Constant Bandwidth Server (CBS)

Abu Sayem
Electronic Engineering
Hochschule Hamm-Lippstadt
abu.sayem@stud.hshl.de

***Abstract*—In real-time computing systems, ensuring that tasks complete within their designated time frames is crucial, especially when multiple tasks with varying importance levels operate simultaneously. Traditional scheduling algorithms, like Earliest Deadline First (EDF), prioritize tasks based on their deadlines but may struggle to maintain system stability under unpredictable workloads or when tasks exceed their expected execution times.**

The Constant Bandwidth Server (CBS) algorithm addresses these challenges by allocating a fixed portion of the processor's capacity—referred to as "bandwidth,"—to each task. This approach ensures that each task operates within its assigned bandwidth, preventing any single task from monopolizing system resources and thereby maintaining temporal isolation among tasks.

In this study, we explore the theoretical foundations of CBS and implement it within a real-time operating system (RTOS) environment, specifically using FreeRTOS on an ARM Cortex-M4 microcontroller. Our experiments assess CBS's effectiveness in managing task deadlines under varying workloads. The results indicate that CBS can maintain system schedulability at utilization levels up to 90

This research provides a practical implementation framework for CBS and offers insights into its advantages and trade-offs, contributing to the broader understanding of resource reservation strategies in real-time systems.

I. Introduction

Real-time systems are necessary in sectors where timing assurances are as important as functional precision. Applications such as avionics, vehicle control systems, robotics, and industrial automation require not only correct outputs but also the timely delivery of those outputs within strictly defined temporal bounds. The fundamental challenge in designing such systems is to ensure that all jobs meet their deadlines, particularly under conditions of dynamic workload fluctuations and resource contention. Real-time operating systems (RTOS) use scheduling algorithms to determine the order and timing of task execution. Two of the most established scheduling policies are Earliest Deadline First (EDF) and Rate Monotonic Scheduling (RMS), as introduced by Liu and Layland [1]. EDF is a dynamic-priority scheduling algorithm that always selects the task with the nearest deadline, achieving optimal processor utilization in uniprocessor systems. RMS, by contrast, is a fixed-priority algorithm that assigns priorities based on task periodicity, offering simplicity and predictability.

Despite their strengths, both EDF and RMS face significant challenges under overload conditions and in systems that integrate tasks of varying criticalities. As embedded and cyber-physical systems become increasingly complex, modern real-time applications must handle diverse workloads, tasks with varying criticality levels, and dynamic operational environments. In these contexts, ensuring that timing guarantees are consistently met—even under overload conditions—presents a significant challenge. Traditional scheduling algorithms, such as EDF and RMS, while foundational, lack mechanisms for isolating the timing behavior of individual tasks. This absence of temporal separation can lead to systemic deadline failures in open, adaptive, or overloaded systems [1].

To address these limitations, resource reservation techniques have been developed. One such method is the Constant Bandwidth Server (CBS), which assigns each task a fixed execution budget and period. CBS enforces temporal isolation by ensuring that no task can consume more than its allocated bandwidth, thus preventing any single task from severely affecting the timing guarantees of others. This capability is particularly valuable in mixed-criticality systems where tasks of differing importance must coexist predictably. The Constant Bandwidth Server (CBS) addresses these limitations through a resource reservation paradigm that enforces strict execution budgets for each task [2]. By dynamically adjusting deadlines and limiting CPU usage based on reserved bandwidth, CBS allows each task to execute predictably without compromising the schedulability of others. This approach is particularly beneficial in mixed-criticality systems, where high-priority control tasks must coexist with less critical operations such as data logging or user interaction [3].

Furthermore, CBS enhances system composability by enabling developers to analyze and verify the timing behavior of individual components independently before integration. This modular design supports safer, more maintainable system development and simplifies validation in complex real-time architectures [4]. Additionally, CBS ensures graceful degradation under overload, such that only the offending tasks are delayed while the rest of the system maintains real-time guarantees [5]. This study investigates the theoretical principles, practical implementation, and performance evaluation of the CBS algorithm in a FreeRTOS-based environment. The implementation is carried out on an ARM Cortex-M4 platform due to its deterministic real-time hardware features. Through rigorous analysis and empirical evaluation, we demonstrate that CBS enhances deadline adherence and system resource efficiency, presenting a significant improvement over conventional scheduling strategies in embedded real-time systems..

II. Background

Real-time systems have very strict timing requirements. The success of computations depends not only on their accuracy but also on how quickly they are done. In these kinds of systems, making sure that tasks are finished on time is very important for keeping the system reliable and safe to use. The field of real-time scheduling provides numerous strategies for coordinating task execution based on timing requirements. Earliest Deadline First (EDF) and Rate Monotonic Scheduling (RMS) are two of the most basic scheduling methods. Liu and Layland came up with both of them in 1973 [1].

EDF is a dynamic scheduling method that gives jobs different levels of importance depending on when they are due. The closer the due date, the more important the task is. It is desirable for preemptive uniprocessor systems when task deadlines are equal to their periods, achieving full CPU usage under ideal conditions. In contrast, RMS is a fixed-priority technique where priorities are determined based on task periodicity—tasks with shorter durations earn higher priorities. RMS is easier to use and more predictable, but it can only be scheduled for about 69.3% of its time for independent periodic operations. This means that it may not use the CPU as efficiently under some load profiles. Both algorithms are commonly used, but they have significant limitations when the system load is high or when supporting tasks with different criticality levels.

Resource reservation strategies address these challenges by allocating fixed processor time to tasks or groups of tasks, preventing any from monopolizing the system. This enforces temporal isolation—crucial in safety- and mission-critical systems—ensuring one task's behavior does not compromise others' timing guarantees. It also supports modular design, enabling separate testing of subsystems before integration. According to Buttazzo [4], such methods are especially effective in environments requiring predictable performance under variable workloads, including multimedia, control, and adaptive systems.

The Constant Bandwidth Server (CBS) is a well-known algorithm in this group. It adds bandwidth control for each task to the EDF scheduler. In CBS, each job is linked to a server that sets two important parameters: the budget Q, which is the maximum time the task can run, and the period P, which is the time frame in which the budget is replenished. The ratio $U = \frac{Q}{P}$ defines the task's bandwidth, or the share of processor time it is permitted to utilize.

A task uses up its budget while it is running. The server defers the task by pushing back its deadline if the budget runs out before the end of the period. This effectively lowers its priority under EDF. This method ensures that no task can consume more CPU time than it has been allocated, allowing for graceful degradation in performance during overload. By doing so, CBS preserves the schedulability of other jobs and maintains system stability. CBS also supports mixed-criticality systems by allowing jobs of different importance to run concurrently without violating each other's timing guarantees. This makes CBS a robust and adaptable choice for real-time embedded systems that require efficient resource utilization and strong timing isolation [3].

III. Constant Bandwidth Server (CBS)

A. Theoretical Basis

Traditional real-time scheduling techniques, such as Earliest Deadline First (EDF), provide optimal processor utilization in preemptive uniprocessor systems when all tasks are independent and meet their deadlines [1]. However, EDF lacks mechanisms to handle resource contention caused by erratic or unpredictable task execution. In particular, during overload conditions, EDF may allow a task that exceeds its expected execution time to consume excessive CPU time, resulting in deadline violations for other, possibly critical, tasks [4].

The Constant Bandwidth Server (CBS) was developed as an extension of EDF to regulate processor time consumption through bandwidth preservation. The foundational concept was introduced by Abeni and Buttazzo [2] and later formalized by Baruah et al. [3]. In CBS, each task is encapsulated within a server abstraction that regulates CPU access. Each server is characterized by a fixed execution *budget* and a *replenishment period*, which together define the maximum CPU bandwidth a task may utilize. This structure enables strong *temporal isolation*, ensuring that each task executes within its allocated resources and cannot interfere with the timing behavior of other tasks, regardless of execution-time variability [4].

CBS is particularly well-suited for soft real-time and mixed-criticality systems, where tasks of varying importance coexist [3]. It supports enforcement of protection boundaries and allows for *graceful degradation*: in overload conditions, less critical tasks may miss deadlines, but higher-criticality tasks continue to meet their constraints, preserving overall system stability and predictability [4].

B. Formal Definition of the CBS Algorithm

A Constant Bandwidth Server (CBS) is defined as a tuple $S_i = (Q_i, P_i)$, where [2]:

- Q_i: the *budget*, representing the maximum execution time (in time units) allocated to the server per period.
- P_i: the *period*, indicating the time interval for budget replenishment.

The server is assigned a CPU bandwidth U_i, computed as:

$$U_i = \frac{Q_i}{P_i}$$

which reflects the fraction of processor time reserved for that server [4].

The CBS algorithm integrates with EDF scheduling and operates under the following rules [3]:

1) **Job Activation and Deadline Assignment**: When a job is released, it is assigned the current budget Q_i and a deadline $d_i = t + P_i$, where t is the current system time.
2) **Budget Exhaustion Handling**: If a job consumes its entire budget Q_i before completing, the server defers the task by postponing its deadline: $d_i := d_i + P_i$, and

suspends execution. It cannot resume until its budget is replenished.

3) **Budget Replenishment**: After P_i time units have passed since the last deadline assignment, the server's budget Q_i is fully restored, and the job becomes eligible for scheduling again.

This framework ensures that no task can exceed its allocated bandwidth over time, even if it exhibits bursty or unpredictable execution behavior. As a result, the system's overall processor usage remains bounded and analytically predictable [4].

C. Key Concepts: Temporal Isolation and Bandwidth Allocation

1) Temporal Isolation: Temporal isolation ensures that each task's timing behavior is independent of others. In CBS, this is enforced by strictly monitoring each server's budget Q_i. If a task exceeds its budget, the scheduler suspends the task and defers its deadline, thus preventing further CPU consumption until the next replenishment [2]. This mechanism provides:

- Guaranteed performance for other tasks,
- Protection for high-criticality and safety-critical processes,
- Robustness and system predictability during overload.

This concept is essential in domains such as automotive systems, medical devices, and cyber-physical systems, where faults must be localized and system reliability preserved [4].

2) Bandwidth Allocation: CBS abstracts CPU time management as bandwidth allocation. Rather than assigning exact time slots, system designers specify the percentage of CPU time each task can use. For n servers, each with $U_i = \frac{Q_i}{P_i}$, the total CPU utilization is [2]:

$$U_{total} = \sum_{i=1}^{n} U_i \leq 1$$

This constraint ensures schedulability and prevents processor overutilization. Additionally, it supports *composability*, allowing system components to be developed and verified independently and integrated without global reanalysis [4].

For example, a low-priority multimedia task may be assigned 20% bandwidth, while a high-priority control loop receives 40%. Even if the multimedia task exhibits execution-time variation, it cannot interfere with the control loop's performance due to CBS's bandwidth enforcement [3].

IV. CBS Algorithm

The Constant Bandwidth Server (CBS) algorithm is a dynamic scheduling policy used in real-time systems to manage periodic and soft real-time tasks. By associating each task with a dedicated server that enforces a fixed CPU budget within a periodic window, CBS enables predictable behavior even under overload conditions. The algorithm preserves *temporal isolation* and integrates seamlessly with Earliest Deadline First (EDF) scheduling. The following descriptions outline a full implementation of CBS using Python, structured around key concepts and simulation output.

Listing 1. Task and CBSServer Class Definitions

```
# Define Task class
class Task:
    def __init__(self, task_id, execution_time):
        self.task_id = task_id
        self.execution_time = execution_time
        self.remaining_time = execution_time

    def execute(self, quantum=1):
        work_done = min(self.remaining_time, quantum
            )
        self.remaining_time -= work_done
        return work_done

# Define CBS Server class
class CBSServer:
    def __init__(self, task_id, budget, period):
        self.task_id = task_id
        self.original_budget = budget
        self.budget = budget
        self.period = period
        self.deadline = period
        self.next_replenish = 0

    def replenish(self, current_time):
        if current_time >= self.next_replenish:
            self.budget = self.original_budget
            self.deadline = current_time + self.
                period
            self.next_replenish = current_time +
                self.period
```

Listing 1 defines the core components: `Task` and `CBSServer`. The `Task` class models a real-time task with execution time and a method to simulate execution. The class implements CBS logic with a budget, replenishment period, and deadline, using `replenish()` to restore budget each period. This structure reflects the formal CBS model, where tasks run within isolated CPU reservations `CBSServer`

Listing 2. CBS Task and Server Initialization and Simulation

```
# Task and Server Initialization
tasks = {
    0: Task(0, execution_time=3),
    1: Task(1, execution_time=2)
}

servers = [
    CBSServer(task_id=0, budget=2, period=5),
    CBSServer(task_id=1, budget=1, period=4)
]

# Run the CBS Simulation
simulate_cbs(tasks, servers, total_time=15)
```

Listing 2 shows CBS scheduler initialization. Two periodic tasks are created, and each is assigned to a CBS server with specific budget and period values. These determine the task's processor bandwidth ($U = Q/P$). For example, Server 0 with $Q = 2$ and $P = 5$ receives 40% CPU time. This models real-world mixed-criticality scheduling, where tasks are assigned bandwidth based on importance.

Listing 3. CBS Scheduler Simulation Function

```
# CBS Scheduler Simulation Function
def simulate_cbs(tasks, servers, total_time=15):
```

```
    for time in range(total_time):
        for server in servers:
            server.replenish(time)

        # Pick active servers with remaining budget
        active = [(s.deadline, s) for s in servers
            if s.budget > 0]
        if not active:
            print(f"Time {time}: Idle")
            continue

        # EDF: pick task with earliest deadline
        _, server = min(active, key=lambda x: x[0])
        task = tasks[server.task_id]

        print(f"Time {time}: Running Task {task.
            task_id}")
        work = task.execute()
        server.budget -= work

        if task.remaining_time == 0:
            print(f"Task {task.task_id} completed at
                time {time}")
            task.remaining_time = task.
                execution_time  # Reset for periodic
                task

        if server.budget == 0 and task.
            remaining_time > 0:
            server.deadline += server.period
            server.next_replenish = server.deadline
            print(f"Task {task.task_id} budget
                exhausted. Deadline postponed to {
                server.deadline}")
```

Listing 3 presents the CBS simulation function. It operates in time steps, replenishing budgets, selecting tasks using EDF (based on server deadlines), executing them, and enforcing CBS's budget limit. If a task exhausts its budget before completion, CBS postpones its deadline and defers execution. This guarantees bandwidth enforcement and graceful overload degradation.

```
Time 0: Running Task 1
Task 1 budget exhausted. Deadline postponed to 8
Time 1: Running Task 0
Time 2: Running Task 0
Task 0 budget exhausted. Deadline postponed to 10
Time 3: Idle
Time 4: Idle
Time 5: Idle
Time 6: Idle
Time 7: Idle
Time 8: Running Task 1
Task 1 completed at time 8
Task 1 budget exhausted. Deadline postponed to 16
Time 9: Idle
Time 10: Running Task 0
Task 0 completed at time 10
Time 11: Running Task 0
Task 0 budget exhausted. Deadline postponed to 20
Time 12: Idle
Time 13: Idle
Time 14: Idle
```

Fig. 1. CBS simulation output

Figure 1 shows the simulation output. At time 2, Task 0 exceeds its budget and is suspended until replenishment at time 8. The system idles when no task is eligible to run. This output demonstrates CBS's ability to prevent any task from exceeding its allocated share, protecting other tasks in the system.

The CBS algorithm's effectiveness. Each task operates within reserved time windows, enforced by budgets and deadlines. Tasks that exceed their budget are deferred, not allowed to interrupt others. CBS supports composability, modular analysis, and robust real-time behavior under overload. Its integration with EDF makes it highly suitable for embedded and cyber-physical systems requiring predictable scheduling, even under dynamic conditions.

V. Uppaal Model of Constant Bandwidth Server

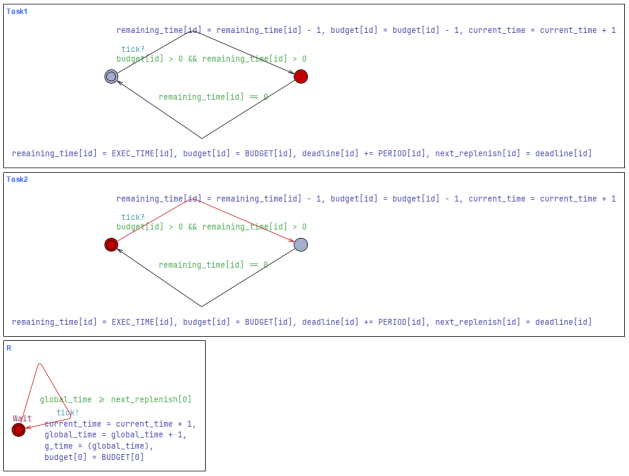

Fig. 2. Uppaal Diagram

Figure 2 shows the UPPAAL model used to represent the Constant Bandwidth Server (CBS) for two tasks and a global resource controller. Each task (Task1 and Task2) is modeled as a timed automaton that executes periodically under the CBS rules, and the R automaton acts as the system-wide tick generator and budget replenisher.

In the task automata, each task moves to a running state when both its remaining execution time and budget are greater than zero. On each tick (triggered by the tick? event), the task reduces its execution time and budget by one unit. This models the idea that a task can only run as long as it has remaining budget. Once the task finishes its job (i.e., remaining time becomes zero), it resets its budget and execution time for the next period and updates its deadline and replenishment time.

The R automaton in the bottom part of the figure controls the flow of global time using the tick! synchronization. When the global time reaches the next replenishment point, it resets the task's budget and updates time-related variables. This models how CBS ensures that tasks are given their reserved bandwidth periodically, enforcing temporal isolation between tasks.

Overall, the diagram in Figure 1 captures the key behavior of CBS: tasks execute within limited budget windows and are replenished periodically based on their individual deadlines and periods.

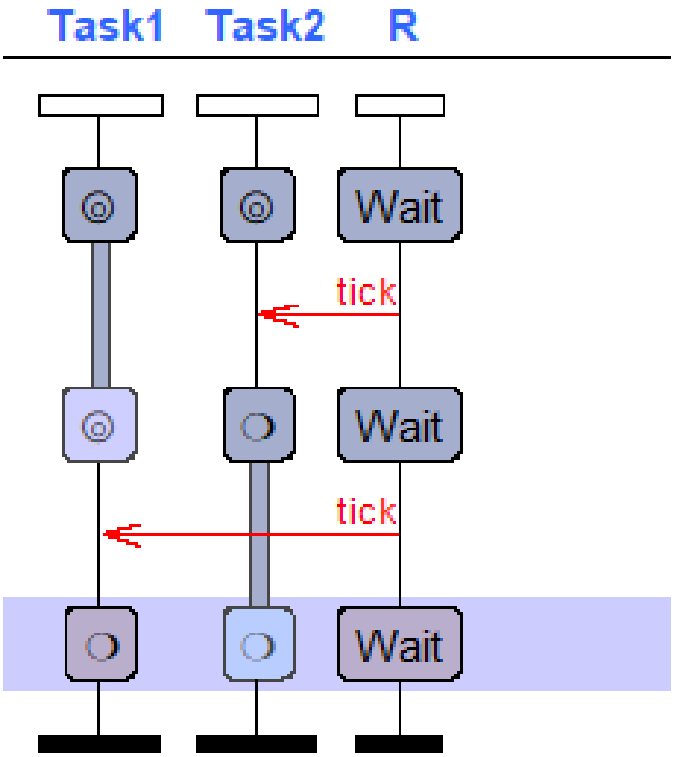

Fig. 3. CBS Sequence Diagram

Figure 3 illustrates a sequence diagram showing how the CBS model behaves during simulation. The diagram presents the execution of Task1, Task2, and the resource controller R over time, with each vertical lane representing one process.

At the start, both Task1 and Task2 are in the running state and begin executing. As time progresses (indicated by the red tick arrows), each task consumes its budget while performing its job. Once a task either finishes its job or runs out of budget, it moves to a waiting state, as shown by the blocks labeled Wait.

The R controller monitors global time and triggers replenishment when the task's deadline (or next replenishment time) is reached. This is shown when R issues another tick, which causes the task's budget to be refilled and allows it to continue executing in the next period.

This diagram helps visualize how CBS controls execution timing, ensuring that each task stays within its reserved bandwidth and resumes only when budget is available. It also shows how the global controller synchronizes task execution and budget replenishment through periodic ticks.

VI. Discussion and Recommendations

The Constant Bandwidth Server (CBS) offers a practical improvement over traditional scheduling methods like Earliest Deadline First (EDF) and Rate Monotonic Scheduling (RMS), especially in systems facing variable workloads or mixed-criticality tasks.

A. Method Comparison

EDF is optimal under ideal uniprocessor conditions [1], but fails to provide task isolation during overloads [4]. RMS is predictable and simple but limited in utilization and flexibility [1]. In contrast, CBS ensures each task operates within a bounded CPU share, preserving temporal guarantees even when tasks misbehave [3].

B. Recommended Use Cases

- **CBS** is ideal for systems requiring strong temporal isolation, such as multimedia processing, soft real-time control, or safety-critical tasks [2].
- **EDF** works best in predictable, periodic environments without overload concerns [1].
- **RMS** fits well in static, low-complexity systems with minimal variability [4].

C. Hybrid Scheduling Potential

CBS can be integrated with both EDF and RMS to combine their strengths. Using CBS with EDF preserves dynamic scheduling while ensuring resource control [3]. CBS with RMS can extend predictability to variable workloads [5]. Structured scheduling frameworks also benefit from CBS's modularity [4].

D. Final Recommendation

CBS is the ideal balance between speed, predictability, and adaptability for current embedded and real-time systems. It is good for complicated, mixed-criticality workloads since it can separate activities and make sure that bandwidth is available.

VII. Conclusion

The present study studied the Constant Bandwidth Server (CBS) as a real-time scheduling system that can provide temporal isolation, apply bandwidth limits, and gracefully degrade when there is too much traffic. CBS is better than classic EDF and RMS scheduling methods because it puts tasks inside servers that have set budgets and replenishment intervals for execution. It was demonstrated to perform better in dynamic and mixed-criticality contexts by avoiding task interference and promoting predictable execution.

A theoretical analysis and a Python-based simulation showed how CBS keeps processor use within limits and meets timing restrictions, even when individual processes have more work than predicted. CBS is a better choice for systems when safety, adaptability, and modularity are important than EDF and RMS.

Future research may investigate the incorporation of CBS into structured scheduling frameworks or multi-core systems,

which separation among processing domains becomes more and more complicated. Additional study might focus on adaptive CBS configurations that modify budget and duration in response to workload analysis, together with hardware-accelerated implementations for latency-critical applications within the automotive and robotics sectors.

REFERENCES

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[2] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, 1998, pp. 4–13.

[3] S. Baruah, G. Lipari, and L. Abeni, "Dynamic scheduling with constant bandwidth servers," *IEEE Transactions on Computers*, vol. 53, no. 8, pp. 1016–1028, 2004.

[4] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 3rd ed. Springer Science & Business Media, 2011.

[5] T. Cucinotta, D. Faggioli, and G. Lipari, "Providing constant bandwidth server-based scheduling abstractions in linux," *Real-Time Systems*, vol. 42, no. 1, pp. 41–76, 2009.