps a sorted list for the delayed set of every timer. *Chronos-const* shifts the organization overhead from the task to the interrupt handler, by keeping an unstructured list for the set of delayed tasks that has to be iterated completely in every tick interrupt. Lastly,

Chronos-harmonic is an optimization specifically for harmonic task sets. Here, the properties of harmonic tasks are exploited to enable a constant time insertion and retrieval of delayed tasks.

In order to examine the impact on the overall time overhead caused by the different methods, multiple scenarios for harmonic and non-harmonic task sets were evaluated on a real-world embedded system. For non-harmonic task sets, both *Chronos* and *Chronos-const* are more efficient than the baseline implementation of FreeRTOS with a single timer. Generally, for the tested configurations, *Chronos-const* always outperforms *Chronos* and reduces the overhead compared to the baseline by up to $\approx 10\times$ in peak and $\approx 6\times$ on average.

As *Chronos-harmonic* can only be applied to harmonic task sets, several tests were performed with harmonic task sets. First, it was evaluated whether the techniques used by *Chronos-const* and *Chronos-harmonic* also reduce the overhead when only a single timer with a fixed period is used. In this scenario, using *Chronos-harmonic* results in the least overhead, as the overhead is reduced by up to 2.33× in peak and 1.83× on average compared to the baseline. For scenarios where multiple timers could be used, *Chronos-harmonic* performs the best by reducing the overhead by up to 5.83× in peak and 4.82× on average.

In conclusion, the presented methods can be used to reduce the overhead in the task handling process that is incurred by an RTOS, if the hardware platform provides multiple timers. As a result, task sets that were not schedulable when a single timer is employed, can be schedulable with multiple timers.

REFERENCES

- [1] W. Hofer, D. Danner, R. Müller, F. Scheler, W. Schröder-Preikschat, and D. Lohmann, "Sloth on time: Efficient hardware-based scheduling for time-triggered rtos," in *2012 IEEE 33rd Real-Time Systems Symposium*, 2012, pp. 237–247.
- [2] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, "An empirical survey-based study into industry practice in real-time systems," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020, pp. 3–11.
- [3] G. Varghese and A. Lauck, "Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility," vol. 5, no. 6, pp. 824–834.
- [4] M. Aron and P. Druschel, "Soft timers: efficient microsecond software timer support for network processing," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, p. 197–228, aug 2000.
- [5] P. Kohout, B. Ganesh, and B. Jacob, "Hardware support for real-time operating systems," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 45–51.
- [6] F. Hagens and K.-H. Chen, "Assessment of efficient dispatching in freertos," in *The 17th annual workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPert)*, 2023, pp. 7–9. [Online]. Available: <https://www.ecrts.org/workshops/ospert23/>
- [7] R. Balas and L. Benini, "Risc-v for real-time mcus - software optimization and microarchitectural gap analysis," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 874–877.
- [8] A. Easwaran, I. Lee, O. Sokolsky, and S. Vestal, "A compositional scheduling framework for digital avionics systems," University of Pennsylvania Department of Computer and Information Science, techreport.
- [9] H. Li, J. Sweeney, K. Ramamritham, R. Grupen, and P. Shenoy, "Real-time support for mobile robotics," in *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings.*, pp. 10–18.

- [10] S. Kramer, D. Ziegenbein, and A. Hamann, “Real world automotive benchmarks for free,” in *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, vol. 130, 2015.
- [11] J. Busquets-Mataix, J. Serrano, R. Ors, P. Gil, and A. Wellings, “Using harmonic task-sets to increase the schedulable utilization of cache-based preemptive real-time systems,” in *Proceedings of 3rd International Workshop on Real-Time Computing Systems and Applications*, pp. 195–202.
- [12] T.-W. Kuo and A. Mok, “Load adjustment in adaptive real-time systems,” in *[1991] Proceedings Twelfth Real-Time Systems Symposium*, 1991, pp. 160–170.
- [13] M. Nasri, G. Fohler, and M. Kargahi, “A framework to construct customized harmonic periods for real-time systems,” in *2014 26th Euromicro Conference on Real-Time Systems*, pp. 211–220.
- [14] G. von der Brüggen, N. Ueter, J.-J. Chen, and M. Freier, “Parametric utilization bounds for implicit-deadline periodic tasks in automotive systems,” in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, ser. RTNS '17. New York, NY, USA: Association for Computing Machinery, pp. 108–117.
- [15] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.
- [16] Richard Barry and The FreeRTOS Team, *Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide*, release version - 1.0 ed., Amazon Web Services. [Online]. Available: <https://www.freertos.org/Documentation/Mastering-the-FreeRTOS-Real-Time-Kernel.v1.0.pdf>
- [17] E. Systems, *ESP32-S3 Technical Reference Manual*, 2024. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-s3_technical_reference_manual_en.pdf
- [18] Cadence Design Systems, Inc., *Xtensa Instruction Set Architecture (ISA) Summary*, 2022. [Online]. Available: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/silicon-solutions/compute-ip/isa-summary.pdf
- [19] Gurobi Optimization, LLC, *Gurobi Optimizer Reference Manual*, 2024. [Online]. Available: <https://www.gurobi.com>