DESIGN AND IMPLEMENTATION OF A REAL-TIME RATE-BASED TASK SCHEDULER FOR REAL-TIME OPERATING SYSTEMS: A CASE STUDY WITH VXWORKS

Tom Springer[1]and Peiyi Zhao[2]

[1]Fowler School of Engineering, Chapman University, Orange, CA., USA
[2]Fowler School of Engineering, Chapman University, CA., USA

ABSTRACT

This paper details the implementation of a rate-based task scheduler into the VxWorks real-time operating system, intendedto enhance resource allocation for distributed real-time systems, such as IoT and embedded devices. Rate-based scheduling dynamically adjusts task execution rates based on system demand, providing a flexible and efficient approach to meeting real-time constraints. The scheduler was integrated into VxWorks and evaluated using the Cheddar scheduling analysis tool and the VxWorks VxSim simulator. Initial results demonstrate improved deadline adherence and resource management under varying loads compared to traditional schedulers. Future work includes porting the scheduler to single-board computers to assess its performance on resource-constrained IoT hardware and extending it to support resource sharing between tasks to address real-time coordination challenges. This research emphasizes the potential of rate-based scheduling for IoT applications, offering a scalable solution for managing the complexity of distributed, real-time environments in future embedded systems.

KEYWORDS

Real-Time systems, Networked Embedded Systems, Real-Time Operating Systems, Internet of Things Applications.

1. INTRODUCTION

Many applications used in aerospace, automotive systems, industrial control, and embedded devices rely on a real-time operating system (RTOS) to manage the strict deadlines that can mean the difference between success and failure. Task scheduling is a critical component of the RTOS as it ensures that tasks are executed within specific time constraints, providing system determinism. Static priority scheduling is a common approach used in real-time embedded operating systems to manage tasks and meet timing constraints. In static priority scheduling, each task is assigned a priority level, and tasks with higher priority levels are given preference in execution over tasks with lower priority levels. Static priority scheduling in real-time systems offers several advantages, making it a popular choice for real-time applications.

One of the primary benefits of static priority scheduling is its predictability. The execution order of tasks is known in advance, allowing for better analysis and guaranteeing that tasks with a higher priority will be scheduled before those with a lower priority. This predictability is crucial for meeting strict timing constraints in real-time systems. Static priority scheduling also provides

David C. Wyld et al. (Eds): IoTE, CNDC, DSA, AIAA, NLPTA, DPPR - 2024
pp. 01-20, 2024. - CS & IT - CSCP 2024 DOI: 10.5121/csit.2024.142301

deterministic behavior, meaning that the system's response time and task execution order are known and repeatable. Deterministic behavior is another benefit that is essential in real-time systems, where meeting deadlines is often critical for the system's correct operation. Additionally, static priority scheduling is simple to implement, tends to have lower overhead costs, and simplifies system behavior analysis. For these and other reasons, virtually all commercial real-time operating systems support static priority scheduling.

While static priority scheduling has advantages, it is not ideally suited for some real-time systems. For instance, distributed real-time systems and signal processing applications have shown that they do not adequately fit into the traditional static priority model due in part to the amount of jitter in the invocation time of the real-time task [1]. The issue is that the priority value does not always map directly to significant task constraints; another issue is addressing the problem of dealing with unknown or varied task execution times. Additional shortcomings include graceful performance degradation during processor overload and ensuring full resource utilization in tightly constrained embedded systems. For example, critical embedded systems are considered unsafe if resource utilization exceeds 50%, requiring system designers not to exceed this threshold. This utilization threshold mandates that processing elements meet the required computing demands, causing an increase in the system's overall size, weight, and power (SWaP). Therefore, the standard design goal in resource-constrained devices is to reduce the SWaP footprint to reduce costs and ensure that the systems can be integrated into compact, portable, and power-constrained environments.

To solve these and other limitations, we are investigating the implementation of rate-based resource allocation methods into an RTOS for use in distributed real-time systems. According to authors [2-5], the rate-based allocation method more naturally models the actual implementation of distributed real-time embedded systems. In a rate-based system, a task specifies its requested progress rate related to the number of computed events within a specified time interval. Examples of rate specifications in a distributed system could include processing *n* packets per second or *m* video frames per second. According to the rate-based approach, a task is guaranteed to progress at its specified rate when adequate resources are available. On the other hand, if adequate resources are unavailable for a task to make the desired progress, the scheduler will either negotiate with the RTOS for a reduced rate of progress or require the task to wait for sufficient resources to become available. This reduction in resource allocation is done by either the user explicitly specifying a reduced rate of progress or the RTOS reducing the task resource requirements when the system identifies that the process is using fewer resources than initially requested.

For this work, we implemented a rate-based scheduling mechanism into VxWorks, a widely used RTOS known for its robustness and reliability. It is used extensively in various industries, including aerospace, automotive, and telecommunications. VxWorks is highly customizable and has multiple development tools, making implementing and managing a complex scheduling algorithm easier. This customization is highly beneficial for implementing a new scheduling algorithm as it allows for fine-tuning task priorities and scheduling policies.

This paper describes the motivation for using a rate-based scheduling mechanism in a real-time system, the implementation in the VxWorks RTOS and key performance outcomes. The outline of the paper is as follows: Section 2 covers the related work including gaps in current solutions, Section 3 describes the design and integration of the rate-based scheduler, Section 4 details the implementation into VxWorks, Section 5 provides details on the experimental setup and testing of the scheduling mechanism, Lastly, Section 6 summarizes the paper including a short discussion on future work.

2. BACKGROUND AND RELATED WORK

Traditional static priority scheduling techniques in an RTOS are based on the concept of a discrete periodic or sporadic task. The task is scheduled such that it completes execution before a predefined deadline. Most real-time execution models are based upon the periodic task model [6], defined by Liu and Layland, or the sporadic task model [7], defined by Mok. In both models, the execution of the task must be finished before the next instance of the task is scheduled for execution. The challenge is that a priority value must be assigned to each task so that each execution instance is completed before their respective deadlines. The most common priority assignment policies include rate-monotonic (RM) scheduling [6], where tasks that run at a higher rate have a higher priority over tasks that run at a lower rate, and the deadline monotonic (DM) policy [8], where tasks are assigned priorities according to their respective deadlines. Using the DM policy, the task with the shortest deadline is assigned the highest priority.

With RM or DM scheduling policies, mapping deadlines to priorities can be straight-forward for tasks with explicit timing constraints. However, it becomes challenging for complex tasks such as a video processing system whose requirement is to receive frames from a remote device at *n* frames per second. While the overall task of processing *n* frames per second has precise timing requirements, it is uncertain how to prioritize the receiving of each individual packet since a single frame is made up of multiple network packets. The problem is that prioritizing a task is based on assigning a value for the network processing. However, since there is no inherent value for this processing a more conservative (i.e. shorter response time) priority value is often selected resulting in reserving more computing resources than may be necessary.

Another issue with static priority-based resource allocation is that assumptions about the environment where the tasks are executed must be made. Specifically, the resources required for task execution, such as processor time, must be known *apriori*. Issues arise if the actual task execution time exceeds the amount that was initially estimated. The problem is that these“misbehaving” tasks can block all lower-priority tasks indefinitely. Static priority scheduling lacks the mechanisms to isolate the misbehaved tasks from disrupting the execution of lower priority well-behaved tasks.

An extension of the task isolation problem mentioned previously is the issue of ensuring graceful performance degradation during periods of overload where a set of misbehaving tasks causes increased compute processing and missed deadlines. In these task overload situations, managing graceful task performance degradation is essential. Unfortunately, under overload conditions, static priority scheduling only allows one form of degradation where higher priority tasks are usually executed. In comparison, lower-priority tasks are executed at a reduced rate or not at all.

2.1. Rate-Based Resource Allocation Models

The concept of rate-based resource allocation is nothing new and was initially designed to support multimedia computing along with other soft real-time applications. Researchers [9] have identified three broad classes of rate-based allocation models: server-based allocation, fluid-flow allocation, and rate-based generalization of the periodic model.

2.1.1. Server-Based Allocation

The sever-based allocation model was derived to manage the problem of scheduling aperiodic tasks in a real-time system. The idea of server-based allocation is based on the creation of a server process invoked periodically to service any aperiodic requests that may have arrived.

Associated with the server is the server's execution time capacity, which gets depleted as the aperiodic task is executed. The aperiodic task is suspended once the capacity has been exhausted until the next server innovation time. The effect is that aperiodic tasks can be modelled as periodic tasks, ensuring aperiodic processing progresses at a well-defined and uniform rate. Server-based algorithms are considered to be rate-based allocations because the execution of the server is not directly connected to the arrival of an aperiodic task. Several server algorithms are described in the literature [10,11,12], with the most familiar being the constant bandwidth server (CBS), which has been included in the Linux kernel since version 3.14. It is called the SCHED_DEADLINE scheduler.

2.1.2. Fluid-Flow Allocation

Fluid-flow allocation models, rooted in bandwidth allocation schemes used by the networking community, are concerned with allocating the processor in a fair manner, with each process receiving its fair share of the processor. Algorithms such as the generalized processor share (GPS) [13] have been instrumental in this area, leading to the development of numerous fair fluid-based allocation algorithms [14,15,16] that form the foundation for building real-time services.

2.1.3. Rate-Based Allocation

A rate-based generalization of Liu and Layland's periodic task model have been developed to allow more flexibility in how a scheduler responds to events that arrive at a uniform average rate. Examples include the (*m,k)* allocation model that requires only *m* out of every *k* events be processed [17] in real-time; the window-based allocation (DWYQ) method, which ensures a minimum number of events are processed in real-time within the context of a sliding window [18]; and the rate-based execution (RBE) model [1], which adjusts the priorities of events that arrive at higher than expected rates. For this work we leverage the RBE model which defines the rate specification of a task as *(x, y, d*) where each task is ensured to process at least *x* events every *y* time unit and each event j will be processed before the deadline d. The specific deadline for processing the j^{th}event for taskt_i is determined by the following recurrence:

$$D_i(j) = \begin{cases} t_{i,j} + d_i if\ 1 \leq i \leq x_i \\ \max\left(t_{i,j} + d_i D_i(j - x_i) + y_i\right) if j > x_i \end{cases} \quad (1)$$

Under this function, task requests that arrive at a faster rate than *x* arrivals every *y* time unit will have their deadlines postponed until the time they would have been assigned had they arrived at the rate of exactly *x* arrivals every *y* time units [1].

2.2. Alternative Real-Time Scheduling Approach

Rate-based allocation models, like RBE, has received significant attention in the research community but has yet to see adoption into commercially available RTOSs. While time-based scheduling algorithms, like round-robin or pre-emptive priority scheduling, are well-established the actual implementation of rate-based schedulers remains confined to experimental kernels and are not incorporated into commercial RTOS deployments. Research prototypes typically focus on multimedia or network traffic management applications but often lack the robustness needed for widespread adoption.Commercial RTOS platforms like VxWorks scheduling models are well-understood, easy to understand and effective for many real-time applications. However, at this time there is no rate-based scheduler that has been fully integrated into the RTOS kernel. Some specialized systems that focus on media applications (e.g. video pro-cessing or

telecommunications) have adopted versions of rate-based scheduling concepts to manage the real-time demands of the application. Though, these systems are custom-built for specific applications and not designed for platforms like VxWorks. Additionally, certain Linux-based real-time multimedia systems use rate-based scheduling, but these systems are not equivalent to the timing guarantees provided in hard real-time systems.

Although rate-based scheduling has not been incorporated into commercial RTOS platforms, there is a growing interest for dynamic and adaptive scheduling mechanisms related to the increase in complex applications such as the Internet-of-Things (IoT) or autonomous systems. These applications often require schedulers that respond dynamically to task execution changes, which could provide the motivation for the widespread adoption of rate-based models. Some operating systems are starting to investigate mixing scheduling models that combine traditional fixed-priority mechanisms with rate-based scheduling. This hybrid approach could serve as a bridge toward integrating more flexible scheduling schemes into commercial RTOS systems. Our work contributes to bridging this gap by in incorporating a rate-based scheduler into a traditional RTOS environment and providing a practical demonstration of how rate-based scheduling can coexist with more traditional methods, potentially encouraging further exploration in this direction. This work stands out because it provides a real-world prototype that could be extended and optimized for broader adoption in various embedded and IoT applications.

3. RATE-BASED TASK SCHEDULER DESIGN

In rate-based scheduling, the focus is not strictly on task priority but rather on ensuring that tasks are scheduled at regular intervals, according to their rates. This aligns closely with concepts of time-triggered scheduling, where the timing behavior of the system is predetermined and controlled according to periodic task executions. Rate-based scheduling combines the theoretical principles of periodic scheduling with practical real-time constraints, ensuring tasks are executed according to their temporal requirements in embedded and time-critical applications.

3.1. Rate-Based Allocation Model

The rate-based allocation model is based on the concepts defined in real-time scheduling theory, specifically RM and DM scheduling policies, but with some modifications to focus more on the rate and frequency of a task as opposed to the priority. In the rate-based model a real-time system is defined as a set of tasks$\{\tau_1, \tau_2, \ldots \tau_n\}$where each task consists of infinite or finite instances of jobs or requests which must be completed by the deadline (depending upon the criticality of the task). Each real-time task is defined as either hard real-time (HRT) or soft real-time (SRT). Each task τ_i is characterized by its worst-case execution time C_i, relative deadline D_i, period P_i, and the resource rate of the task $U_i = C_i/P_i$. How tasks are allocated is defined by the task type, HRT or SRT, and constrained by the overall system utilization where:

$$U = \sum_{i=1}^{n} C_i/P_i \quad (2)$$

A third aperiodic non-real-time task (NRT) type can also be defined for resource allocation based on reserving a minimum amount of utilization for the NRT tasks. Allocation for HRT tasks are guaranteed rates that are equal to the task's required rate. SRT task allocation is based on the system load. If the system is underloaded ($U \leq 1$) then the SRT task is assigned to its requested rate. If the system is overloaded ($U > 1$) then an SRT task may be assigned resources that is less than what is requested. Allocation of SRT tasks involves employing a weighted-proportional resource allocation approach that is used to assign SRT tasks to available resources. For NRT

tasks a minimum resource utilization is calculated which is proportional to the number of NRT tasks. The formula [1] to calculate NRT utilization is provided below:

$$\beta = \frac{NRT_n}{NRT_n + SRT_n \times S \times (1+U_{srt})^2} \; x \; (1 - \; U_{srt}) \tag{3}$$

Where NRT_ndefines the number of NRT tasks, SRT_n represents the number of SRT tasks, S indicates the scale factor (based on the relative importance of the SRT and NRT tasks in the system), U_{SRT} is the resource rate (utilization) of all SRT tasks in the system and $U_{HRT}$is the resource rate of all HRT tasks in the system. For SRT tasks the actual utilization is defined as (4) and the actual NRT utilization is defined as (5).

$$U'_{SRT} = min\{U_{SRT}, 1 - \; U_{HRT} - \; \beta\} \tag{4}$$

$$U'_{NRT} = 1 - \; U_{HRT} - \; U'_{SRT} \tag{5}$$

It is important to note that SRT task starvation is possible if the system is overloaded. To reduce the potential for SRT task starvation a proportional allocation policy [19] can be adopted where each SRT task would receive a percentage of available resources that is proportional to its desired resource rate. In this way the actual resource rate of each SRT task would be lowered so that total system utilization is U ≤ 1. However, this approach does not guarantee that any SRT task will actually meet their initial deadlines since none will receive their requested rates. For this work all the SRT tasks have the same priority to their required resources so resources are allocated using a proportional allocation policy where each SRT task receives a utilization amount proportional to their desired rate. The proportional allocation formula [1] is provided below:

$$wU_{SRT_i} = \frac{U_{SRT_i}}{\sum_{i=1}^{SRT_n}(U_{SRT_i})} \times U'_{SRT} \tag{6}$$

The resource allocation policy would be invoked when a new HRT or SRT is scheduled or when an HRT or SRT task completes execution. Changes to NRT task utilization β will also invoke changes to the SRT resource rate.

3.2. Rate-Based Scheduling

The rate-based task set is scheduled by using the earliest deadline first (EDF) algorithm with additional support for dynamically adjusting the task's rate or period, known as a mode change. While EDF is based on proven real-time scheduling policies where task utilization and period are fixed it has been shown that under certain conditions EDF can still guarantee deadlines even when there are dynamic mode changes. The EDF algorithm defines a feasible schedule as $U \; \leq 1$ which can support arbitrary period changes considering the utilization of a departing task can be thought of as $1 - \; u_i$ when the task deadline is reached. Therefore, a modified task with utilization factor $\leq (1 - \; u_i)$ can also be considered schedulable.

More specifically a task that has gone through as mode change has either increased its rate and/or period or decreased its rate and/or period. The question is how a feasible EDF schedule can still be maintained while dynamically adjusting the task constraints? Researchers who developed similar models have provided proofs to the correctness of the EDF when adjusting task constraints. Readers are encouraged to review the detailed proofs [20, 21] for additional information

3.2.1. Rate-Based Mode Change

When the system becomes overloaded this invokes a mode change by reducing the resource rate of SRT tasks so that total utilization is $U \leq 1$. The rate-based scheduling mechanism adjusts the relative deadline of all SRT tasks based upon the weighted resource allocation policy. The goal is to extend the relative deadline of each SRT task while leaving WCET unchanged. There is a case when a task WCET will need to change based upon a user's inaccurate estimation of the WCET. By progressively extending the deadline and adjusting the WCET of a task such that $U \leq 1$ by definition a feasible schedule can be obtained. The effect of increasing or decreasing the rate of a task is shown in Figure 1. At time *t*, the utilization of a task increase from *u* to *u'* and subsequently the effect of decreasing the rate of a taskis shown at time *t'*. The value of *d* represents the initial deadline while *d'* represents the extended deadline and *d'n* defines a calculated compressed deadline for the next deadline while *d''* defines the subsequent deadline for the next period. Additionally, *p* represents the initial task period where *p'*or *p'n*represent the calculated and next period similar to how the deadlines are calculated.

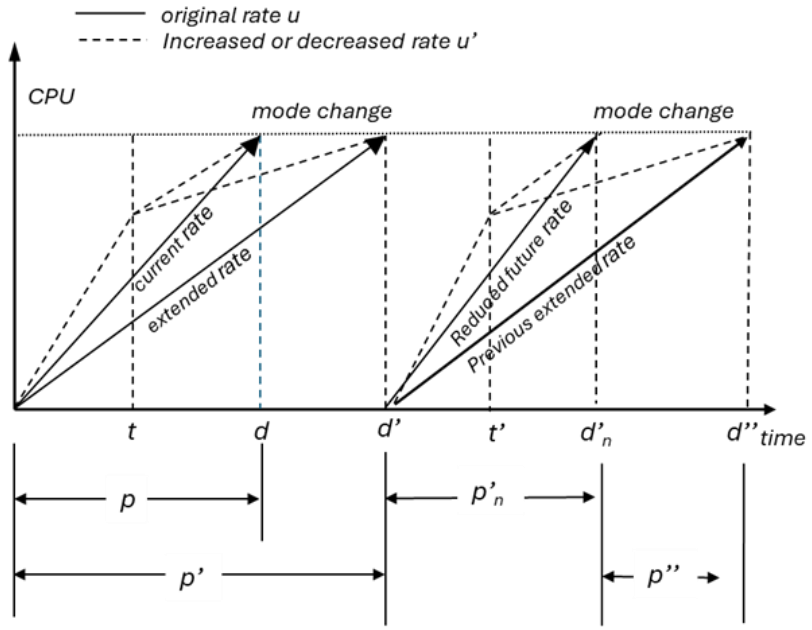

Figure 1: Task Rate Utilization

Mode changes triggers the weighted proportional allocation method mentioned above to ensure no deadlines are missed after the mode change. To facilitate a mode change, the rate-based scheduler extends the previous task rate. The extended period is calculated as follows:

$$P'_{SRT_i} = \frac{C_{SRT_i}}{wU_{SRT_i}} \tag{7}$$

here P'_{SRT_i}defines the new period, C_{SRT} defines the *wcet* for a SRT task and $wU_{SRT}$the new task rate for the SRT task. Extending the task period also requires extending the next deadline of the SRT task. The extended SRT deadline is calculated below:

$$D'_{SRT_i} = D_{SRT_i} + P'_{SRT_i} \tag{8}$$

where D'_{SRT_i}defines the new calculated period. To account for tasks that still need to be scheduled in the current period they also need to consider the period changes during subsequent deadlines. In the case where idle time is generated for an SRT task that is still using the previous deadline at its release time (which is before the current release time), the following mechanism is applied:

$$D'_{SRT_i} = \max\{t + C_{SRT_i}, D_{SRT_i} + P'_{SRT_t}\} \quad (9)$$

$$P_{SRT_i} = D_{SRT_i} - P'_{SRT_i} \quad (10)$$

In the case when there is no idle time, but tasks experience deadline jitter; the previous period, then a temporary one is factored in before the newly extended period is calculated. The following formula calculates the period starting from the current time but not the release time.

$$D'_{SRT_i} = \max\{t + \frac{C_{SRT_i}}{wU_{SRT_i}}, D_{SRT_i} + P_{SRT_t}\} \quad (11)$$

$$P'_{SRT_i} = D'_{SRT_i} - t \quad (12)$$

3.2.2. Rate-Based Scheduling for Non-Real-Time Tasks

While HRT and SRT have deadlines NRT tasks are not considered to have any deadlines associated with them. In order to schedule an NRT task with EDF the idea is to assign an artificial deadline for each NRT task (Note: Other aperiodic event-driven tasks, may have hard or soft, real-time requirements and would be assigned artificial deadlines as well for their respective task type (i.e. HRT | SRT)). All NRT tasks share the percent utilization reserved for them at system startup. Like SRT tasks, NRT tasks are allocated resources based on a proportional average which is computed as follows:

$$wNRT_i = \frac{\max\{\beta, 1 - U_{HRT} - U_{SRT}\}}{NRT_n} \quad (13)$$

where $wNRT_i$ represents the new utilization average, U_{HRT} is the total utilization for all HRT tasks and U_{SRT} defines the total utilization of SRT tasks, NRT_n represents the number of NRT tasks while β defines the bandwidth reservation for all NRT tasks. The artificial period for an NRT task is done by assigning an artificial WCET to each NRT task.

$$P_{NRT_i} = \frac{C_{NRT_i}}{wU_{NRT_i}} \quad (14)$$

The period of an NRT task remains fixed at run time while the task WCET may be adjusted to achieve optimal response times for each NRT task. The period remains fixed to reduce the overhead of continued weighted resource allocation computations because of the transitory nature of NRT tasks.

To illustrate how tasks are scheduled during a rate-based mode change consider the following example. Tables 1 and 2 represent a nominally loaded and overloaded taskset where overloaded means that $U > 1$.

Table 1: Rate Based Task Set (nominal)

Task Type	WCET	PERIOD	DEADLINE
HRT	2	10	10
HRT	3	15	15
SRT	3	20	20
SRT	5	50	50
NRT	4 (pseudo)	-	80 (pseudo)

Table 2: Rate-Based Task Set (overloaded)

Task Type	WCET	PERIOD	DEADLINE
HRT	2	10	10
HRT	3	15	15
SRT	3	20	20
SRT	5	50	50
NRT	4 (pseudo)	-	80 (pseudo)

Figure 2a and 2b shows the scheduling of rate-based tasks. Figure 2a details how each task type is scheduled without needing mode changes since the system is underloaded. The NRT task is treated as a sporadic task with a pseudo *wcet* of 8 as well as a pseudo *period* of 80. The pseudo values are calculated using a fixed rate 0.1 for the NRT task. Note that the NRT task(s) could receive more than the pseudo *wcet* if there is slack in the schedule. Figure 2b illustrates what happens to the allocation of the SRT tasks when the system is overloaded. The HRT tasks are still allocated their target resource rate. If any HRT task cannot be allocated its requested target rate, it will not be admitted. The NRT task(s) are always allocated their reserved bandwidth, but multiple NRT tasks would have to share that bandwidth. As indicated in Table 2, the task set is overloaded so that the SRT tasks will require proportional allocation per equation 6. For this example, the first SRT period is extended from 20 to 25, and the second SRT task's period is extended from 50 to 65, as shown by the arrows in Figure 2b.

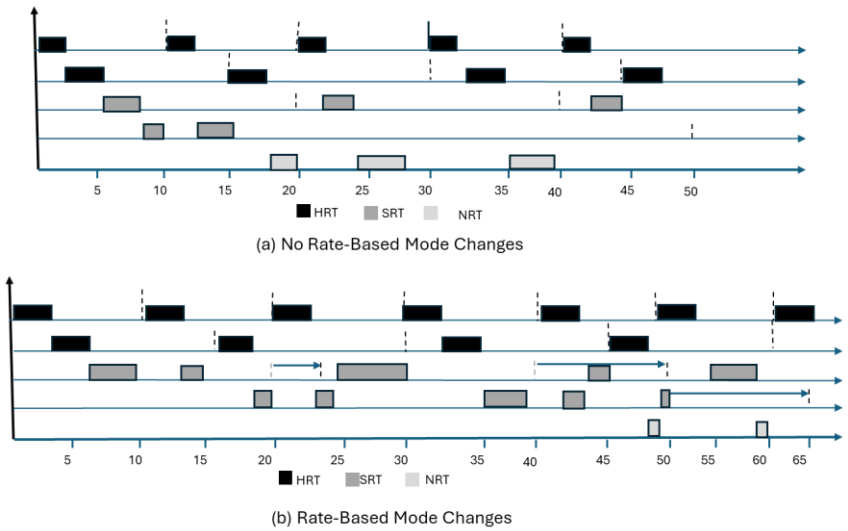

Figure 2: Rate-Based Scheduling Example

3.2.3. Rate-Based Design Considerations

A rate-based scheduler dynamically allocates CPU time based on task execution rates requiring frequent calculations and adjustments which introduces additional overhead. In contrast, fixed-priority schedulers have minimal overhead since the task priorities are static and do not require additional calculations. To minimize CPU overhead HRT tasks are considered static and do not require additional calculations. Additionally, a pseudo *wcet* is calculated, using eq. (14), to be a fixed value at run time. This avoids the significant additional overhead of resource allocation that is triggered by the weighted resource allocation process defined by eq. (6) when an NRT process enter or leave the system. Other design considerations involves the scalability of multiple tasks with varying execution times and system loads. As the number of tasks increases, the complexity of maintaining rate guarantees also increases, which may cause performance degradations due to the increased overhead in managing the task list. However, unlike typical Linux based platforms which could host hundreds of tasks, most embedded processing involves a much smaller task load. While there is no specific limit on the number of tasks admitted, the assumption is that the task load will be on a much smaller scale as opposed to a Linux based system.

Balancing the fairness of CPU allocation of tasks is also an important consideration. While rate-based scheduling offers better fairness compared to priority-based approaches there is a tradeoff when implementing fairness. If the system is overloaded with HRT tasks, then SRT or NRT tasks could suffer from starvation. Fairness is adjusted by using the weight-based model (i.e. eq. 6) so higher weight tasks may not be starved.

3.2.4. Rate-Based Scheduler RTOS Design

In VxWorks, the design of a rate-based scheduler fits into the overall RTOS architecture by interfacing with some core kernel components, including task management, interrupt handling, and time management. The rate-based scheduler is integrated into the task management system, which handles task creation, deletion, and creation. For each new task, additional attributes are added to the task control block (TCB), including the task rate set during task creation.The tick timer is used in VxWorks to generate periodic interrupts, measure time, and schedule tasks. The rate-based scheduler uses the tick timer to manage task execution times and schedule tasks according to their rates. For example, if a task were to run every 20 milliseconds, the timer would interrupt the scheduler to invoke the task after that interval. In VxWorks, the system clock is used for the timer interrupts and, by default, is set to 60 ticks per second, which translates to approximately 16.67 milliseconds. If a higher resolution tick count is needed, VxWorks will provide a function to change the default tick rate. For this work, we typically set the tick rate to 100 ticks or 10 milliseconds. VxWorks allows higher resolution tick rates if needed, but the higher the rate, the higher the overhead in that every clock tick represents a potential context switch.

Like all RTOSs, a task is placed in a task queue based on priority. The rate-based scheduler modifies this queue by maintaining a list of tasks sorted by their execution rates, dynamically reordering the queue based on the execution rates. In this way, tasks would be selected for execution based on their rate, which requires modifications to the TCB to track the following execution times. Fortunately, VxWorks provides built-in mechanisms for extending the TCB. The algorithm for creating a rate-based task into the RTOS is illustrated by the pseudo code provided below:

Algorithm1 Task RB Create

```
Procedure taskRBCreate(struct tcb tcb) now
    ← tickGet()
    If taskInQueue(tcb) then return
    End if
    if type = TASK TYPE NON REALTIME then
     if tcb.weight = 0 then
            tcb.weight←1
        else
            tcb.weight←nrtTaskWeights
        end if
        usrSchedBlk.u nrt ← usrSchedBlk.u nrt + tcb.weight usrSchedBlk.n nrt
        ← usrSchedBlk.n nrt + 1
        tcb.deadline ← tcb.period ← usrSchedBlk.current.nnrt - sysClkGet()
        tcb.releasetime ← now
        qReadySort()
    else if type = TASK TYPE SOFT REALTIME then usrSchedBlk.u srt
        ← usrSchedBlk.u srt + srtActUtil(tcb) usrSchedBlk.n srt ←
        usrSchedBlk.n srt + 1
    else if type = TASK TYPE HARD REALTIME then usrSchedBlk.u
        hrt ← usrSchedBlk.u hrt + hrtTgtUtil(tcb) usrSchedBlk.n hrt ←
        usrSchedBlk.n hrt + 1
    else
       return
    end if
    if usrSchedBlk.u hrt + usrSchedBlk.u srt + BETA >1.0 then resourceAllocation(tcb)
   end if
    if tcb.release time <now then
        kernelPanic("Releasedinthepast!")
   endif
    dcb.etime←0
    qReadyInsert(dcb)
end procedure
```

The *taskRBCreate* algorithm is designed to manage the scheduling of tasks in a real-time operating system. It starts by obtaining the current system tick count and checking if the task is already in the ready schedule queue. If the task is not in the queue, it updates various counters and parameters based on the task type (non-real-time, soft real-time, or hard real-time). It adjusts the task's weight for non-real-time tasks and updates the scheduler's non-real-time utilization and task count. It updates the respective utilization, and task counts for soft and hard real-time tasks. The algorithm ensures the scheduler is not overloaded by checking the total utilization against a threshold. If the task's release time is past, it triggers a kernel panic. Finally, it resets the task's execution time and inserts it into the ready queue. Additionally, during task creation, the resource allocation algorithm determines the utilization for each task type to ensure that tasks are appropriately scheduled, and their resource requirements are managed efficiently.

Algorithm2 resource Allocation

```
Function resource Allocation(struct tcb tcb)
    Int util
    if type = TASKTYPEHARDREALTIME then
        util ← utilHRT(tcb)
    else if type= TASKTYPESOFTREALTIME then
        util ← utilSRT(tcb)
    else if type =TASK TYPEBEST EFFORT then
        util ← max(()BETA,(usrSchedBlk.uhrt-usrSchedBlk.usrt)*
```

```
tcb.weight)usrSchedBlk.unrt
        tcb.period ← usrSchedBlk.nnrt*sysClkGet( )
        tcb.deadline←tcb.period
    else
        return-
    1 end if
    return util
end function
```

The resource allocation algorithm is designed to determine the utilization of a task based on its type and update its scheduling parameters accordingly. It first checks the type of the task: if it is a hard real-time task, it calculates the utilization using the *utilHRT* function; if it is a soft real-time task, it uses the *utilSRT*function. For best-effort tasks, it calculates the utilization using a formula that considers the maximum of a constant BETA and the difference between hard and soft real-time utilizations, scaled by the task's weight and normalized by the non-real-time utilization. It then sets the task's period and deadline based on the number of non-real-time tasks and the system clock. Finally, the calculated utilization is returned.

4. VXWORKS IMPLEMENTATION

For this work, we adopted a hybrid rate-based scheduling policy where rate-based tasks are allocated separately, but the traditional kernel scheduler is used for intra-kernel resource allocation. Rate-based scheduling manipulates the ready queue by changing task priorities and activating or suspending tasks. This hybrid approach was taken for a couple of reasons. One is that researchers noticed mixed results [2] when applying a single rate-based resource allocation policy to the problems of kernel and application processing. The second reason is that implementing a unified custom scheduler is a relatively complex task compared with manipulating the ready queue. While a custom scheduler framework is provided, VxWorks cautions that the operating system is not guaranteed to function as expected if a custom scheduler is used. In addition, there are no guarantees about the compatibility of a custom scheduler across different VxWorks releases, as the scheduler interface may be changed.

Figure 3 illustrates the implementation of rate-based tasks into VxWorks. The Rate-Based Ready queue is managed by the function responsible for scheduling the tasks. Monitoring the execution time of a task is performed as follows: whenever a task starts executing, it sets an absolute time (in ticks) for when the task's wcet is met. This time is added to a task event queue, which triggers an event when a task is met. If another rate-base task pre-empts a rate-based task, the remaining execution time is updated by subtracting the time that has passed since the last release.

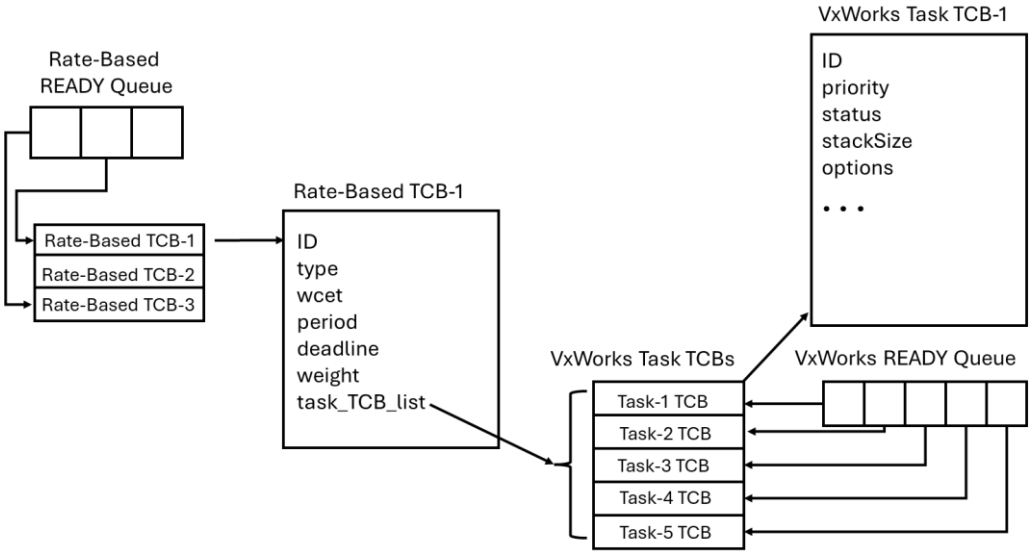

Figure 3 Rate-Based Task Implementation in VxWorks

The rate-based scheduler is implemented as an extension to the traditional scheduler that uses the *tickAnnounceHookAdd( )* function to register a hook function that is called at each tick interrupt. The hook function is responsible for manipulating the ready queue as well as changing their priorities according to the rate-based scheduling policy. The following code example illustrates the hook use:

```
/* task interrupted by tick */
void usrTickHook TASK_ID tid)
{
/* statistic information */
/* call kernelRateBasedHook() */
if (_func_kernelRateBasedHook != NULL)
_func_kernelRateBasedHook(tid);

/* other work */ ... }
```

Tasks are characterized by the following attributes: *{type, wcet, period, deadline}*. The arguments specific the task type (HRT, SRT, NRT), the worse-case execution time, the period and the relative deadline. The resource rate is defined as $wcet/period$. HRT tasks are guaranteed to receive an actual resource rate (u_{act}) equal to the target rate (u_{tgt}) if sufficient CPU resources is available; otherwise, the process will not be admitted as a real-time process. A SRT task receives (u_{act}) less than or equal to (u_{tgt}) depending on available resources. NRT tasks receive a predefined utilization percentage (β) in proportion to their weights as described in section 3.4. To support rate-based tasks a new kernel function call is provided to allocate and initialize a task without activation.

```
TASK_ID taskRBCreate
   (
    char          *name, /* name of new task */
    RB_TASK_TYPE   type  /* type of task (HRT,SRT, NRT) */
    Int            wcet; /* execution time (in ticks)  */
    int           period;  /* task period */
    int            deadline,/* relative deadline */
```

```
    int           options,   /* task option word */
    size_t        stackSize, /* stack size needed */
    FUNCPTR       entryPt,   /* entry point of task */
    _Vx_usr_arg_t arg1,      /* task argument one */
    _Vx_usr_arg_t arg2,      /* task argument two */
    _Vx_usr_arg_t arg3,      /* task argument three */
    _Vx_usr_arg_t arg4,      /* task argument four */
    _Vx_usr_arg_t arg5,      /* task argument five */
    _Vx_usr_arg_t arg6,      /* task argument six */
    _Vx_usr_arg_t arg7,      /* task argument seven */
    _Vx_usr_arg_t arg8,      /* task argument eight */
    _Vx_usr_arg_t arg9,      /* task argument nine */
    _Vx_usr_arg_t arg10      /* task argument ten */
)
```

The rate-based scheduler for VxWorks is designed after the prototype rate-based earliest deadline scheduler presented by authors in [21] with modifications.

5. EXPERIMENTAL SETUP AND TESTING

We integrated our rate-based scheduler prototype into the Cheddar [22] open-source GNU GPL real-time scheduling simulator/analyzer for the initial testing. By leveraging Cheddar, we could model and analyze our scheduler's timing behavior in a controlled environment. This approach allowed us to identify potential issues and optimize performance early in development. In our Experimental Setup, we created various task scenarios and workloads to simulate real-world conditions. Cheddar provided detailed insights into the scheduler's performance, helping us fine-tune parameters and ensure it met all real-time constraints. This pre-implementation testing phase was instrumental in validating our design, ultimately saving development time and reducing the risk of costly errors when deploying the scheduler in VxWorks.

For our testing, we created a task set consisting of five tasks. This task set was designed to represent a typical workload in a real-time system. We included two HRT tasks, two SRT tasks, and one NRT task to simulate an event-based sporadic task. The initial task workload was configured with a total system utilization of 70%. We then randomly modified the execution times of each task, except for the NRT task which remained constant, to increase the workload up to 110% in increments of 5%. The purpose of these modifications was to test the scheduler's performance under varying workloads. The periods of each task were not modified, except by the SRT tasks in the rate-based scheduler to compensate for utilization overload. We used three scheduling algorithms: Rate-Monotonic (RM), Deadline Monotonic (RM), and Earliest Deadline First (EDF) for comparative analysis of the Rate-Based (RB) scheduler. To simulate the non-real-time task, we implemented it as a periodic task in RM and a sporadic task in DM and EDF. The task wcet and period were calculated using an overall system utilization of 10%, which represents the β factor used by the RB scheduler.

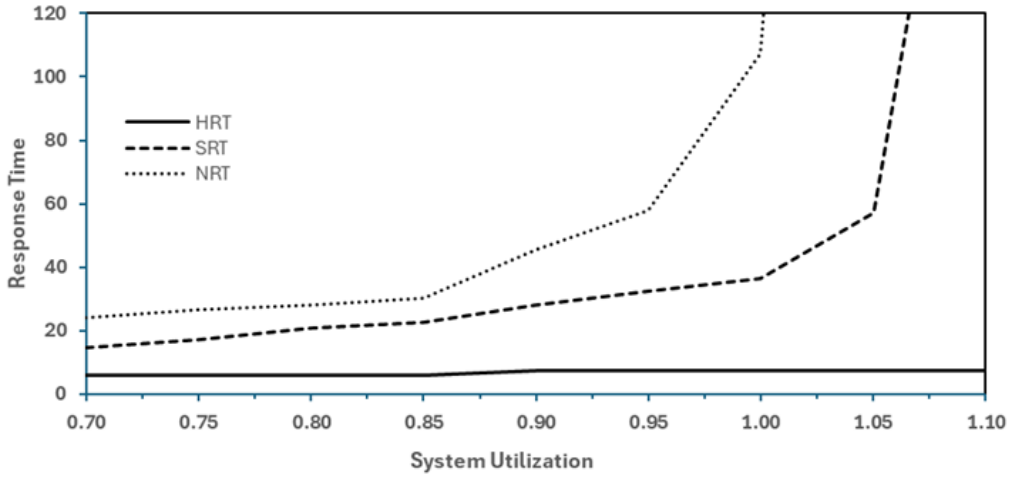

Figure 3a: RM Scheduling

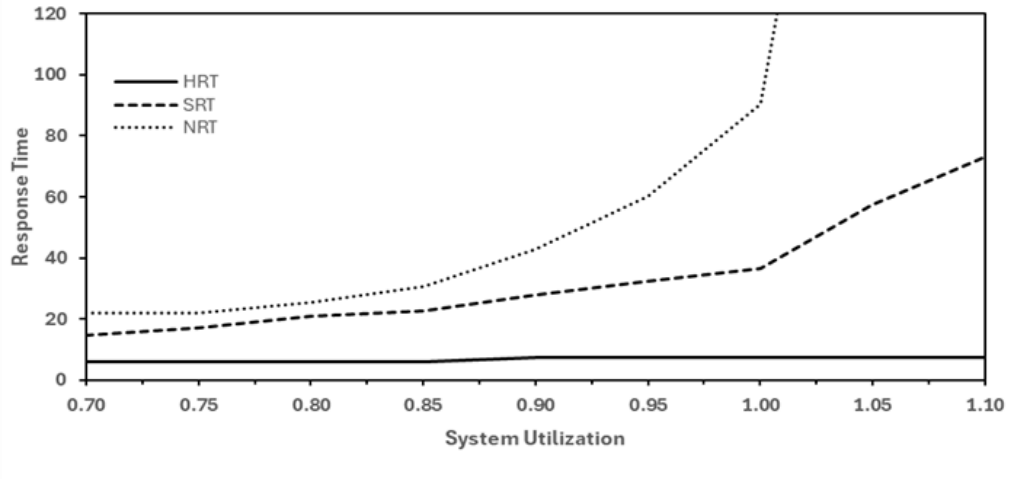

Figure 3b: Deadline Monotonic Scheduling

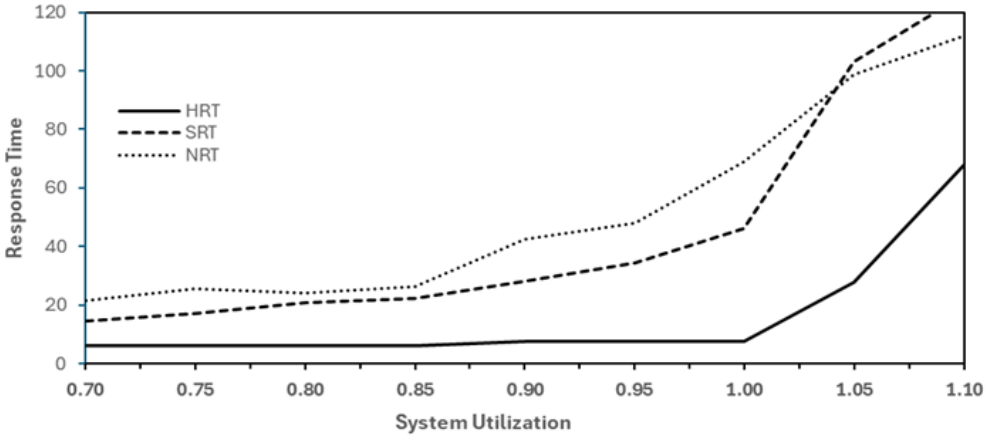

Figure 3c: Earliest Deadline First Scheduling

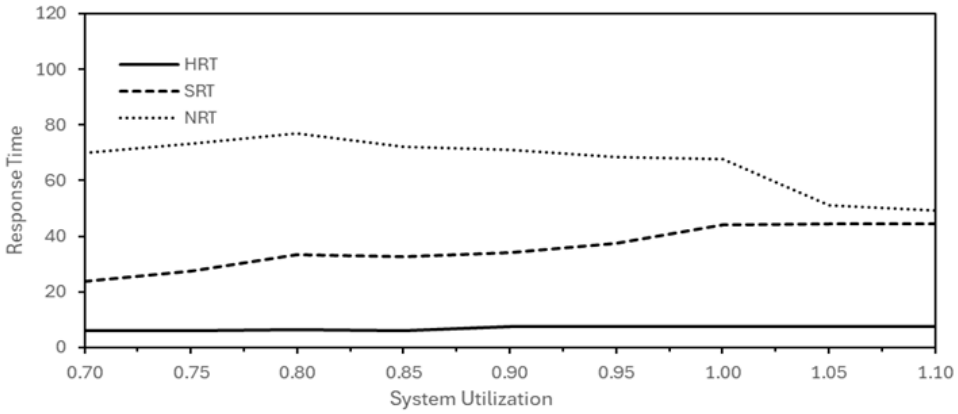

Figure3d: Rate-Based Scheduling

Figure 3a illustrates the RM scheduling, which represents how the different task types perform under rate-monotonic scheduling. Notice that the hard real-time tasks are unaffected, while around 85% of the NRT real-time response times grow increasingly worse. This is expected as the utilization increases since the NRT tasks are assigned a lower priority than the other task types. Additionally, the SRT tasks experience rapidly increasing response times around 100% utilization. This is also expected because as utilization increases, the HRT tasks will be allocated their targeted resource rate, delaying resources to the SRT and NRT tasks. The DM scheduler depicted in Figure 3b performs better because it assigns priorities based on the task's deadline, not the period. Also, notice that with DM, the SRT and NRT task response times increase at a slower rate than those using RM scheduling. Compared to RM and DM, the EDF scheduler performs better if the total utilization is less than or equal to 100%. Notice that as soon as the utilization exceeds 100%, all task types experience significant increases in response times. This, too, is to be expected because of the cascade effect in Earliest Deadline First (EDF) scheduling, which occurs when a single task missing its deadline causes subsequent tasks to miss their deadlines. This leads to a cascade effect where delayed tasks push back the execution of subsequent tasks, causing a chain reaction of missed deadlines and increased response time. Clearly, the RB algorithm is superior for a rate-based task set. While the average response time is higher for NRT tasks and slightly higher SRT, no HRT tasks miss their deadlines, and both SRT and NRT task types maintain relatively uniform response times even after the systems become overloaded. Figure 4 depicts the number of deadline misses suffered by the RM, DM, and EDF algorithms.

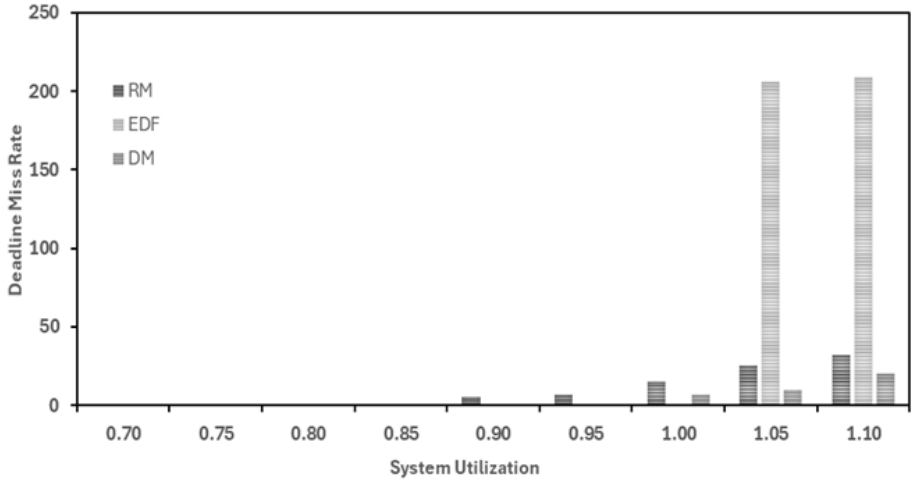

Figure 4: Task Deadline Miss Rate

For implementation into an actual RTOS platform we used VxWorks 7 along with the *VxSim* simulator to implement and integrate the rate-based scheduler. *VxSim* simulates the VxWorks environment on a host machine, providing a virtual platform for testing and evaluating applications without the need for physical hardware. It supports full network capability and can simulate multiple instances, making it ideal for comprehensive testing of distributed applications. We executed tasks in VxWorks using the fixed priority pre-emptive scheduler and our integrated rate-based scheduler for our tests. We used a sample task set to compare the schedulers indicative of real-time and non-real time processing likely to be performed on an IoT device. The sample real-time task workloads consisted of the following tasks:

1. **HRT: Sensor Data Processing:** The task that involve reading data from sensors at regular intervals, such as temperature monitoring systems or motion detectors.
2. **SRT: Multimedia Streaming:** Handling audio or video streams where data packets need to be processed and delivered at a consistent rate to ensure smooth playback.
3. **HRT: Control Systems:** Tasks in industrial automation where control signals must be sent to machinery at precise intervals to maintain proper operation.
4. **SRT: Network Packet Processing:** Managing network traffic where packets need to be processed at a certain rate to maintain network performance and avoid congestion.
5. **NRT: Periodic Data Logging/Aggregation:** Recording data from various sources when available.

Except for the NRT process, each task performed an infinite loop consisting of a *read()*or *write()* operation on a UDP socket or virtual device using *VxBus* followed by a computation phase with a known execution time. In addition, we configured message generators to send and receive data with the desired size and rate to the corresponding sending/receiving task. The goal was to evaluate how rate-based allocation performed compared to VxWorks's traditional priority-based method. Similar to how it was evaluated with the Cheddar simulator, we modified the computation phase of each task to simulate overall system utilization.

Tables 3 and 4 provide the overall minimum, average, and maximum response times for the fiver real-time tasks described above. The response time is measured in ticks, each tick can be from 1-1.99 milliseconds depending upon when the tick value was acquired.

Table 3: Fixed Priority Scheduler (VxSim)

	Task Utilization~ 0.65				Task Utilization ~1.0				Task Utilization ~1.1			
Task Type	**Min**	**Avg**	**Max**	**Miss**	**Min**	**Avg**	**Max**	**Miss**	**Min**	**Avg**	**Max**	**Miss**
Sensor Data Processing	6.0	6.0	6.0	0%.	16.0	16.0	16.0	0%	50.0	59.0	24.0	0%
Multimedia Streaming	6.0	7.2	12.0	0%	8.0	11.4	24.0	20%	5.0	6.5	29.0	5%
Control Actuation	1.0	1.0	1.0	0%	3.0	3.0	3.0	0%	3.0	3.0	3.0	0%
Network Packet Processing	15.0	16.2	20.0	0%	29.0	39.4	50.0	40%	13.0	21.3	58.0	2.5%
Data Logging	35.0	36.2	37.5	0%.	37.0	42.5	45	0%	166.0	180.0	400.0	25%

Table 4: Rate-Based Scheduler (VxSim)

	Task Utilization ~0.65				Task Utilization ~1.0				Task Utilization ~1.1			
Task Type	**Min**	**Avg**	**Max**	**Miss**	**Min**	**Avg**	**Max**	**Miss**	**Min**	**Avg**	**Max**	**Miss**
Sensor Data Processing	12.0	16.0	20.0	0%.	54.0	64.0	74.0	0%	50.0	59.0	74.0	0%
Multimedia Streaming	6.0	6.0	6.0	0%	8.0	8.4	8.0	0%	5.0	6.5	8.0	5%
Control Actuation	1.0	1.0	1.0	0%	3.0	3.0	3.0	0%	3.0	3.0	3.0	0%
Network Packet Processing	15.0	15.0	15.0	0%	29.0	29.0	29.0	0%	13.0	21.3	27.0	0%
Data Logging	37.0	37.0	37.0	0%.	200.0	200.0	200.0	0%	166.0	180.0	187.0	0%

The results are mixed when the overall system utilization is nominal the traditional fixed priority scheduler performs better regarding Vresponse times though the response times for HRT tasks in the rate-based scheduler are comparable to the fixed-priority scheduler. While the SRT tasks response times are increased in the rate-based scheduler both SRT and NRT response times are more uniform as compared the fixed-priority scheduler.Additionally, rate-based scheduling experienced no task deadlines misses with there was a heavy or overloaded task load.

6. CONCLUSIONS AND FUTURE WORK

Rate-based resource allocation mechanisms provide a dynamic approach to managing computational resources, that stand out for their ability to adjust task execution rates based on system demands. Their unique advantage lies in their suitability for real-time systems, where meeting strict deadlines and ensuring resource efficiency are paramount. This is especially true in environments with fluctuating workloads like distributed IoT networks. The implemented scheduler was thoroughly evaluated using the Cheddar scheduling analysis tool, enabling theoretical verification of schedulability and timing performance. Additionally, the VxWorks VxSim simulator was used to verify the scheduler's practical performance in a controlled environment, mimicking real-world execution. Initial results indicate that the rate-based

scheduler successfully manages task execution under varying system loads, improving deadline adherence compared to traditional fixed-priority and round-robin scheduling techniques. This work underscores the potential for a rate-based scheduling approach as an efficient solution for ensuring predictable and reliable performance in real-time distributed systems, offering an alternative to existing scheduling mechanisms in real-time operating systems like VxWorks.

For future work, we have a clear plan to extend the implementation of the rate-based scheduler to run on a single-board computer (SBC), such as the Raspberry Pi orSTM32L4 IoT node. This step will allow us to assess the scheduler's performance and adaptability on resource-constrained devices commonly used in IoT and embedded applications. By transitioning from a simulated environment to actual hardware, we aim to identify any hardware-specific challenges and optimize the scheduler to better handle the constraints of SBCs, such as limited memory, processing power, and energy efficiency. The detailed plan for future work provides a roadmap for the research, helping the reader understand the next steps and the potential impact of the research. Another key area for future work involves expanding the scheduler's capabilities to support resource sharing among cooperating tasks. In real-time systems, tasks often need to share resources such as memory, communication channels, or sensors, which introduces the potential for contention and blocking, leading to missed deadlines. We plan to incorporate resource-sharing mechanisms that maintain the predictability and efficiency of the rate-based scheduler while preventing priority inversion and reducing the likelihood of deadlocks. This will involve integrating priority inheritance or priority ceiling protocols and dynamically developing techniques for managing shared resources based on the rate-based task execution model. Such enhancements will make the scheduler more robust and adaptable for real-world applications that require coordination between multiple interdependent tasks.

ACKNOWLEDGEMENTS

This work was funded in part by a grant from by the Wind River Corporation grant #US1133982

REFERENCES

[1] K. Jeffay and S. Goddard, "A theory of rate-based execution," Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054), Phoenix, AZ, USA, 1999, pp. 304-314, doi: 10.1109/REAL.1999.818858.

[2] Jeffay, Kevin, and Steve Goddard. "Rate-based resource allocation models for embedded systems." International Workshop on Embedded Software. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

[3] Goddard, S., Jeffay, K., Analyzing the Real-Time Properties of a Data flow Execution Para-digm using a Synthetic Aperture Radar Application, Proc. 3rd IEEE Real-Time Technology & Applications Symp., Montreal, Canada, June 1997, pp. 60-71.

[4] Goddard, S., Jeffay, K., Managing Memory Requirements in the Synthesis of Real-Time Systems from Processing Graphs, Proc. of 4th IEEE Real-Time Technology and Applica-tions Symp., Denver, CO, June 1998, pp. 59-70.

[5] K. Jeffay, G. Lamastra, A Comparative Study of the Realization of Rate-Based Computing Services in General Purpose Operating Systems, Proceedings of the Seventh IEEE Interna-tional Conference on Real-Time Computing Systems and Applications, Cheju Island, South Korea, December 2000, pages 81-90.

[6] 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, January 1973, pp. 46-61.

[7] A.K.-L., Mok, Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment, Ph.D. Thesis, MIT, Dept. of EE and CS, MIT/LCS/TR-297, May 1983.

[8] J. Leung, and J. Whitehead, On the complexity of fixed-priority scheduling of periodic, real-time tasks, Performance Evaluation, 2, 1982, pp. 237-50.

[9] K. Jeffay, D. Bennett, Rate-Based Execution Abstraction for Multimedia Computing, Proc. of the Fifth Intl. Workshop on Network & Operating System Support for Digital Audio & Video, Durham, NH, April 1995, Lecture Notes in Computer Science, Vol. 1018, pp. 64-75, Springer-Verlag, Heidelberg.

[10] L. Abeni, G. Buttazzo, Integrating Multimedia Applications in Hard Real-Time Systems, Proc. of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998, pp. 4-13.

[11] M. Spuri, G. Buttazzo, Efficient Aperiodic Service Under the Earliest Deadline Scheduling, Proc. 15th IEEE Real-Time Systems Symp., Dec. 1994, pp. 2-11.

[12] M. Spuri, G. Buttazzo, F. Sensini, Robust Aperiodic Scheduling Under Dynamic Priority Systems, Proc. 16th IEEE Real-Time Systems Symp., Dec. 1995, pp. 288-299

[13] A. K. Parekh and R. G. Gallager, A Generalized Processor Sharing Approach To Flow Control in Integrated Services Networks-The Single Node Case, ACM/IEEE Transactions on Networking, Vol. 1, No. 3, 1992, pp. 344-357

[14] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, C. Plaxton, A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems, Proc. 17th IEEE Real-Time Systems Symposium, Dec. 1996, pp. 288-299.

[15] I. Stoica, H. Abdel-Wahab, K. Jeffay, On the Duality between Resource Reservation and Proportional Share Resource Allocation, Multimedia Computing & Networking ‘97, SPIE Proceedings Series, Vol. 3020, Feb. 1997, pp. 207-214.

[16] C.A. Waldspurger, W.E. Weihl, Lottery Scheduling: Flexible Proportional-Share Resource Management, Proc. of the First Symp. on Operating System Design and Implementation, Nov. 1994, pp. 1-12.

[17] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines, IEEE Transactions on Computers, April 1995.

[18] R. West, K. Schwan, and C. Poellabauer, Scalable scheduling support for loss and delay constrained media streams, Proceedings of the 5th IEEE Real-Time Technology and Appli-cations Symposium, Vancouver, Canada, June 1999.

[19] Kevin Jeffay, F. Donelson Smith, Arun Moorthy, and James Anderson. Proportional share scheduling of operating system services for real-time applications. In Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS98), pages 480–491. IEEE, December 1998.

[20] Jeffay, Kevin, and Steve Goddard. "The rate-based execution model." CSE Technical reports (1999):

[21] Brandt, Scott A., et al. "Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes." RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003. IEEE, 2003.

[22] Cheddar - open-source GNU GPL real-time scheduling simulator/analyzer. https://beru.univ-brest.fr/cheddar/