Task-Based Constant Bandwidth Server in the Zephyr Operating System

Alexander Paschoaletto
SoftCPS/ISEP, Porto, Portugal

Paulo Baltarejo Sousa
SoftCPS/ISEP & INESC TEC, Porto, Portugal

Luis Miguel Pinho
SoftCPS/ISEP & INESC TEC, Porto, Portugal

Tiago Carvalho
SoftCPS/ISEP & INESC TEC, Porto, Portugal

Abstract

The Constant Bandwidth Server (CBS) is a widely used method to support aperiodic soft real-time tasks in a system that uses dynamic scheduling algorithms, such as Earliest Deadline First (EDF), while providing end-to-end temporal guarantees through bandwidth reservation. We have recently proposed an approach to integrate CBS with the open-source real-time operating system, Zephyr, which involves developing CBS as a separate kernel component that can be shared by multiple execution contexts. In this paper, we propose an alternative approach, which provides each task with a dedicated CBS instance, which enables fine-grained control over task execution. The paper also presents a richer support for EDF scheduling in Zephyr, which is used to support the Task-Based CBS. The proposed method is validated through test cases, demonstrating its efficiency in supporting aperiodic real-time tasks with bandwidth constraints in Zephyr.

2012 ACM Subject Classification Computer systems organization → Real-time operating systems; Theory of computation → Concurrent algorithms

Keywords and phrases Constant Bandwidth Server, Zephyr Operating System

Digital Object Identifier 10.4230/OASIcs.AEiC.2026.6

Funding The work is co-funded by the Portuguese Innovation Agency (ANI) through the COMPETE 2030 and NORTE 2030 Programmes, with the support of the European Regional Development Fund (ERDF), within projects GenerIoT (COMPETE2030-FEDER-00355700) and HomePot (NORTE2030-FEDER-01241500).

Acknowledgements The authors would like to thank the anonymous reviewers for their helpful comments and feedback.

1 Introduction

Typically, real-time applications consist of a heterogeneous set of activities (i.e., tasks) with varying timing constraints and execution characteristics competing for common processing resources such as central processing unit (CPU) and memory. Some tasks may require strict temporal guarantees with a periodic execution pattern and a well-defined worst-case execution time, with any missed deadline having significant consequences, such as in control loops for industrial processes or robotic systems. On the other hand, a significant number of tasks may have a varying execution pattern, often referred to as an *event-driven* pattern, such as tasks related to sensor processing and computer vision tasks, with timing variations affecting performance that may eventually be recovered without compromising correctness. These two categories of tasks are typically referred to as *hard* and *soft* real-time tasks, respectively.

The theory of real-time scheduling has mostly been developed under the assumption that all tasks are periodic and that their worst-case execution times (WCET) can be bounded. However, it is well known that calculating the WCET is a difficult problem, especially for complex applications such as multimedia and data-dependent computations, where the WCET may be much longer than the actual expected time [1]. Moreover, aperiodic and sporadic tasks, which are now widespread in many embedded systems, do not easily fit into the periodic model. Hence, it is difficult to use schedulability tests when hard real-time guarantees are required.

A well-known approach to solve this problem is to use resource reservation servers, which offer temporal isolation via bandwidth allocation to each activity. In this way, soft and aperiodic tasks can be supported along with hard real-time tasks without compromising schedulability. Among these techniques, bandwidth-preserving servers have been extensively studied because of their capacity to offer utilization bounds while allowing efficient CPU utilization with dynamic scheduling algorithms like earliest deadline first (EDF). In addition to offering predictability [7], servers are also one of the main techniques to offer isolation in mixed-critical systems [8].

The Constant Bandwidth Server (CBS) is one of such reservation-based mechanisms, and allows soft or non-real-time tasks to execute together with hard real-time tasks scheduled under EDF while preserving temporal isolation [1][6]. Each served task is assigned a maximum execution budget, limiting the interference caused by execution overruns and enabling schedulability analysis of hybrid task sets.

The implementation of these servers depends in large part on the support provided by real-time operating systems (RTOS). In fact, the RTOS should provide appropriate scheduling primitives and timer support together with kernel abstractions in order to manage the execution of the server and the synchronization and deadline control of the tasks. The decisions made at the operating system (OS) level have significant effects on the integration and flexibility of the reservation techniques that are used in various scenarios.

This paper builds upon our previous proposal [16], which presented a Constant Bandwidth Server (CBS) implementation on top of the Zephyr OS [20]. In that paper, we implemented CBS as a set of kernel objects that function as a separate server accepting execution requests from tasks or interrupt handlers. This architecture, here called *Decoupled* CBS, does not rely on any particular task and provides a generic interface that can be shared among various execution contexts.

Based on this foundation, and motivated by the integration in a satellite software framework, we propose an alternative CBS integration model that we call *Task-Based* CBS, in which the reservation mechanism is mapped onto the tasks' applications and thus each task using CBS is managed by its own private CBS. This model provides another trade-off between modularity, integration, and control. Moreover, this paper describes the novel EDF scheduling support in Zephyr that is used to support the proposed Task-Based CBS model.

The paper is structured as follows. The next section provides the background, with an overview of CBS, Zephyr, and related work. Section 3 introduces the richer EDF support provided in Zephyr, and then section 4 the Task-based CBS implementation proposed in this paper. Section 5 provides a comparative analysis of both implementations, both by performing a behaviour and performance analysis in representative test cases, as well as assessing the cases of use of the each of the approaches. Finally, Section 6 concludes with a summary and discussion of future work.

2 Background and Related Work

This section provides a summary of the Constant Bandwidth Server (CBS), as well as the Zephyr Operating System, before presenting the related work on the implementation of CBS in real-time operating systems, and finally describing the previous proposed implementation of CBS in Zephyr.

2.1 Constant Bandwidth Server

We consider a preemptive uniprocessor real-time system composed of n hard real-time tasks $\tau_1 ... \tau_n$ and m soft real-time tasks $j_1 ... j_m$. Each hard real-time task τ_i has worst-case execution time C_i, relative deadline D_i, and period T_i, with $0 \leq C_i \leq D_i$ and $D_i = T_i$. Hard tasks are independent and scheduled directly by EDF. Soft tasks are executed by Constant Bandwidth Servers (CBS), and inherit the server absolute deadline (d_i).

A CBS is characterized by two parameters (Q, P), where Q is the *server maximum budget* and P is the *server period*. The ratio $U_i = Q_i/P_i$ represents the *server bandwidth*, corresponding to the fraction of processor capacity reserved for the served workload. The current server budget varies between 0 and Q during execution and is denoted by q.

The server bandwidth $U_i = Q_i/P_i$ can be treated as equivalent to the utilization of a periodic task. For a system with n periodic tasks scheduled by EDF and total utilization

$$U_p = \sum_{i=1}^{n} C_i/T_i \tag{1}$$

and m CBS servers serving aperiodic tasks with total bandwidth

$$U_s = \sum_{j=1}^{m} Q_j/P_j \tag{2}$$

the total system utilization becomes

$$U_r = U_p + U_s \tag{3}$$

which must satisfy $U_r \leq 1$ to guarantee schedulability under EDF. This condition holds due to the isolation provided by the bandwidth reservation mechanism [7].

The parameters Q_i and P_i are determined off-line by the system designer [6]. At runtime each server maintains the remaining budget q_i and the absolute deadline d_i, initially set to $d_i = 0$.

1. A CBS is *idle* when no jobs are pending or executing; otherwise it is *active*.
2. When a job arrives at an idle server, the following condition is verified:

   $$q_i \geq (d_i - r_{job})U_i \tag{4}$$

   where r_{job} is the arrival time of the job. If the condition holds, the server deadline and budget are reset to $d_i = r_{job} + P_i$ and $q_i = Q_i$.
3. During execution the budget q_i decreases with processor usage. When the budget is exhausted, it is replenished to Q_i and the deadline is postponed by P_i, i.e., $d_i = d_i + P_i$.

The server deadline is used by the EDF scheduler to determine the execution priority of the server relative to other tasks.

2.2 Zephyr

Zephyr [20] is an open-source RTOS which focuses on featuring a lightweight, yet scalable kernel that suits a myriad of devices, from embedded microcontrollers to complex multi-core systems, and it is currently ported to more than 600 boards and over 10 architectures. It also includes features such as semaphores, mutexes, interrupts, timers and atomic variable operations.

As in the majority of OSs, support to concurrent activities in Zephyr is provided using threads. Zephyr offers a handful of scheduling algorithms that can work together in a hierarchical manner, with the main adopted policy being the user-defined static priority: each thread is assigned an integer value which can be positive or negative, and the ready thread with the *lowest* number holds the highest priority. Positive values are meant for preemptive threads, while negative values are for cooperative threads. A cooperative thread cannot be preempted, except for the execution of interrupt service routines (ISRs), and will remain in the CPU until it voluntarily yields or blocks waiting for external resources or synchronization mechanisms such as a mutex [19].

When two or more threads of the same priority level are eligible for execution, the scheduler relies on an auxiliary scheduling algorithm to determine which will take the CPU. The currently available methods are First-in-First-out (FIFO), the default, that selects the thread that has been waiting longest, the Round-Robin (RR) scheduler, which assigns a time slice for each thread, successively preempting them in favour of the others at the end of each slice, and the EDF policy, where the thread with the earliest absolute deadline is chosen to execute. As for the data structure for the ready queue, the kernel can be configured at build time with one of a few implementation choices such as simple linked-lists, red/black trees and the multi-queue approach for static priority assignment.

2.3 Related Work

To the best of our knowledge, CBS has been implemented with some variations on two open-source OSs: Linux and RTEMS, as well as the PikeOS commercial OS. Linux introduced the SCHED_DEADLINE scheduler in kernel version 3.14 to handle real-time tasks with an algorithm built from EDF and CBS concepts [9]. In this algorithm, operating system threads declare a maximum "runtime" Q, a relative deadline D and a period P, and whenever the thread implementing a task becomes ready for execution, the CBS condition (Equation 4) is applied to calculate the absolute deadline. Then EDF proceeds with executing the ready thread with the earliest absolute deadline [10].

However, a fundamental change from the original CBS is that the condition is applied at *every* moment that the thread becomes ready (e.g. resuming after a suspension) instead of just at the arrival instant. Another significative change is that, whenever the declared "runtime" is exhausted, the thread is immediately suspended until its absolute deadline (it then resumes with a replenished "runtime" and an absolute deadline postponed in P time units). A similar hard reservation approach is used in the PikeOS implementation [18], which implements hard CBS-servers, with the goal to provide protection to unbounded interference to lower-priority, but higher criticality jobs. This is based on a previous implementation in the research LitmusRT OS, which nevertheless was never released [5].

As for RTEMS, an RTOS qualified for and widely used in the space domain, the existing CBS implementation [4] targets single-core applications and each CBS can only be attached to one periodic task at time (thus, they will only execute the released jobs of this one task).

There are also differences when compared to the original CBS model: when the job exceeds its allowed budget, it is removed from the scheduler (i.e. blocked until its next period, even if there is available CPU capacity)[1], which requires a more complex analysis [2, 3].

The CBS condition is only verified when the task attempts to unblock, and if satisfied, the CBS remains blocked until its next period instead of being replenished. Moreover, the budget consumption is measured with tick interrupts rather than through the usage of dedicated timers [17].

These implementations lose the *work conserving* property of the theoretical model (see [3]) and are prone to greater overhead, as the CBS condition is checked more frequently. Nevertheless, they are able to mitigate the *deadline ageing* effect [11], guaranteeing a minimum execution time to a task in a fixed interval of time. The approach for Zephyr presented on this paper (similar to the one in [16]) follows the original CBS: the CBS condition is only checked when a job is pushed to the server, and budget exhaustion events only postpone the deadline instead of suspending the job.

2.4 Decoupled CBS

This section summarizes the Decoupled CBS architecture and its implementation in Zephyr RTOS as presented in [16]. The implementation is available as a branch of the Zephyr project hosted on GitHub [12].

In this design, a CBS *job* is implemented as a regular C function executed within the context of a CBS thread. Jobs can be activated from any execution context (thread or ISR) and do not possess intrinsic timing attributes such as deadlines, execution time bounds, or activation periods. Instead, the CBS enforces the timing constraints required to preserve system schedulability when executing alongside hard real-time tasks scheduled under EDF.

Each CBS instance is composed of three main components: a Job Queue, a Thread, and a Timer. Multiple CBS servers may coexist in the system, each maintaining its own instance of these elements. The Job Queue stores pending jobs using a FIFO structure. The CBS thread is responsible for retrieving and executing jobs whenever the server becomes eligible to run under the EDF scheduler. The Timer is used to track the server's budget consumption during execution.

Jobs are inserted through the `k_cbs_push_job()` interface, which wraps the underlying Zephyr message queue API. Upon insertion, the CBS condition described in Section 2.1 is evaluated. If the condition holds, the server budget is replenished and its absolute deadline updated. When jobs are available, the CBS thread wakes up and enters the scheduler run queue. Once its absolute deadline becomes the earliest among runnable tasks, it gains the CPU and executes the next job in the queue.

Budget consumption is tracked using a kernel timer associated with the CBS. The timer is started when job execution begins and stopped when execution finishes. If the timer expires before the job completes, the server budget is exhausted and replenished according to the CBS rules (Section 2.1), with the server deadline postponed accordingly. During preemptions, the timer is temporarily stopped and resumed when the CBS thread regains the CPU so that only actual execution time contributes to budget consumption.

The implementation exposes a minimal API consisting of the `K_CBS_DEFINE` macro for static server declaration and the `k_cbs_push_job()` function for submitting jobs to the server queue. The CBS behaviour can also be configured at build time through Zephyr's Kconfig system, including options for enabling the CBS subsystem, configuring queue sizes, adjusting thread stack size, and enabling logging facilities.

[1] Note that the RTEMS documentation refers to this suspension as pulled to "background" until the next period, which can be misleading, as background execution is sometimes used to denote execution when the core is free and no other task is ready.

3 Richer EDF support

The new EDF scheduler greatly enhanced the capabilities of the pre-existing Zephyr EDF API, as the latter has an simplified implementation of EDF, lacking native support for period management, integrated deadline miss detection, and flexible runtime control over task timing behaviour. The goal of the work was to develop an API with a usability close to fixed priority approaches, so that tasks would be able to alternate between EDF and rate monotonic (RM) with minimal effort, while also providing means for synchronous deadline miss detection and the ability to run EDF on single-shot tasks as well.

A set of new methods and changed methods are provided for enabling the new EDF scheduler:

- `k_thread_period_set`: optionally set the period of which the EDF task should be running, in milliseconds. Note this function is merely a setter and will produce no immediate effect on the task scheduling. If configured, the period will be later enforced by calls to the `k_thread_deadline_set` method. If desired, the task can invoke this method more than once to update its own period. Doing so will cause the current cycle to immediately shift towards the new value [2].
- `k_thread_deadline_miss_callback_set`: optionally sets a callback function, as well as a custom argument for it, to be executed whenever a task misses its absolute deadline. This method is also just a setter and will not produce any immediate effect apart from updating the EDF manager control block. A task may invoke this API more than once to update or enable/disable the synchronous track of the deadline state.
- `k_thread_deadline_set`: this is the most important method for enabling EDF, and it was changed to implement the new functionality. If the task has configured a period, it applies suspensions between cycles to provide regular activation intervals (also warning about period overruns, if detected). Furthermore, automatically starts/stops the EDF timer in case a deadline miss callback is configured. Although a separate function could have been provided, the decision was to reduce the number of new functions, since the new behaviour is only activated when a period is set.

Thus, the present API offers a handful of options for a task to be scheduled by EDF, allowing tasks to be non-periodic. One can configure it for a periodic section or just enable EDF for parts of its logic which execute only once. It is also possible for a task to enable a deadline miss detection, and through a custom callback decide what to do when an overrun happens. Period overflows are also detectable, and all settings can be changed or toggled mid-execution. Moreover, all these settings are put in motion by a single method, `k_thread_deadline_set`, and in practice any given task already fit for fixed priorities can switch to EDF with minimal adaptations required.

Figure 1 illustrates the execution flow of the `k_thread_deadline_set` method, central to the approach (in yellow the steps specific for the CBS, which will be explained in the next section). First, it is important to understand the flow for the EDF behaviour. When called, if no period or deadline callback is configured, this method will simply set the absolute deadline of the calling task . If a deadline callback is configured, the EDF timer will start right after the deadline is set. This flow is the upper path in the figure.

[2] The next activation is always calculated based on the most updated period, so if a shorter one is configured mid-cycle, the task will always consider the new value when calculating the time to suspend until the next cycle. Defining a period smaller than the task's current execution time simply means no task suspension at all

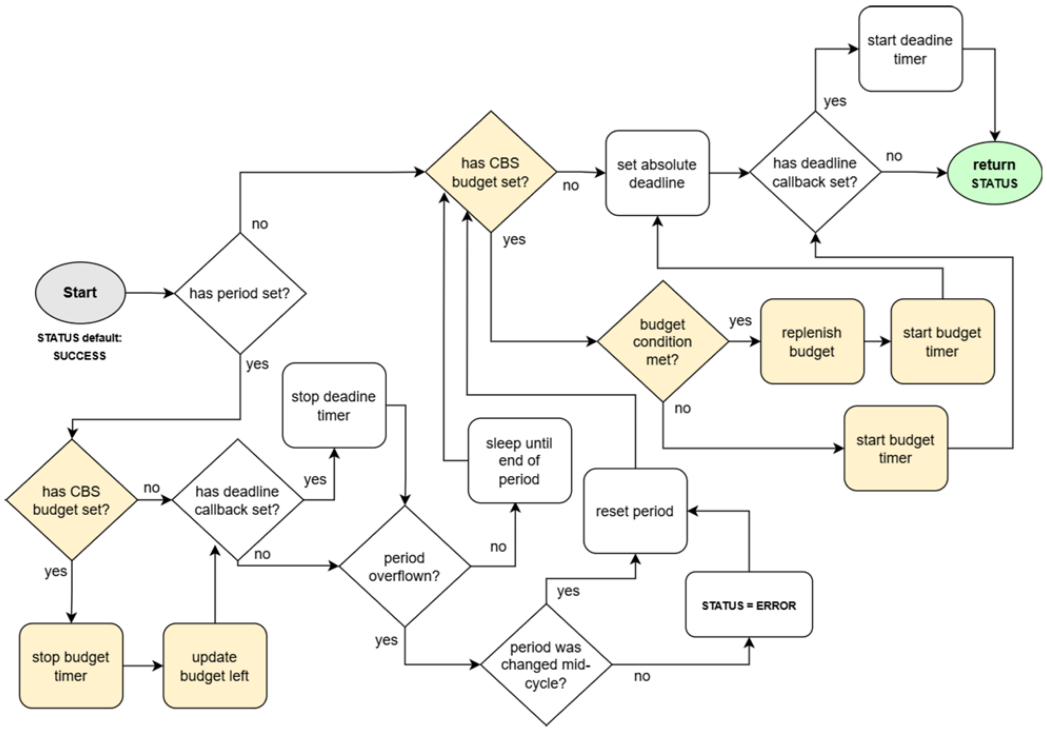


Figure 1 k_thread_deadline_set algorithm with the CBS extension.

If there is a defined period, the lower path is taken instead, being responsible for both the end of the last cycle and the beginning of the next cycle. As such, the first thing to do is check if the previous cycle finished on time, and, if true, forcing the task to suspend for the remainder of the configured period. By the time that is completed, the new cycle begins and the task may set its new absolute deadline, following the same steps of the upper path. Naturally, if at every time this method is invoked the current cycle is about to end, the EDF timer must be interrupted before anything else to prevent false deadline miss detections. That is the reason why the configuration or not of a callback is verified on both routes: the same check is performed, but the outcomes derived from it are different. The method also performs a check for whether or not the task updated the period length during its last cycle, as in `k_thread_deadline_set` the period is an extra setting that can be altered with a call to an auxiliary `k_thread_period_set` function. For the EDF method, this verification only needs to take place if there was a period overflow in the latest cycle. Two possibilities arise from this event: either it happened because the task executed (or was preempted) for longer than expected, or it deliberately altered this value to be shorter than the previous execution time. The latter implies that the application knows what it is doing and it is not considered an error, but the former clearly denotes an error situation which would be considered a deadline miss. Under these terms, the `k_thread_deadline_set` will return an error status upon completion, allowing the calling task to take action if desired. Otherwise, the execution returns the default successful status. Thus, for the developed EDF API, deadline miss events are detected immediately (with the assistance of timers) and period overflow events are detected asynchronously, by the time the `k_thread_deadline_set` method gets to run again. Although undesired, period overflows are not necessarily critical in EDF, hence why it was assumed an asynchronous detection would be sufficient.

This implementation was evaluated in [15], showcasing the correct behaviour of the EDF API implementation (available at [13]).

4 Task-based CBS

The original Decoupled CBS implementation presented a few limitations, which became clear when integrated in a wider project aimed at porting a satellite software framework to Zephyr [15]. First, the Decoupled CBS approach required configuring the CBS in a static manner (at compile time), whereas the satellite software framework initializes nearly all of its components at runtime. Second, the decoupled CBS in Zephyr was tailored for executing arbitrary jobs (materialized as isolated functions) within an internal thread, whereas in the framework all executive entities are explicitly defined at application level through either tasks or interrupts. Finally, the original CBS was developed around the very simple EDF implementation contained in Zephyr, not taking advantage of a richer EDF Zephyr implementation, developed for the framework [15].

The main design change between theory and this new implementation was thus to make the CBS tailored for the regulation of application tasks rather than arbitrary jobs. In this model, each task opting to be scheduled by EDF would now be able to have one exclusive CBS to modulate its execution, as if each task cycle corresponded to a job release in the theoretical model. A task-centric approach greatly simplifies the implementation and cuts some edge cases that could not be avoided in the contribution made for Zephyr in [16]. If the jobs are now cycles of one same task and cycle starts are necessarily registered, then there is never more than one job at once in the system. This means there is no need to check if the CBS was idle by the time a new job comes simply because it will always be finished with the previous cycle by the start of the next, even if it missed a deadline. Thus, the CBS can always be assumed idle when a new job is invoked and the existence of a job queue becomes redundant.

The satellite software framework itself: a) has a considerable amount of additional procedures both for ensuring data integrity (which is expected for space-grade software) and enabling the framework's own business logic, and b) could not be made available as the framework is not public. Therefore, the Task-based CBS was separated as an independent contribution, featuring the components necessary for the periodic EDF [13] and CBS [14] to work. In this approach, there is exactly one CBS for each application task, and the task execution itself counts as the "job" to be regulated by the configured bandwidth as soon as the task declares its deadline. This alternative design was conceived to mirror the RTEMS offering, but without the hard reservations to keep the close resemblance with the theoretical CBS model. The resulting implementation was also improved to leverage the detachment from the framework and better integrate itself with the kernel while maintaining the rationale above.

Using the richer EDF support, a simple example on how to program a periodic CBS task is presented on Listing 1, and has the source code available in [14]. One can notice the time values for budget, deadline and period are now defined with Zephyr's built-in macros, ensuring a greater flexibility than the previous approach. The periodic handling of the task that happens within Zephyr's `k_thread_deadline_set` API, which received the necessary changes internally, as specified in the previous section, but will still behave *exactly* the same as the standard API when no period, budget or deadline callback is defined for the given task. Thus, this contribution is completely additive, without any breaking changes compared to the mainline Zephyr API. Optionally, the developer may invoke `k_reschedule` right after setting the task's deadline to signal a rescheduling point to the kernel, which does not happen by default in Zephyr once the task updates its own deadline.

Listing 1 Basic CBS thread in the Task-based version.

```
void task(void *bgt, void *ded, void *per) {
    /* setup */
    k_timeout_t budget = K_USEC(bgt);
    k_timeout_t deadline = K_MSEC(ded);
    k_timeout_t period = K_MSEC(per);
    k_tid_t thread = k_current_get();

    /* budget set is mandatory for threads with CBS */
    k_thread_cbs_budget_set(thread, budget);
    /* period set is optional for EDF */
    k_thread_period_set(thread, period);

    /* loop */
    while(true){
      k_thread_deadline_set(thread, deadline);
      k_reschedule();
      do_work();
    }
}
```

The CBS support in `k_thread_deadline_set` consists on applying extra steps (in yellow in Figure 1) at the start and end of the method to check if the task has a configured CBS, and if so, perform the necessary logic corresponding to a job's end and beginning, respectively. When the CBS is enabled, the task will check, in the upper path right before setting the new deadline, if the bandwidth condition is met, i.e. if there is still a considerable amount of budget left from a previous execution cycle compared to the current absolute deadline. If true, the deadline is in fact recalculated and the budget is replenished to match the predefined bandwidth. If false, the task will not recalculate its deadline (or refill the budget). That happens because an unmet condition implies the task can be served with the current deadline without risking schedulability guarantees, thus making the deadline setting unnecessary. If the task has configured a period, the lower path of Figure 1 is traversed, and the CBS both stops the budget timer and updates the budget level to reflect what's been consumed.

The steps added for the CBS are responsible for updating a task's budget based on how much was spent during the previous execution cycle. However, they assume for calculations that the task remained uninterrupted in the CPU between calls to `k_thread_deadline_set`, which is not true if a preemption ever happens in between. Some additional procedure was therefore required to suspend the consumption when preemptions take place; in practical terms, this meant introducing some code on context switches to pause the budget timer when a CBS task is forced to leave the CPU, and resume it once it returns. This is exactly the strategy adopted in [16], from where the CBS timer was also taken from. To that effect, the first step then was to include a reference to the task's CBS in its own control block, thus enabling the kernel to fetch the CBS data from the thread when performing a switch. Then, the methods shown in Listing 2 were placed in the relevant context switching paths: `z_cbs_switched_in` to update the incoming thread's CBS, and `z_cbs_switched_out` to handle the outgoing counterpart. Both took as argument the CBS pointer inserted in the thread control block.

Listing 2 Context switch functions required for the CBS budget tracking.

```
void z_cbs_switched_in( TLCH_CBS_t* cbs ){
    if( cbs->isActive ){
       k_timer_start(
          &( cbs->budgetTimer ),
          K_TICKS( cbs->currentBudget ),
          K_NO_WAIT
       );
       cbs->startTick=k_uptime_ticks();
    }
}
void z_cbs_switched_out(TLCH_CBS_t* cbs){
    if(cbs->isActive){
       k_timer_stop(&(cbs->budgetTimer));
       cbs->currentBudget-=(k_uptime_ticks()-cbs->startTick);
    }
}
```

As for the executed logic, it goes as follows. When a CBS-enabled task leaves the CPU, the scheduler checks if the CBS in question was active (consuming budget) or not. If so, it simply stops the budget timer (which will prevent deadline recalculations) and updates the budget left by subtracting the delta since the cycle start. Later on, when the task returns to the CPU, the budget timer is re-started with the updated value as expiration interval and the starting tick is also updated for future calculations. These steps guarantee that on the possibility of subsequent preemptions in one same cycle, the consumed budget will always reflect the actual execution time.

This task-based approach for the CBS, as explained in [15], has some particularities which surprisingly enable a simplification of the decoupled design (without altering any of the theoretical model properties). The most significant one is that, as each CBS is exclusive to a task, there is always at most one job for each CBS in the system, and thus no longer a need for a queue to store incoming jobs. Furthermore, since the job release coincides with the call to `k_thread_deadline_set`, which itself accepts the relative deadline of the task as a parameter, it made perfect sense to consider this value as the CBS period, ultimately sparing the need for an extra method in the API just for this effect.

Finally, the CBS condition would always be checked by the end of `k_thread_deadline_set`, independently of whether the task had or not periodic behaviour. This check would not be strictly needed every time: for example, on subsequent calls of this method for a non-periodic task, it would be still active by the second call (thus, the condition check was not necessary). However this would require additional checks for both the task state and a comparison between the new relative deadline and the last, which together would likely take the same amount of overhead as just checking the condition for every case. This is just an heuristic assumption, however, and verification is reserved for future work.

5 Evaluation

In order to assess the CBS implementation, a comparative analysis was performed, focusing on the algorithm behaviour correctness as well as the impact of the runtime overheads. This analysis is based on data collected from representative test cases running on a real device, a Seeed Studio XIAO ESP32C3, a 32-bits RISC-V microcontroller officially supported by Zephyr, the same cases and platform as in [16], therefore allowing a direct performance comparison between the two approaches. The code sample used for these tests is available on GitHub [14].

The test cases were:

- Test Case 1: A simple combination of one hard task with $(C_1, T_1) = (4, 7)$ time units and one CBS with $(Q, T) = (3, 8)$ time units, yielding a combined utilization factor of approximately 0.95. The hard task begins its periodic releases at $t = 0$. Two jobs are pushed to the CBS: $J_1(C = 4)$ at instant $t = 3$ and $J_2(C = 3)$ at $t = 13$.
- Test Case 2: A combination of two hard tasks with $(C_1, T_1) = (2, 6)$ and $(C_2, T_2) = (3, 9)$ time units, respectively, and one CBS with $(Q, T) = (2, 6)$ time units, yielding a combined utilization factor of 1. Both hard tasks begin their periodic releases at $t = 0$. Three jobs are pushed to the CBS: $J_1(C = 3)$ at instant $t = 2$, $J_2(C = 3)$ at $t = 12$ and $J_3(C = 1)$ at $t = 20$.

For each test case the timer tick resolution was fixed at 100us, and for each task set different values were selected for the time unit multiplier, being 1000ms, 100ms, 10ms and 1ms, to verify possible variations in performance caused by overheads. Figures 2a and 2b present example executions for the two range end cases. The bottom line shows the available budget, with associated events (B_COND: condition for replenishing the budget met, and B_ROUT: budget replenished).

Execution varies with the chosen time unit, mainly due to inaccuracies in `k_busy_wait`, which causes tasks to run slightly shorter than configured at finer granularities (e.g., 10% less at 1 ms) [16]. While this affects actual execution times, it also shifts the timing of CBS events, leading to different scheduling outcomes across configurations.

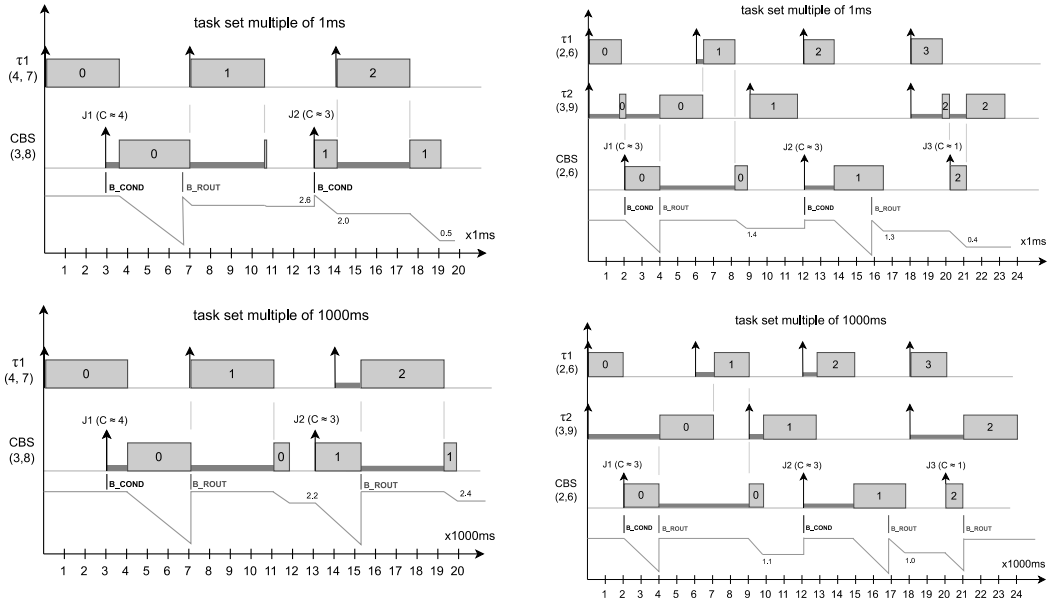
Despite these differences, the CBS mechanism behaves correctly. Changes in execution duration and task start times alter when conditions like budget exhaustion and deadline recalculation occur, sometimes triggering events (e.g., B_ROUT) earlier or not at all. However, in both test cases, the CBS consistently adapts to these variations, preserving its intended semantics even though the resulting schedules differ.

Both tests cases are based on examples provided by one of the CBS authors [7] and are available on GitHub, alongside with the kernel implementation itself, as a sample code [12]. The defined values of C were implemented as calls to the `k_busy_wait()` Zephyr API function, which allows a thread to remain in CPU for the approximate time passed to it. The tracing events were recorded synchronously as each task was triggered, started and finished, but only logged after a predefined amount of time to decrease the overhead.

Application-wise, the main difference between the former implementation and the task-based one is that the CBS emulated the irregular job arrivals by setting a new period at the end of every cycle, whereas the previous version used a timer to trigger the jobs. If desired, one could employ a timer for both solutions, however that would not leverage the new implementation features, which would not make much sense. The task sets were subjected to the same time unit multipliers of $1000ms$, $100ms$, $10ms$ and $1ms$ as before.

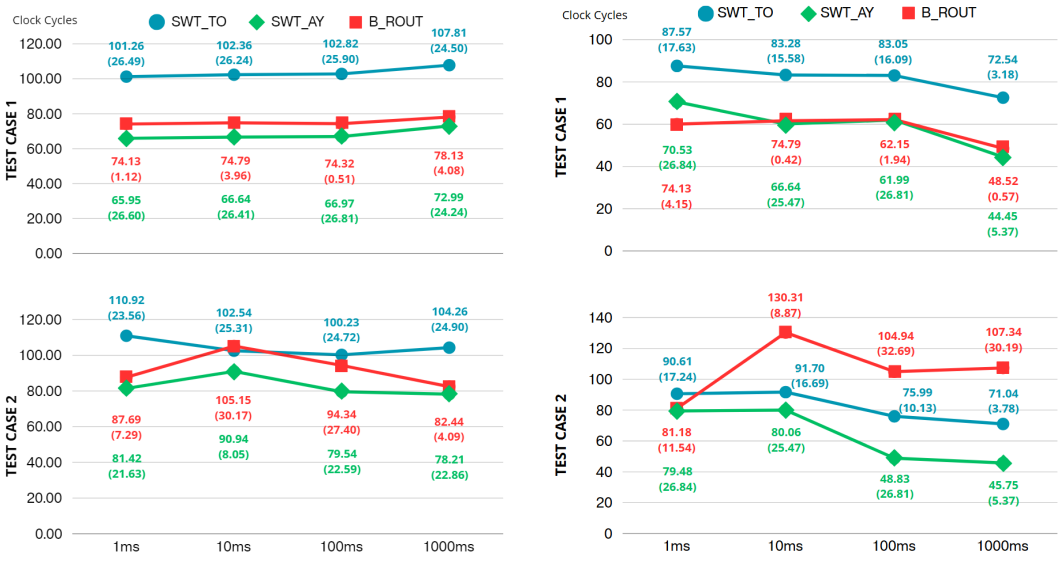
The scheduling outcome (i.e. the order of execution of the tasks), as expected, remained exactly the same as illustrated in Figures 2a and 2b, with the `k_busy_wait` inaccuracies once again driving the only changes between theory and practice (due to actual execution times decreasing as the unit multiplier went down to $1ms$). Thus, the observed behavior equally matched the theoretical one in regards to scheduling events, and the task-based implementation can be considered adequate.

As for the overhead analysis, one could also expect the outcome to be similar as before, since the kernel interactions required for the CBS to function accordingly remained identical between implementations: the CBS timer would still need to be paused on "switched away" (SWT_AY) events, and resumed on "switched to" (SWT_TO) events. The budget replenishing routines also remained the same, leaving no obvious culprits to explain significative



(a) Test Case 1. **(b)** Test Case 2.

Figure 2 Test cases.



(a) Previous results [16]. **(b)** Updated implementation.

Figure 3 Comparison of average overheads and deviations (in clock cycles) between the previous and updated CBS implementations.

differences between the approaches. However, they use different data structures and kernel objects in their implementation which could possibly change the compilation and resource optimization strategies, therefore impacting the overheads.

Figures 3 present a side-by-side comparison between the results reported in [16] and the updated implementation. Overall, the new approach demonstrates improvements in both overhead reduction and temporal consistency. In particular, SWT_TO is reduced by up to 32.7% in Test Case 1, while B_ROUT decreases by as much as 37.9% for larger time units, indicating a more efficient handling of CBS events. Additionally, the updated results exhibit smoother and more predictable trends across different time granularities, suggesting reduced sensitivity to timing artifacts. Although some localized increases remain (e.g., B_ROUT in Test Case 2 at specific time units) these are limited in scope.

Moreover, Table 1 complements the collected data by including the most frequent maximum overheads for each batch of samples (O_f) and the absolute highest values observed throughout all samples (O_{max}). Once again there are noticeable differences when compared to the results for the previous approach (from [16]).

Table 1 Highest measured overheads comparison with previous approach, in clock cycles.

Event	Task-based CBS						Previous approach					
	Test Case 1			Test Case 2			Test Case 1			Test Case 2		
	O_f	O_{max}	n	O_f	O_{max}	n	O_f	O_{max}	n	O_f	O_{max}	n
SWT_TO	107	107	104300	107	107	423700	128	128	55500	128	128	54500
SWT_AY	99	99	104500	99	99	423600	93	133	55500	93	139	54500
B_ROUT	62	261	45000	93	191	223700	75	153	28800	154	154	36700

By inspecting the average overheads, one can see the worst case for switching in to the CPU (SWT_TO) is now of 91.70 clock cycles and happens at the $10ms$ range, whereas for the previous approach it was of 110.92 clock cycles in the $1ms$ range. That is a reduction of 17.33%, and a worst case shifted to a less impactful time unit. A similar story is observed for the `SWT_AY` event, where the time unit remained $10ms$ for both approaches but the task-based CBS fared 80.06 clock cycles, versus the previous 90.94 clock cycles for the version in [16]. That is a reduction of 11.96%.

By comparing the new results (1) with the overheads in [16], it is also noticeable the absolute worst measures have reduced dramatically for the context switching, from 128 and 139 clock cycles in the previous approach to 107 and 99 clock cycles in the task-based CBS for the SWT_TO and SWT_AY events, respectively. Thus, the task-based CBS appears to cause less impact on context switches, and since those are fairly frequent events on multiprogramming environments, it may be considered a significative advantage over the approach in [16] (especially on large task sets).

However, the less frequent budget exhaustion event, B_ROUT, seemed to tell a different story. The average values for Test Case 1 indeed remained lower than 75 clock cycles for all time units, which is a slightly better result than the previous approach for the same conditions, however in Test Case 2 the task-based option yielded considerably higher overheads than the previous approach, for all time units except $1ms$. The worst average measure reached 130.31 clock cycles on $10ms$, and the absolute worst overhead observed throughout all samples (O_{max}) reached 261 clock cycles, or 69% more than the 154 clock cycles for the approach in [16].

Thus, assuming the same circumstance of the three events happening one after the other in their worst case scenario, the maximum overhead associated with the task-based approach would be of 467 clock cycles, or $2.91us$ considering the ESP32C3's 160MHz clock speed.

That is very close to the $2.63us$ of the previous implementation, and still much lower than the tick rate employed for tests (10kHz or $100us$). Performance-wise, this implementation is also considered acceptable.

The performed evaluation allows to conclude that both implementations correctly support bandwidth reservations and follow the expected CBS theoretical model behaviour when employing EDF scheduling. Although differences were observed with respect to their behaviour and overhead, both implementations proved to be effective in supporting aperiodic execution while maintaining temporal isolation. Also, the measured worst-case overhead of CBS operations was below $3us$ for both approaches, demonstrating that both proposed solutions are viable for use in a real-world setting.

It is important to note that the previous approach and the newly-presented task-based approach can be used interchangeably to some extent (for example, the tests conducted make use of the same task sets), however they also pose intrinsic differences that might be considered by system developers when crafting their applications. The previous version requires a background thread and a message queue to function and can be leveraged by interrupts to offload work into a deadline-controlled environment, whereas the new task-based approach requires some extra adaptation to fulfil this usage.

It is also hardly possible to have a task-based CBS running different workloads, as it is exclusive to a single task. A handful of tasks can be set to operate with their own CBS, each with a different job that would otherwise could be pushed to a single CBS in the previous approach. However, for other applications the task-based approach might be better suited, for example when there is a desire to provide bandwidth guarantees for all EDF tasks in the system, and as for resource usage, the task-based strategy might consume less memory and generate less preemptions than the previous approach for the same amount of servers in the application due to no requirement for a message queue or external job dispatchers for each.

In essence, the previous version is recommended for offloading asynchronously-generated work and the new task-based CBS is recommended for establishing additional execution guards on systems with more unpredictable tasks.

6 Conclusion

In this paper, we examine the integration of CBS within the Zephyr RTOS, suggesting a new Task-Based CBS, which involves embedding a private CBS entity within each individual application task. In order to support the proposed approach, a richer EDF implementation for Zephyr is also provided. The new CBS approach provide a different architectural trade-offs with respect to modularity, integration, and control, in relation to previous work. The CBS implementation has been developed within the Zephyr kernel and tested against representative sets of tasks, showing that it correctly supports bandwidth reservations and follows expected CBS theoretical model behaviour when employing EDF scheduling. Although differences were observed with respect to the behaviour and overhead, in relation to the previous approach, the new approach proved to be effective in supporting aperiodic execution while maintaining temporal isolation. The runtime overhead was found to be below $3us$, demonstrating that the proposed solution is viable for use in a real-world setting. As part of future work, an extension of the proposed approaches to multiprocessor systems will be carried out, including support for bandwidth inheritance, thereby extending the applicability of CBS-based scheduling within Zephyr.

References

1 L. Albeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 4–13, 1998.

2 Petr Benes. Porting of resource reservation framework to rtems executive. Master's thesis, Czech Technical University, 2011. Accessed at: April 15th, 2026.

3 Alessandro Biondi, Alessandra Melani, and Marko Bertogna. Hard constant bandwidth server: Comprehensive formulation and critical scenarios. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 29–37, 2014. doi:10.1109/SIES.2014.6871182.

4 Gedare Bloom and Joel Sherrill. Scheduling and thread management with rtems. *SIGBED Rev.*, 11(1):20–25, 2014. doi:10.1145/2597457.2597459.

5 Brandenburg, Björn B. and Gül, Mahircan, and Vanga, Manohar. A tour of litmusrt - reservation types. https://www.litmus-rt.org/tutorial/manual.html#reservationtypes. Accessed on: April 15th, 2026.

6 Giorgio Buttazzo and Enrico Bini. Optimal dimensioning of a constant bandwidth server. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 169–177, 2006. doi:10.1109/RTSS.2006.31.

7 Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Science, 2011.

8 Denis Hoornaert, Golsana Ghaemi, Andrea Bastoni, Renato Mancuso, Marco Caccamo, and Giulio Corradi. Mcti: mixed-criticality task-based isolation. *Real-Time Syst.*, 60(2):328–365, July 2024. doi:10.1007/s11241-024-09425-5.

9 Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline scheduling in the linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016. doi:10.1002/spe.2335.

10 Linux Kernel Development Community. The linux kernel - deadline task scheduling. https://docs.kernel.org/scheduler/sched-deadline.html. Accessed on: April 15th, 2026.

11 L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. Iris: a new reclaiming algorithm for server-based real-time systems. In *Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004.*, pages 211–218, 2004. doi:10.1109/RTTAS.2004.1317266.

12 Alexander Paschoaletto. Decoupled CBS implementation on zephyr. https://github.com/alexpaschoaletto/zephyr/tree/cbs-standalone. Accessed on: April 15th, 2026.

13 Alexander Paschoaletto. Periodic EDF implementation on zephyr. https://github.com/alexpaschoaletto/zephyr/tree/edf-periodic. Accessed on: April 15th, 2026.

14 Alexander Paschoaletto. Task-based CBS implementation on zephyr. https://github.com/alexpaschoaletto/zephyr/tree/edf-periodic-cbs-thread-based. Accessed on: April 15th, 2026.

15 Alexander Paschoaletto. Space is so monotonic: Introducing dynamic schedulers to satellite software. Master's thesis, Instituto Superior de Engenharia do Porto, Rua Dr. António Bernardino de Almeida, 431, October 2025. Available: http://hdl.handle.net/10400.22/30782.

16 Alexander Paschoaletto, Paulo Baltarejo Sousa, Luís Miguel Pinho, and Tiago Carvalho. Supporting soft real-time tasks in zephyr with constant bandwidth servers. In *2025 28th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 1–9, 2025. doi:10.1109/ISORC65339.2025.00017.

17 RTEMS Project. Rtems source files. https://gitlab.rtems.org/rtems/rtos/rtems/-/tree/main?ref_type=heads. Accessed on: April 15th, 2026.

18 Manohar Vanga, Andrea Bastoni, Henrik Theiling, and Björn B. Brandenburg. Supporting low-latency, low-criticality tasks in a certified mixed-criticality os. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, RTNS '17, pages 227–236, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3139258.3139274.

19 Zephyr Project. Zephyr 3.7.99 scheduling documentation. https://docs.zephyrproject.org/latest/kernel/services/scheduling/index.html. Accessed in: April 15th, 2026.
20 Zephyr Project. Zephyr real-time operating system. https://www.zephyrproject.org/. Accessed at: April 15th, 2026.