Nip it in the Bud: Job Acceptance Multi-Server

Anna Friebe[1], Tommaso Cucinotta[2], Filip Marković[3], Alessandro Vittorio Papadopoulos[1], and Thomas Nolte[1]
[1]*Mälardalen University (MDU)*, Sweden
[2]*Scuola Superiore Sant'Anna*, Italy
[3]*University of Southampton*, United Kingdom

***Abstract*—Computationally demanding tasks with highly variable execution times may require parallel processing. Scheduling such tasks with low deadline miss rates but without significant overprovisioning is challenging. This issue arises in applications like nonlinear optimization for Model Predictive Control (MPC). The Constant Bandwidth Server (CBS) provides timing isolation, supporting both hard and soft real-time tasks. However, scheduling parallel, time-varying jobs across multiple CBS instances requires static job-to-server assignments, which can lead to resource underutilization due to queued jobs awaiting specific servers. This paper introduces the Job Acceptance Multi-Server (JAMS), a mechanism in which multiple CBS instances share a common job queue, enabling flexible job dispatching for parallel workloads. JAMS incorporates a job dismissal mechanism to address overloads, ensuring that only jobs with guaranteed resource availability are accepted. Each CBS instance checks if it can complete a job by its deadline, given probabilistic knowledge on its execution times, dismissing unfeasible jobs to avoid excessive tardiness across queued tasks. Implemented in Linux, JAMS is evaluated with computation times drawn from an MPC task and synthetic datasets. The extensive experimental results we provide, demonstrate that JAMS effectively controls the deadline miss rate, maintaining it below a specified design threshold.**

***Index Terms*—probabilistic scheduling, job dismissal**

I. Introduction

In real-time systems, computational workloads can often be decomposed into smaller tasks that execute either in parallel or sequentially. Certain computations require strict sequential execution, but multiple such computations may be active concurrently, operating on different data. For instance, in Model-Predictive Control (MPC), an optimization problem is solved to determine a state trajectory and control commands over a defined horizon, with a time-varying convergence, particularly in nonlinear systems. As shown in [1], such tasks for different time windows can be processed in parallel on predicted states, updating results when states become available. Methods based on particle filters such as FastSLAM are often parallelized [2], [3] and may use an adaptive number of particles, resulting in varying computation times [4]. A state estimate can be obtained from a smaller number of particles although it may be less accurate [4]. Allocating computational resources to a collective group of tasks with a unified objective rather than to individual tasks enables efficient resource utilization..

The Constant Bandwidth Server (CBS) [5] provides timing isolation and supports integration of both soft and hard real-time tasks. In multicore systems, CBS instances may be allowed to migrate across cores, although each instance's utilization is capped at 1 [6]. Thus, CBS alone is not sufficient for resource assignment to a group of parallel tasks working toward a combined outcome, i.e. a thread pool.

To address this limitation, we introduce the Job Acceptance Multi-Server (JAMS), where multiple CBS instances share a global job queue, and queued jobs are dispatched when a server becomes available.

If a task in a server misbehaves and requires more computation resources than assumed at design time, the server guarantees that tasks outside the server still receive the required resources. However, job tardiness within the server may grow unboundedly, potentially disrupting the system functionality even when other tasks have sufficient computational resources. In such cases, JAMS utilizes a job dismissal mechanism to prioritize jobs with a high likelihood of meeting their deadlines, mitigating overload situations and allowing for smoother system recovery.

Job dismissal, used in some schedulability and response time analysis of real-time systems [7], [8], assumes that jobs are dismissed upon deadline misses or at another specified dismissal point, enhancing analytical tractability and mitigating adverse effects from misbehaving tasks or inaccuracies in execution time estimates. For instance, if the analysis relies on a Worst Case Execution Time (WCET) that was erroneously estimated, dismissing a job once it consumed its erroneously considered WCET at analysis and admission time, ensures that the analysis still holds for the jobs that behave correctly.

Despite these advantages of job dismissals, many real-time schedulers do not provide such a feature. One alternative is to implement in-task job abortion [9], or through dedicated programming abstractions, such as the *deadline exception* introduced in [10]. While this is often a suitable alternative that allows for tailor-made abortion points and can be implemented in addition to the proposed JAMS mechanism, it requires that the tasks are bug-free and non-compromised. Tasks cannot consider the total load of the system in the dismissal decision. Furthermore, dismissing a job at the time when it is about to start execution, rather than at a dismissal point during execution, frees resources for other jobs. To address these challenges, JAMS ensures that jobs dispatched to a server are guaranteed a certain amount of computation time prior to their deadline, by dismissing tardy jobs if needed..

Contribution: The contribution of this paper is twofold:

1) The JAMS framework, which enables a group of CBS servers to pick jobs from a global shared queue,

This work has been funded through the Knowledge Foundation grants No. 20240011, 20170214, and 20230146.

facilitating flexible dispatching of jobs with varying computation times and a joint goal.

2) A job acceptance policy that dispatches jobs for processing only if there are resources guaranteeing a high probability for them to meet their deadlines, dismissing jobs queued beyond acceptable queue times.

II. Related Work

Task models for parallel workloads include the fork-join model [11], where a main thread executes sequentially up until a point where it forks into a number of threads that execute a parallel part of the computation. This is followed by a synchronization where the parallel threads are terminated and the main thread continues. An extension is the parallel synchronous task model [12], where parallel computational segments may be consecutive, and two segments may have different number of threads. Lupu and Goossens [13] presented a multi-thread periodic task model, where each task periodically generates a number of subprograms (threads). Hard real-time fixed-priority schedulability tests for constrained deadlines are provided. In the parallel MPC example from [1], optimizations start periodically, but they overlap, so there are no points when threads simultaneously synchronize and join.

Another common parallel task model is the gang task model, where a number of threads are required to run concurrently because they interact. Coscheduling of such processing working sets was introduced by Ousterhout [14]. A one-gang-at-a-time policy [15] has been proposed to reduce interference and turn the multicore parallel scheduling into an equivalent of uniprocessor scheduling. Recently, soft real-time scheduling of gang tasks has been considered [16], including presenting server-based scheduling policies and schedulability tests. For the tasks in this paper, we consider functionality that may be run in parallel, but there is no interaction or need for coscheduling, so the gang task model is unnecessarily restrictive.

Scheduling of a set of tasks with multiprocessor bandwidth reservations has been implemented to support hierarchical scheduling with bounded delay [17]. A similar approach has been taken to support real-time containers [18]. Both these approaches use control groups to separate task sets and modify the Linux CBS implementation `SCHED_DEADLINE` to schedule the root level of a hierarchical schedule, while the lower level is scheduled with fixed priority. In JAMS, CBS is used for the low-level scheduling.

For systems without job dismissal, it is clear that the average requested computational resources must be lower than the average provided resource for these systems to be stable [19]. In such a stable system, there exist points in time when the job queue is empty [20]. In a system where jobs are discarded, stability can be achieved without this requirement, but some of the requested computational resources may be rejected due to the discarding policy. Jobs may also be delayed or discarded before they start due to a lack of resources, scheduling priorities, and discarding policy. Chen *et al.* [21] compared the effect on the deadline miss rate of varying the dismiss point after the deadline for a uniprocessor system where task computation times were independent discrete random variables and the average resource demand was lower than the supply. While not directly applicable to our multiprocessor use case, potentially with overload and correlated computation times, a later dismiss point resulted in a higher deadline miss rate, but the rate converged. Manolache *et al.* [8] analyzed task graphs with stochastic computation times with an upper bound on the number of concurrently active instantiations of each task graph. They concluded that denying service to a newly arrived task graph considerably reduces the number of states and schedulability analysis time compared to rejecting the oldest instantiation in the system. Pazzaglia *et al.* [22] compared the effect on control robustness of different strategies of handling deadline misses, including different dismissal policies. For these control applications, killing a job at the deadline or skipping the next job if a deadline has been missed were both preferable to queueing jobs.

In the area of mixed criticality systems, LO-criticality jobs are aborted or dismissed to ensure that HI-criticality jobs meet their deadlines [23]–[25]. One of the criticisms from system engineers discussed in [23], [26] of the assumptions of Mixed-criticality systems is that LO-criticality jobs should receive some service if at all possible. In a resilient system [23] started jobs run to completion, and a task is considered robust if it can safely drop one *non-started* job in any extended time interval [27]. In the bailout protocol [26], HI-criticality jobs that overrun their budget continue their execution. The overrun is compensated for by not starting LO-criticality jobs, and by accounting for jobs that use less resources than budgeted until the system has returned to normal execution mode. In [28] HI-criticality jobs were monitored in a multiprocessor system, and if one such job risked exceeding its isolation-based WCET, concurrently running LO-criticality jobs were suspended to ensure they didn't interfere.

In the Robust Earliest Deadline algorithm [29] EDF-scheduled tasks in a uniprocessor system are associated with values, deadline tolerances and a criticality level. If a job arrival leads to a WCET-based overload situation, non-critical jobs with low value are rejected from the ready queue. Rejected jobs may be recovered if jobs complete early. Due to the preemptive scheduling, partly computed jobs may be aborted. In [30], this approach was applied to aperiodic jobs in a Total Bandwidth Server (TBS). It has been pointed out that QoS guarantees for individual tasks cannot be provided in such a case, but a separate server for each task is required [31].

In [32] weakly hard real-time systems were introduced, enabling specification of a minimum number of met or consecutively met deadlines in every window of a specified number of task invocations, or a maximum number of consecutive deadline misses. These are alternative or complementary to the probabilistic approach.

Tong *et al.* [33] proposed holistic budgeting of Directed Acyclic Graph (DAG) tasks to decrease DAG drop rates. The slack from nodes that complete early and from nodes that cannot start due to overruns in predecessor nodes is used to complete overrunning nodes. This leads to a lower drop rate

than dropping the DAG when one node overruns, showing the advantages of holistic budgeting and scheduling of potentially parallel work. In queuing theory, a significant amount of work analyzes queues with reneging or customer abandonment. Kruk *et al.* [7] analyzed EDF queues where jobs are dismissed at their deadline. Ward [34] surveyed results for queues with reneging. Reneging can refer to client abandonment, where a client (job) leaves the queue according to a probability distribution over the waiting time or reneging when a deadline is met, or a buffer is full. Ward concludes that for the overloaded many-server case, the story is complete. Towsley and Panwar [35] introduced Stochastic Earliest Deadline policies in the case where deadlines are not known to the scheduler, and analyzed the finite buffer case. If a job arrives at a full buffer, the policy of removing the job stochastically closest to its deadline is at least as good as the arbitrary policy. Whitt [36] investigates overloaded queues with different abandonment time and service time distributions. Abandonment time distributions significantly affect the mean queue length and waiting time. Most work considers abandonment distributions that depend on job waiting times, but there is also admission control or buffer overflow management that takes into account the state of the queue as a whole. More recently, Whitt has reviewed work on time-varying queues [37]. In [38] Whitt studied service rate controls to stabilize queue performance with time-varying arrival rate. Performance measures were mean queue lengths and mean waiting times. It was shown that any control that asymptotically stabilizes mean queue lengths cannot also stabilize mean waiting times.

In networking, packets sometimes need to be dropped due to full buffers. In Active Queue Management, packets are dropped preemptively before the buffer is full, to reduce congestion and waiting times. Random Early Detection [39] has been proposed for congestion avoidance, where packets are dropped with a probabilistic approach. More recently, CoDel has been proposed [40], that is based on the minimum waiting time in the queue over a specified time interval. If this minimum waiting time exceeds a target value, packets start to be dismissed. The time between successive dismissals is decreased until the target waiting time is met.

III. System Model and Notation

A. Task Model

We consider a task τ that releases at most κ jobs at the same instant. Job release instants are separated by at least a minimum separation time p. Each job J_j has the arrival time a_j. A job computation time c_j is an outcome of the random variable $\mathcal{C}$; thus, the job's finishing time f_j and response time are outcomes of random variables as well. We denote the response time random variable $\mathcal{R}$. The task has a relative deadline D, so each job has the deadline $a_j + D$. The job computation times are upper bounded by the WCET $c^{\uparrow}$.

The computation time quantile c_ϕ is defined as:

$$c_\phi := \inf\{x : \mathbb{P}[\mathcal{C} \leq x] \geq \phi\} \tag{1}$$

TABLE I
Overview of notation.

Symbol	Description
τ, J_j	Task, job j of τ.
κ	Maximum number of concurrently released jobs of τ.
p	Minimum separation of job release instants of τ.
D	Relative deadline of τ.
a_j, f_j	Arrival and finishing time of J_j.
c_j, $c^{\uparrow}$	Computation time of J_j and the WCET of τ.
$\mathcal{C}$, $\mathcal{R}$	Computation time and response time random variables.
c_ϕ^ω	The ϕ quantile of the computation times of τ for a given ω.
n	Number of servers.
S_i	Server i.
P_i, Q_i, U_i	Period, maximum budget and utilization of S_i.
$U_{i\Sigma}$	Total utilization on S_i's processor.
q_i, d_i	Remaining budget and deadline of S_i.
t, δt	Time, interval when a server is executed.
$g_{i,j}$	Guaranteed computation time from S_i prior to J_j's deadline.
$\delta_{i,j}$	Difference between the deadlines of J_j and S_i.
$\mathcal{W}$	Random variable, queue wait time.
$\mathcal{B}$	Random variable, remaining work of served jobs.
$\mathcal{L}$	Random variable, work of queued jobs.
$\mathcal{G}$	Random variable, guaranteed computation time of a job.
Δ	Time interval length.
$\Upsilon(\Delta)$	Random variable, work arriving in an interval of length Δ.
ρ	Bound on the work arrival rate.
p_d	Dismissal probability.

The random variable $\mathcal{W}$ is the waiting time of an arriving job. The random variable $\mathcal{B}$ is the remaining work of serviced jobs, and the random variable $\mathcal{L}$ is the work of all queued jobs. The work arriving in an interval of length Δ is a random variable denoted by $\Upsilon(\Delta)$. $\mathcal{L}$ and $\Upsilon(\Delta)$ are sums of computation time random variables, and $\mathcal{B}$ is upper bounded by such a sum.

The notation used in the paper is outlined in Table I.

B. CBS Background

We recall the CBS [41] operation. A real-time task τ is scheduled by an associated CBS S_i, described by its static parameters: maximum budget Q_i and period P_i, resulting in a utilization or bandwidth of $U_i = \frac{Q_i}{P_i}$. S_i also keeps track at run-time of the remaining budget q_i and the absolute deadline d_i, two dynamic quantities that change with the current time $t \in \mathbb{N}$. Let δt denote the length of a generic time interval in which (a job of) a server has been executed on a CPU. S_i is *idle* at time t, if τ has no pending jobs at t, that is if at time t, there exists no job J_j such that $a_j < t < f_j$. If S_i is not *idle* at time t, it is *active* if it has remaining budget ($q_i > 0$). If the budget is depleted, it is *recharging*. A flowchart of the runtime changes of the server is shown in Fig. 1. For the purpose of this paper, each server runs on a specified processor. The total utilization of the processor of S_i is denoted by $U_{i\Sigma}$. We assume partitioned EDF scheduling of servers, flanked with a per-CPU utilization-based test, which ensures each CBS gets its reserved budget within its deadline

C. Scheduling Parallel Workload With a Group of CBS

The task τ is scheduled by a JAMS to support a potentially parallel workload. n CBS instances, each with server period P and maximum budget Q, are set up to share the same job

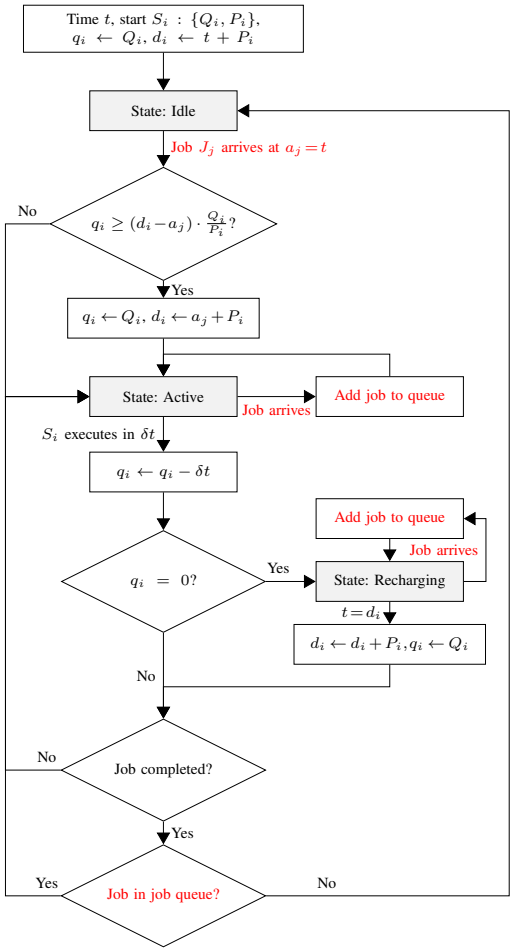

Fig. 1. Flowchart of the CBS update at runtime. The job handling to be updated by the JAMS is marked in red.

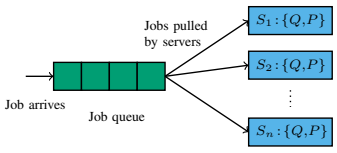

Fig. 2. Overview of JAMS.

queue, as illustrated in Fig. 2. Essentially JAMS implements a thread pool of n worker threads with guaranteed processing capacity, along with an interface to submit work to be processed, and retrieve information about the work's status and the result if it is ready. The operation of JAMS is described in Section V.

D. Definition and Assumptions

We define the concept of JAMS overload in a time interval:

Definition 1. *A JAMS is overloaded in the interval* $[t_0, t_1]$ *if the total computation times of arriving jobs in the interval exceeds the expected provided resource by the servers, that is if* $\frac{n \cdot Q \cdot (t_1 - t_0)}{P} < \sum_{j | t_0 \leq a_j \leq t_1} c_j$.

In the remainder of the paper, we make the assumptions outlined below. Asm. 1 indicates that for intervals longer than Δ_B, individual job computation times become negligible relative to the total workload, and the amount of arriving work in the interval is characterized by a work arrival rate bound ρ. Asm. 2 implies that a job arriving when an idle server has maximum budget must be accepted.

Assumption 1. *For time intervals longer than* Δ_B*, the amount of arriving work* $\Upsilon(\Delta)$ *is bounded by a work arrival rate bound* ρ *and a probability* ϵ*, such that* $\mathbb{P}[\Upsilon(\Delta) > \rho \cdot \Delta] < \epsilon, \Delta > \Delta_B$.

Assumption 2. *The server configuration allows for completing a job with the computation time quantile within the task's deadline, that is* $c_\phi \leq Q \cdot \lfloor \frac{D}{P} \rfloor$.

IV. Motivating Examples and Problem Formulation

Example 1. *A task* τ *releases one job* J_j *every* $p = 20$*. The result of* J_j *is expected at latest at* $a_j + 60$*, we have* $D = 60$*. There is a fallback option in the case of occasional deadline misses. The computation time of a job* J_j *is a random variable, that takes the value of* 20 *with probability* 0.9 *and* 38 *with probability* 0.1*, the probability mass function is* $(\mathbb{P}[\mathcal{C} = 20] = 0.9, \mathbb{P}[\mathcal{C} = 38] = 0.1)$*. The computation time random variables are independent and identically distributed (i.i.d.).*

For this example, an average of 21.8 units of work arrive every 20 time units. Assuming that each processor provides 20 computational units in each period, the task in Example 1 cannot be scheduled on a single processor or in a single CBS.

When scheduling τ, we have some options. We may split τ into s tasks τ_1, τ_2, ..., τ_s with $p = 20 \cdot s$, where τ_1 releases J_1, J_{1+s}, $J_{1+2 \cdot s}$, $\cdots$, and τ_2 releases J_2, J_{2+s}, $J_{2+2 \cdot s}$, $\cdots$ and so on. These different tasks can be scheduled as exclusive tasks on isolated processors or in separate CBS instances. For this example, the task is divided into two tasks ($s = 2$), each task scheduled in a CBS with maximum budget $Q = 15$ and period $P = 20$. An illustration is provided in Fig. 3. On average, a job requires 21.8 units of computation, and the server provides 30 units for each job arrival. Due to the varying computation times, some jobs will not start at their arrival time. For instance, if a job J_j with computation time 38 starts at its arrival time, the next job in the same server, J_{j+2} will begin the latest 13 time units after its arrival, at $a_{j+2} + 13$. This job will be waiting even if other servers are idle at this time. If the computation time of J_{j+2} is also 38, it may finish at $a_{j+2} + 66$, and miss its deadline as illustrated in the separate queues case of Fig. 3. From simulation with the reservation of competing servers positioned at the beginning of each period, we see that the deadline miss rate is approximately 1%. A server is idle and not throttled for about 21% of the time.

Using a joint job queue and starting a waiting job as soon as any server is available reduces the risk of missing a deadline. Simulating the same task served by the same CBS instances sharing a joint job queue, the proportion of time where at least one server is idle and not throttled is about 41%, and the deadline miss rate reduces to approximately 0.19%. J_{j+2} is delayed as a direct effect of the long job

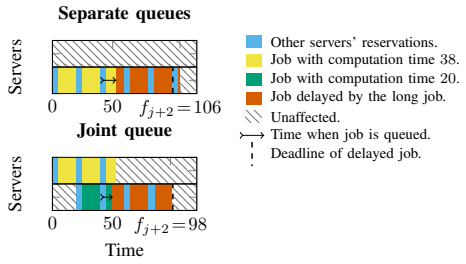

Fig. 3. Illustrations of jobs with delayed start due to a long computation time in Example 1. Arrows indicate the time with a non-empty queue.

J_j, but if J_{j+1} has a short computation time, J_{j+2} starts latest at $a_{j+2}+10$. J_{j+2} will still meet its deadline if it has long computation time as illustrated in the joint queue case of Fig. 3. Using a joint queue for time-varying jobs, such as an MPC-task, decreases the deadline miss rate.

Example 2. *We use the task in Example 1, but with a different computation time probability mass function where long computation times are more common. In this example,* $(\mathbb{P}[\mathcal{C}=20]=0.4,\mathbb{P}[\mathcal{C}=38]=0.6)$.

The task in Example 2 has an average computation demand of 30.8 units per job, higher than the computational resource of 30 that is provided by the servers. This means that the queues will soon start growing, and all jobs will miss their deadlines.

Although job dismissal often is not explicitly implemented in schedulers, it may be implicit. Consider a single-thread task scheduled with `SCHED_DEADLINE` in Linux. If a job overruns and the task is implemented to self-suspend until the time for the next job activation, it will immediately continue with the next job. In a sequential task where jobs retrieve the most up-to-date data, this implicitly implements the skip-next policy [22] if the next job starts once new data has arrived. However, with the need for parallel jobs comes a need to specify the data connected to each job. The most straightforward way to handle this is to put the data or jobs in a queue, leading to the risk of unbounded queue length and tardiness in an overload situation. Support for dismissal of items from the queue is a remedy to this problem.

Problem Formulation

Devise a job acceptance policy with low overhead for JAMS, with the following properties when applied to a task as outlined in Section III:

1) Out of jobs that are accepted, a configured proportion ϕ meet their deadline.
2) If the JAMS is in a long overload interval, the proportion of dismissed work approaches the proportion of computation requirement that exceeds the computation resource provided by the servers.
3) Given computational resource that exceeds the average requirement, a bound on the worst case job dismissal probability is derived, based on the arriving work in each possible interval.

V. JAMS and Job Acceptance Test

In this section, we outline the operation of the proposed JAMS, and provide an analysis of its properties.

A. JAMS Operation

Jobs arrive at the JAMS, which can access the states of its CBS. The JAMS is configured with the task relative deadline D, a quantile ϕ of jobs that shall meet their deadline if they are not dismissed, an estimate of this computation time quantile c_ϕ, and the task's WCET $c^\uparrow$. To ensure that resources are provided to jobs with a reasonable chance of meeting their deadline, a job J_j is transferred to a server S_i only if the server can guarantee sufficient resources prior to the deadline of J_j. The runtime operation of a JAMS and one of its CBS servers is illustrated in Fig. 4.

When a job arrives to the JAMS, and there exist CBS in state **Idle**, the arriving job is offered to these CBS in arbitrary order. If there is no such CBS that accepts the offer, the job is added to the FIFO job queue with its arrival time. A CBS with more suitable q and d may pull the job from the queue later. When a job is pushed to the queue, any jobs at the front of the queue that cannot be provided sufficient computational resources prior to their deadline by any server are dismissed. Then the arriving job (with its arrival time) is added to the job queue.

When a CBS S_i is offered an arriving job or tries to find a job on the queue to pull, it considers the amount of computation time it can guarantee prior to the job's deadline. For a job J_j with deadline a_j+D, the guaranteed computation time till the job deadline is denoted as $g_{i,j}$. A job is accepted if at least c_ϕ computation time can be guaranteed prior to the deadline, i.e.:

$$g_{i,j} \geq c_\phi. \tag{2}$$

When a server attempts to pull a job from the queue, it first considers the job at the front of the queue. If this is not accepted, it continues with the next job until a job is accepted or the end of the queue is reached.

Race conditions may occur that are not included in the illustration Fig. 4. We note that multiple servers may simultaneously access the job queue. Two versions of the JAMS are considered and evaluated. One uses the computation time quantile configured from the start. The other estimates the quantile from the computation times of completed jobs. In this case, c_ϕ of the JAMS is updated according to the P^2 algorithm [42] when jobs are completed in the servers. The quantile estimate will be the computation time value where the proportion ϕ of observed computation times are below c_ϕ.

Example 3. *Let us consider Example 1, scheduled by a JAMS with* $Q=15$, $P=20$ *and* $n=2$ *as illustrated in Fig. 3. Let* $c^\uparrow = 40$ *and require* $\phi=0.95$ *of started jobs to meet their deadlines.*

For this example, the computation time quantile is $c_\phi=38$. If the guaranteed computation time prior to a job's deadline is at least 38 it will be accepted, otherwise it will be left on the queue.

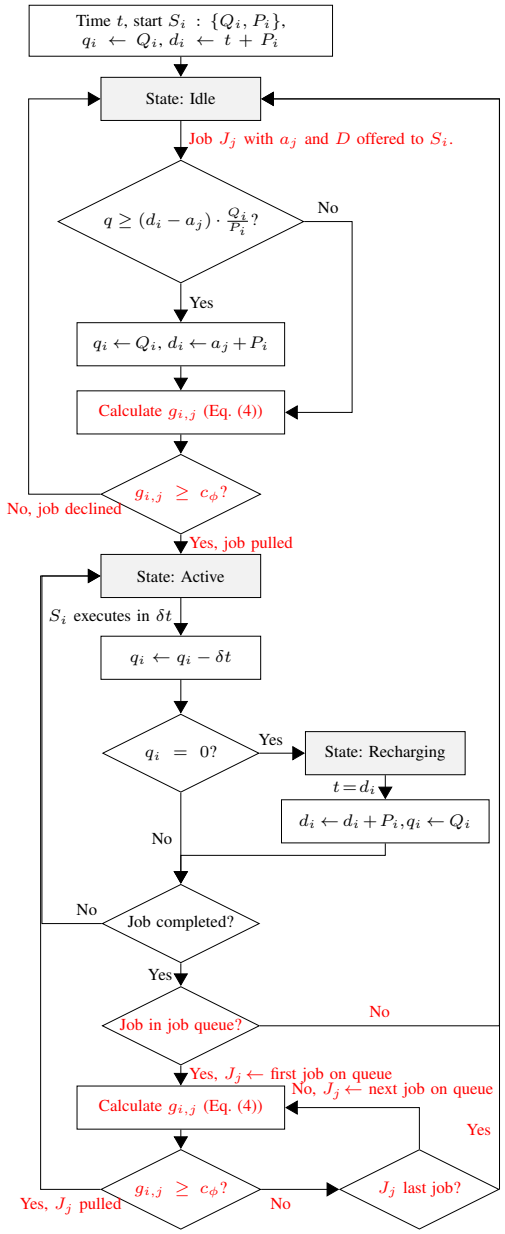

Fig. 4. Flowchart of a JAMS CBS.

Example 4. *Let us again consider Example 1, scheduled by a JAMS with $Q = 15$, $P = 20$ and $n = 2$ as illustrated in Fig. 3. Let $c^{\uparrow} = 40$ and require $\phi = 0.8$ of started jobs to meet their deadlines.*

For this example, the computation time quantile is $c_\phi = 20$. If the guaranteed computation time prior to a job's deadline is at least 20, it will be started. Otherwise, it will be left in the queue.

B. Maximum Job Queue Length

Since jobs at the front of the queue that cannot be provided sufficient computational resources prior to their deadline by any server are dismissed when a new job arrives, the maximum number of queued jobs can never exceed the number of jobs arriving during the task's relative deadline D. With the maximum number of jobs arriving instantaneously κ, and the minimum separation time of subsequent job arrival instants p, this means that the maximum job queue length is bounded by $\kappa \cdot \left\lceil \frac{D}{p} \right\rceil$.

C. Job Queue Waiting Time

We want to bound the waiting time of a job J^* that arrives at the JAMS. Denote the remaining work of the at most n jobs executing in the servers at the arrival of J^* with the random variable $\mathcal{B}$, and the work of all queued jobs at the arrival with the random variable $\mathcal{L}$. The waiting time of J^* is bounded in Theorem 1.

Theorem 1. *The waiting time $\mathcal{W}$ of a job J^* that arrives at a JAMS where $\mathcal{B}$ is the remaining work of currently running jobs, and $\mathcal{L}$ is the total work of jobs on the queue at the arrival of J^* is bounded by Eq.* (3).

$$\mathcal{W} \leq P \cdot \left\lceil \frac{\mathcal{B}+\mathcal{L}}{n \cdot Q} \right\rceil \tag{3}$$

Proof. The mean work to complete in a server before starting J^* is $\frac{\mathcal{B}+\mathcal{L}}{n}$, and this is completed latest at $P \cdot \left\lceil \frac{\mathcal{B}+\mathcal{L}}{n \cdot Q} \right\rceil$. At the latest at this point at least one server is available to start executing the work of J^*. □

D. Guaranteed Computation Time

When a server S_i requests to pull a job from the queue at time t, the guaranteed computation time $g_{i,j}$ provided by S_i prior to the deadline of J_j at the front of the queue is calculated. To simplify these expressions we denote the difference between J_j's deadline and the server's current deadline by $\delta_{i,j} = a_j + D - d_i$. If $\delta_{i,j} < 0$, then a part of the leftover budget q_i of the current server activation can be guaranteed. The server's execution ends latest at $t + U_{i\Sigma} \cdot (d_i - t)$, considering the total utilization $U_{i\Sigma}$ on the processor. If $\delta_{i,j} \geq 0$, the full leftover budget q_i of the current activation is guaranteed, plus Q units for each full server period after d_i before J_j's deadline, and possibly a part of the budget in the last server period before the job's deadline. The guaranteed computation time $g_{i,j}$ can be computed as:

$$g_{i,j} = \begin{cases} \left[q_i - [U_{i\Sigma} \cdot (d_i - t) - (a_j + D - t)]^+ \right]^+, & \delta_{i,j} < 0, \\ q_i + Q \cdot \left\lfloor \frac{\delta_{i,j}}{P} \right\rfloor + \\ \quad + \left[Q - [U_{i\Sigma} \cdot P - \delta_{i,j} \mod P]^+ \right]^+, & \delta_{i,j} \geq 0, \end{cases} \tag{4}$$

where $[x]^+ := \max(x, 0)$.

For analysis purposes, we bound $\mathcal{G}$ in Theorem 2.

Theorem 2. *A job J^* that arrives to a JAMS with remaining work on the servers $\mathcal{B}$ and queued work $\mathcal{L}$ will be guaranteed at least $\mathcal{G}$ computation resource prior to its deadline as outlined in Eq.* (5)

$$\mathcal{G} \geq Q \cdot \left\lfloor \frac{D}{P} \right\rfloor - Q \cdot \left\lceil \frac{\mathcal{B}+\mathcal{L}}{n \cdot Q} \right\rceil \tag{5}$$

Proof. The waiting time $\mathcal{W}$ of J^* is bounded in Eq. (3). The time remaining until the deadline when the job is pulled by a CBS is $D-\mathcal{W}$. The guaranteed computation time of a server within this time is at least Q times the number of full server periods in $D-\mathcal{W}$, giving:

$$\mathcal{G} \geq Q \cdot \left\lfloor \frac{D-\mathcal{W}}{P} \right\rfloor \geq Q \cdot \left\lfloor \frac{D}{P} \right\rfloor - Q \cdot \left\lfloor \frac{P \cdot \left\lceil \frac{\mathcal{B}+\mathcal{L}}{n \cdot Q} \right\rceil}{P} \right\rfloor$$
$$= Q \cdot \left\lfloor \frac{D}{P} \right\rfloor - Q \cdot \left\lceil \frac{\mathcal{B}+\mathcal{L}}{n \cdot Q} \right\rceil \qquad \square$$

E. Dismissal Probability

A job J_j will be dismissed if $g_{i,j} < c_\phi, \forall i$. Therefore the probability of dismissing a job is bounded in Eq. (6), by inserting the bound on the guaranteed computation time in Eq. (5), and in the last step using the bound on the served work $\mathcal{B} \leq n \cdot c^{\uparrow}$.

$$\begin{aligned} \mathbb{P}[\mathcal{G} < c_\phi] &\leq \mathbb{P}\left[Q \cdot \left\lfloor \frac{D}{P} \right\rfloor - Q \cdot \left\lceil \frac{\mathcal{B}+\mathcal{L}}{n \cdot Q} \right\rceil < c_\phi \right] \\ &= \mathbb{P}\left[\left\lceil \frac{\mathcal{B}+\mathcal{L}}{n \cdot Q} \right\rceil > \left\lfloor \frac{D}{P} \right\rfloor - \frac{c_\phi}{Q} \right] \\ &\leq \mathbb{P}\left[\left\lceil \frac{\mathcal{L}}{n \cdot Q} \right\rceil > \left\lfloor \frac{D}{P} \right\rfloor - \frac{c_\phi}{Q} - \left\lceil \frac{c^{\uparrow}}{Q} \right\rceil \right] \end{aligned} \tag{6}$$

1) Dismissals in an Overload Scenario: Consider a scenario where the total work on the queue remains high for a sufficient amount of time so that all servers are running at full capacity, and some jobs are being dismissed because of insufficient guaranteed computation time prior to their deadline. At this point, the average work completion rate by the servers is $n \cdot \frac{Q}{P}$. Let the average work arrival rate during the overload scenario be $\gamma \cdot n \cdot \frac{Q}{P}, \gamma > 1$. This implies that the proportion of dismissed work is $1-\gamma^{-1}$, consistent with steady-state queuing theory analysis of an overload system with abandonment [36]. The dismissal decision for a job is independent of the computation time of the specific job. If computation times are independent random variables, the proportion of dismissed jobs is also $1-\gamma^{-1}$. The dismissal probability of a specific job may be higher or lower due to variations in the work arrival rate and is bounded by Eq. (6).

We consider Example 2. In this case, computation times are i.i.d. and the required average computational requirement is 30.8 per job arrival, and the provided resource is 30. This gives $\gamma = \frac{30.8}{30} \approx 1.027$ and a dismissal rate of about 2.6%.

2) Dismissals With Sufficient Average Capacity: For a system with sufficient average capacity, a bound on the dismissal probability of a job in JAMS is outlined in Theorem 3.

Theorem 3. *For a JAMS that is not overloaded in any intervals longer than Δ_L, the probability p_d that a job is dismissed is bounded by Eq.* (7)*.*

$$p_d \leq \max_{\Delta} \left(\mathbb{P}\left[\left\lceil \frac{\Upsilon(\Delta)+\mathcal{B}}{n \cdot Q} \right\rceil > \frac{\Delta+D}{P} - \left\lceil \frac{c_\phi}{Q} \right\rceil \right] \right) \tag{7}$$

Proof. A system that is not overloaded in any long interval has some points in time when the queue is empty, and at least one server is ready to start executing an arriving job immediately. Denote a time when the last remaining idle server goes to active state with t_0. Now we consider a job J^* arriving at $t_0+\Delta, \Delta > 0$, assuming that all servers are busy in the interval $[t_0, t_0+\Delta]$. J^* will start at the latest when one server is no longer processing the remaining work of at most $n-1$ jobs that were running at t_0 or work that arrived in the interval except for the work of J^*. If J^* starts before $t_0+\Delta+D-P \cdot \left\lceil \frac{c_\phi}{Q} \right\rceil$, it will not be dismissed.

Denote the remaining work of jobs running at t_0 as $\mathcal{B}$ and the work arriving in this interval except for the work of J^* as $\Upsilon(\Delta)$. J^* will not be dismissed if:

$$t_0 + P \cdot \left\lceil \frac{\Upsilon(\Delta)+\mathcal{B}}{n \cdot Q} \right\rceil \leq t_0 + \Delta + D - P \cdot \left\lceil \frac{c_\phi}{Q} \right\rceil \tag{8}$$

Restructuring of this condition gives an upper bound on the probability that J^* is dismissed as:

$$\mathbb{P}\left[\left\lceil \frac{\Upsilon(\Delta)+\mathcal{B}}{n \cdot Q} \right\rceil > \frac{\Delta+D}{P} - \left\lceil \frac{c_\phi}{Q} \right\rceil \right] \tag{9}$$

For an arbitrary job J_j, if it arrives to a JAMS with at least one server idle, it will not be dismissed due to Assumption 2, and the bound is trivial. If it arrives to a JAMS with all servers active, there exists a Δ to the most recent point when the last server went to active state. Then Eq. (9) bounds the dismissal probability of J_j with this Δ. Eq. (7) is greater than or equal to Eq. (9), so it bounds the dismissal probability of J_j. $\square$

A task with average computation time requirement below that provided by the servers will experience no dismissals if for each interval length Δ, the arriving work $\Upsilon(\Delta)$ and the work $\mathcal{B}$ remaining to be served at the start of the interval when the last server goes to active are bounded by Eq. (8). For a system where $D > P \cdot \left(\left\lceil \frac{c^{\uparrow}}{Q} \right\rceil + \left\lceil \frac{c_\phi}{Q} \right\rceil \right)$ and work arrival rate bound $\rho \leq \frac{n \cdot Q}{P}$ we have dismissal probability at most ϵ in Eq. (9) for all $\Delta \geq \Delta_B$ from Assumption 1.

As Δ grows, Eq. (8) goes toward the condition that the provided computational resource needs to be higher than the average demand. $\Upsilon(\Delta) + \mathcal{B}$ in Eqs. (7) to (9) is a sum of computation time random variables. If computation times are independent random variables, the sum can be calculated by convolution. If computation times are correlated, convolution of upper bounding probabilistic WCET (pWCET) distributions can be applied [43], or a bound can be derived in other ways [44], [45]. If computation times are described by Markov Models, these can be utilized to calculate the sum [46], [47].

Let us go back to Example 3. We need to consider Δ as multiples of 20, as jobs are released with $p = 20$. For $\Delta = 20$, we have a bound on the dismissal probability as $\mathbb{P}\left[\left\lceil \frac{\Upsilon(20)+\mathcal{B}}{2 \cdot 15} \right\rceil > \frac{20+60}{20} - \left\lceil \frac{38}{15} \right\rceil \right] = \mathbb{P}\left[\left\lceil \frac{\Upsilon(20)+\mathcal{B}}{30} \right\rceil > 1 \right]$. In this case $\mathcal{B}$ may contain almost the full work of 1 job, if the queue was empty just a short time prior to t_0. $\Upsilon(20)$ contains 1 job, that arrived at t_0. It is clear that with this task and configuration, we cannot guarantee that a job arriving one period after the start of a queue is not dismissed. With computation times 20 or 38, two jobs will not fit in the total budget of 30.

For $\Delta = 40$ and $D = 60$, $\Upsilon(40)$ contains two jobs and the dismissal probability bound is $\mathbb{P}\left[\left\lceil\frac{\Upsilon(40)+\mathcal{B}}{30}\right\rceil > 2\right]$. All three jobs in $\mathcal{B}+\Upsilon(40)$ need to have computation time 20 to ensure no dismissal, and the probability of this is 0.9^3. The dismissal probability bound for jobs arriving within two periods from t_0 is $1-0.9^3 = 0.271$. For $\Delta = 60$, we need to consider a total of 4 jobs in $\mathcal{B}+\Upsilon(60)$. For $D = 60$, we have $\mathbb{P}\left[\left\lceil\frac{\Upsilon(60)+\mathcal{B}}{30}\right\rceil > 3\right]$, and all four jobs need to have computation time 20 to ensure no dismissal, the probability of this is about 0.34.

Considering the lower computation time quantile in Example 4, this will lead to lower dismissal probability bounds, as the term $\lceil\frac{38}{15}\rceil$ is replaced by $\lceil\frac{20}{15}\rceil$.

Example 5. *Let us consider Example 3, but now with a longer relative deadline of* 100*.*

With the relative deadline of 100, we would instead have the dismissal probability bound for $\Delta = 20$ as $\mathbb{P}\left[\left\lceil\frac{\Upsilon(20)+\mathcal{B}}{2\cdot 15}\right\rceil > \frac{20+100}{20} - \lceil\frac{38}{15}\rceil\right] = \mathbb{P}\left[\left\lceil\frac{\Upsilon(20)+\mathcal{B}}{30}\right\rceil > 3\right]$. We see that even in the worst case where both jobs contributing work to $\Upsilon(20)+\mathcal{B}$ have computation time 38, we can guarantee that there is no dismissal at this point.

With $\Delta = 40$, we have $\mathbb{P}\left[\left\lceil\frac{\Upsilon(40)+\mathcal{B}}{30}\right\rceil > 4\right]$, and the dismissal probability is 0.

With $\Delta = 60$, $\mathbb{P}\left[\left\lceil\frac{\Upsilon(60)+\mathcal{B}}{30}\right\rceil > 5\right]$ at least one of the jobs needs to be short to avoid a dismissal, so the dismissal probability bound equals the probability of all jobs being long, 0.1^4.

F. Probability of Meeting the Deadline

We show that for a job that is accepted by a server, the deadline will be met with at least probability ϕ, provided the probability is at least ϕ that the job's computation times is below c_ϕ. For independent computation times, ϕ represents a bound on the probability that $\mathcal{C} < c_\phi$ for each job that is run. For correlated computation times, individual jobs may have higher or lower probability of exceeding the quantile. In this case the average ratio of started jobs that meet their deadline to started jobs is at least ϕ.

Theorem 4. *Assuming i.i.d. computation times, a job J_j that is accepted by a CBS in JAMS has at least probability ϕ of meeting its deadline.*

Proof. A job that is accepted is guaranteed at least c_ϕ computation time prior to its deadline. From this if follows that $\mathbb{P}[\mathcal{R} \le D] \ge \mathbb{P}[\mathcal{C} \le c_\phi] \ge \phi$. □

Theorem 5. *The rate of jobs accepted by a CBS in JAMS that meet their deadline is at least ϕ.*

Proof. A job J_j that is accepted is guaranteed at least c_ϕ computation time prior to its deadline. The outcome of the acceptance test is independent of the computation time of J_j. Therefore, the rate of accepted jobs that meet their deadlines is at least the rate of jobs with computation time below c_ϕ, that is ϕ. □

We note that using a precomputed computation time quantile derived from a pWCET distribution that upper bounds the computation time distribution ensures that the probability bound on meeting the deadline holds for each accepted job even with correlation.

A general bound on the probability of meeting the deadline, that holds even for jobs with computation time $c_j > c_\phi$ is derived in Eq. (10).

$$\begin{aligned}\mathbb{P}[\mathcal{C} \le \mathcal{G}] &\ge \mathbb{P}\left[\mathcal{C} \le Q\cdot\left\lfloor\frac{D}{P}\right\rfloor - Q\cdot\left\lceil\frac{\mathcal{B}+\mathcal{L}}{n\cdot Q}\right\rceil\right] \\ &= \mathbb{P}\left[\left\lceil\frac{\mathcal{B}+\mathcal{L}}{n\cdot Q}\right\rceil + \frac{\mathcal{C}}{Q} \le \left\lfloor\frac{D}{P}\right\rfloor\right]\end{aligned} \tag{10}$$

We return to the comparison of Examples 3 and 4. We have seen that a lower computation time quantile leads to lower dismissal probability, but this comes at the price of a higher probability of deadline miss for started jobs.

Finally, we note that an upper bound on the dismissal probability p_d, and a lower bound on the probability that an accepted job meets its deadline ϕ, imply a lower bound on the probability of meeting the deadline for every job as $(1-p_d)\cdot\phi$.

VI. Implementation and Overheads

The mechanism described in Section V has been implemented as a multi-threaded C program on the Linux Operating System, managing a limited-size shared queue that exposes special blocking operations to push jobs into the queue and pull admitted jobs out of it, alongside performing the actual dismissal operations described in Section V. Our JAMS implementation may optionally use a kernel module we realized for faster and more accurate access to the `SCHED_DEADLINE` state parameters at runtime, as described below.

A. JAMS Push and Pull Operations

The push operation is a simple blocking operation that pushes jobs into a FIFO-ordered queue, managed through a classical circular buffer. In the JAMS design, jobs are pushed all with the same relative deadline; thus, their submission order corresponds to a deadline-based order. In the experimentation described below, we configured the queue size never to hit the size limit in a push operation. After enqueuing a job for processing, the push operation notifies other threads possibly waiting on a pull through a condition variable. The pull operation is more involved and contains the majority of our JAMS implementation: we retrieve the runtime left and absolute deadline of the CBS server (see below), then we scan the jobs waiting in the queue, starting from the earliest submitted job, checking whether the budget available till the deadline is sufficient for the estimated percentile of the job computation time, using Eq. (2): in such a case, the job is pulled out of the queue and returned to the caller for processing, otherwise we move forward to check the next job in the queue. In the latter case, the job is not immediately dismissed but left for the other threads in the JAMS CBS group, which will, in turn, evaluate the job based on their own CBS instantaneous parameters. A job that is not picked

up by any server is eventually dismissed. Whenever the pull operation cannot find admissible jobs to process in the queue, it blocks on a condition variable, waiting for a push operation.

B. Reading SCHED_DEADLINE Parameters

The SCHED_DEADLINE scheduler available on Linux has a direct API to read the statically configured maximum runtime, relative deadline, and period of a CBS server (through the sched_getattr() syscall), but it lacks an API to retrieve the leftover runtime and absolute deadline. However, these parameters can be accessed through the /proc filesystem, reading the per-task sched special file, where the dl.runtime and dl.deadline parameters can be found for tasks under the SCHED_DEADLINE policy. The runtime value accessible this way is only guaranteed to be updated at each periodic system tick, with a frequency of HZ. Reconfiguring the kernel with a 1ms HZ results in a more accurate reading. However, the /proc interface is designed for debugging rather than for production use, so it suffers from inherent inefficiency. For example, all quantities are wastefully formatted in decimal notation from the kernel space and have to be converted back into user space. Therefore, we realized a kernel module that allows for accurately reading the SCHED_DEADLINE parameters directly in binary format, with much greater efficiency. On the Raspberry Pi 4 board used for our experimentation, reading the current parameters using the /proc interface resulted in a $228 \pm 35 \mu s$ per-reading overhead, while with our kernel module, this was reduced to a $10.8 \pm 2.4 \mu s$ per-reading overhead. Note that, in JAMS, worker threads need to retrieve this information each time a job is evaluated for processing to apply the acceptance policy properly.

C. JAMS Overheads

This subsection discusses the computational and memory overheads due to using JAMS, in its current implementation. The time required for JAMS push and pull operations is bounded by a constant time per call. The most significant overhead is observed during 'pull' when reading the server parameters, for which a significant improvement is discussed in Section VI-B (reducing it from $228 \mu s$ down to $10.8 \mu s$). Regarding how often we pull, in a case where JAMS is mainly idle, nearly every server will wake up and attempt a pull at each job arrival. When JAMS servers are mainly busy, each server will perform a pull per processed job, roughly with a rate equal to the arrival rate divided by the number of servers. With our implementation and evaluation set-up, the average pull time with the improvement discussed in Section VI-B is $49 \mu s$, and the average push time is $9 \mu s$. In our evaluation scenarios, job computation times are $10 - 300 ms$, making those overheads quite negligible from a practical standpoint. Regarding the memory overhead, arrival times must be stored for all queued jobs.

VII. Experimental Evaluation

In this section, we provide results from the experimental evaluation[1] of our proposed JAMS mechanism, implemented as detailed in Section VI. The behavior of the proposed JAMS and dismissal mechanism is evaluated in the presence of both synthetic and realistic workload scenarios. We compare the deadline miss rate and job dismissal rate for different configurations and workloads. We also show how application of the JAMS acceptance policy affects the response times throughout an execution sequence. A comparison is performed between applying the JAMS dismissal policy with a statically precomputed quantile of the computation times versus using an online estimated quantile. Comparisons are performed against a baseline with multiple servers that always pull the first available job from the queue without applying any dismissal policy.

Two types of workload are considered in the evaluation: synthetically generated computation times with a lognormal distribution and recorded computation times from an MPC task.

A. Computation Time Data

The MPC task is based on the Unmanned Ground Vehicle (UGV) path planning with obstacles example of the libmpc++ library [48]. The Sequential Quadratic Programming algorithm SLSQP [49] from the NLopt library [50] is used for the optimization. Predictions and control are computed over a 6-step horizon. A list of waypoints is used, and when a waypoint is reached, the next is obtained from the list in a circular manner. The simulated UGV is run in two environments, with smaller or larger obstacles, resulting in lighter or heavier computational load. The starting point, waypoint list, and configuration of the optimization are the same. The libmpc++ optimization time logging is modified to use the clock_gettime(CLOCK_THREAD_CPUTIME_ID) syscall, and log the computation time of the optimization rather than the response time.

10 runs of 5000 optimization steps are performed in each environment. The computation time data collection is performed on a Raspberry Pi 3B+ with PREEMPT_RT where frequency scaling and USB have been disabled. The task is scheduled with the highest priority FIFO scheduling and pinned to a core with the cpuset utility. Computation times are retrieved from the libmpc++ log files. The experimental Cumulative Distribution Function (CDF) of the MPC computation times are reported in Fig. 5. There is a high degree of correlation among the computation times, since they are affected by the UGV state dynamics.

We also used synthetic computation times that have been generated with lognormal distributions. These have been shown to be a good fit for computation time distributions in some applications [51]. We have generated 10 lognormal traces, drawing the average randomly from the range $[40, 60]$ms and the standard deviation from the range $[30,40]$ms. The distribution is bounded to the range $[10,160]$ by discarding samples outside this range. The synthetically generated computation time CDFs are reported in Fig. 6. Means and standard deviations shown in the figure are empirical from the samples in the bounded range, with the exception of the last dotted line. In this case these are the mean and standard deviation of the lognormal distribution prior to bounding the range.

[1] Code and data for the artifact evaluation are available at https://github.com/annafriebe/RTAS_25_JAMS_AE.

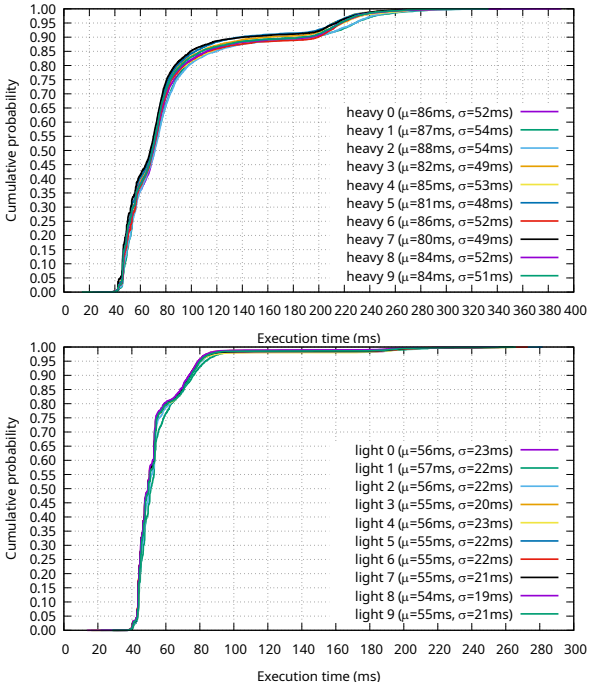

Fig. 5. Experimental CDFs of computation times for the heavier (top) and lighter (bottom) MPC traces.

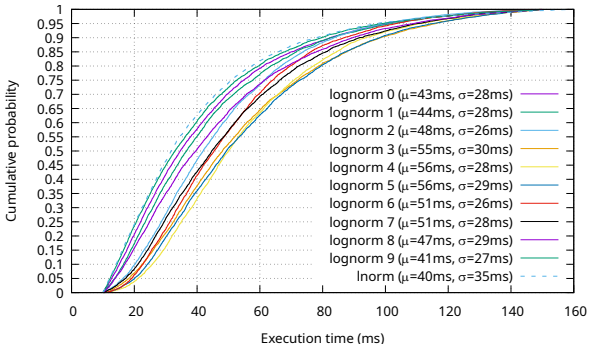

Fig. 6. Experimental CDFs of synthetic i.i.d. computation times generated from lognormal distributions (applying a truncation of the distributions in the [10,160]ms range).

B. Evaluation Program and Test-Bed Setup

The described JAMS mechanism has been used in the program we realized for the experimental validation, where one thread was dedicated to submitting jobs to JAMS via the push operation, while n worker threads used the pull operation. Each thread was attached to a `SCHED_DEADLINE` reservation with configurable `dl_runtime` and `dl_deadline = dl_period` parameters. Reservations were configured in partitioned EDF mode[2]. Job computation times were provided as trace files input to the program, produced according to the workload scenarios described in Section VII-A. The thread submitting jobs to the JAMS has been periodically activated, with a specified job inter-arrival period. The worker threads have been using the special pull operation to retrieve the admitted jobs and then process them by performing wasteful computations for the amount of time of each job, as instructed by the trace file that was read at the beginning of the program. Finally, they measured the response time for the job, storing it in an in-memory array. At the end of the program, all the stored response times have been dumped into an output file. In all experiments, jobs are released periodically every 80ms, equal to the server period P.

The experiments have been run on a Raspberry Pi 4 Model B board equipped with an Arm Cortex-A72 quad-core CPU and 3.7GiB of RAM, with the CPU frequency locked at the maximum value via `cpufreq`. The JAMS was configured with 2 worker threads pinned down on cores 2 and 3, and the thread submitting requests pinned down on core 1. The 3 cores have been isolated from the general OS workload using the `isolcpus` boot parameter of the kernel. Additionally, all experiments were carried out at runlevel 1 to avoid starting unnecessary services on the platform.

C. Experiment 1 – MPC Traces

In the first experiment, we performed a program run using the 10 traces from the MPC use-case whose distribution is reported in Fig. 5. We have reported the obtained deadline miss percentage and dismissed jobs percentage in Fig. 7. The relative task deadline is 480 ms, chosen from the MPC 6-step horizon and the 80 ms period. The top plot shows the results obtained processing the 10 more demanding traces, tagged "heavy", using a per-CBS allocation bandwidth of 60%. When using no dismissal, the system turns out to be quite overloaded throughout the run, resulting in deadline miss rates between 30% and 60%, as visible from the "No dismissal" violet dots cloud at the bottom right of the plot. When enabling the JAMS acceptance policy configured statically with the known 95th percentile of the input traces, we obtain roughly 1% of deadline misses at the expense of a 10% of job dismissals (green orthogonal crosses cloud). The results are aligned with our deadline-miss theoretical expectations in Theorem 5, as the system would be supposed to guarantee at least a 95% of deadline hit rate under these conditions. Switching to using a dynamic percentile estimator, we obtain the green oblique crosses cloud, having a slightly lower dismissal rate at the expense of doubling the deadline miss ratio, which stays safely within the 5% design bound (highlighted as the green vertical dashed bar in the plot).

Moving to the plot's blue and brown dots cloud series, we can see a similar behavior obtained by configuring JAMS with a different target percentile, namely 90th and 85th (blue and brown series, respectively). In both cases, the obtained results confirm what just discussed above, with the difference that, when JAMS uses a lower computation time percentile in its configuration, it tends to admit a higher number of jobs, obtaining a higher percentage of deadline misses, which keeps staying safely below the theoretical bounds (10% and

[2]This was obtained via disabling the in-kernel access control by writing -1 to `sched_rt_runtime_us` in `/proc/sys/kernel`, then setting the needed affinity masks on SCHED_DEADLINE threads.

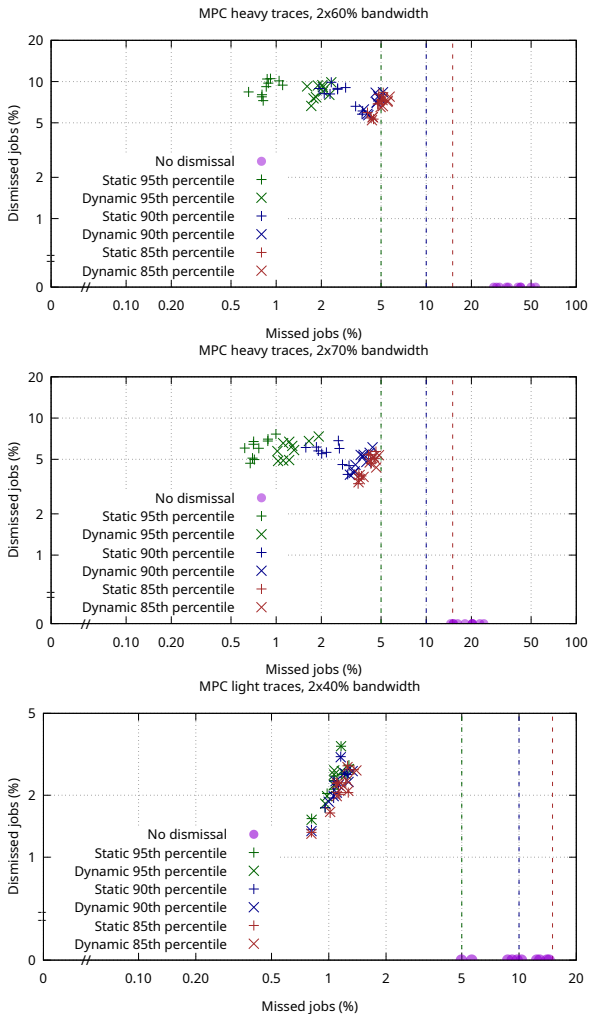

Fig. 7. Missed jobs (on the X axis) in % over the processed jobs, compared to the dismissed jobs % (on the Y axis), obtained with no dismissals baseline (violet dots), and JAMS with configurations of the percentile (various point colors), both when statically configured and dynamically estimated (orthogonal vs oblique crosses). Note that the X and Y axes are on a logarithmic scale, broken close to the origin so that 0 values can also be visualized.

15% marked with a blue and brown dashed vertical bar in the two plots).

To mitigate the need to dismiss jobs with a heavy workload, we can assign more computational resources to JAMS, increasing the per-CPU bandwidth from 60% to 70%. The effects of such a change on the various configurations are shown in the second plot of Fig. 7. In this case, the baseline runs exhibit significantly fewer deadline misses (reduced from the previous 30%-60% down to 15%-30%); however, they are still above the desirable threshold of 5%, for example. The JAMS mechanism, in this case, also proves to be beneficial, managing to keep the deadline-miss rate for the processed jobs within the design bounds (the usual three percentiles are shown in the picture), applying a dismissal rate roughly equal to half the one of the previous case with the per-CPU bandwidth of 60%.

The bottom plot in Fig. 7 shows the results obtained processing the 10 lighter traces, tagged "light", using a per-

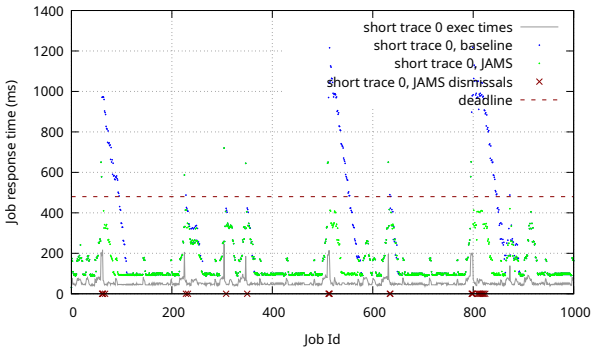

Fig. 8. Excerpt of computation times for the MPC lightweight trace 0 (gray line), and the per-job response times obtained using JAMS vs the baseline.

CBS allocated bandwidth of 40%. Under these conditions, our baseline without dismissals results in a deadline-miss rate between 5% and 16%, visible in the "No dismissals" violet dots at the bottom right of the plot. Enabling our JAMS dismissal policy, with a configured 95% of target deadline-miss bound, we obtain deadline-miss rates between 0.5% and 1.5% (again safely below the design bound), at the expense of a dismissal rate between 1.5% and 3.5%.

In the collected experimental data, a final piece of information worth a look at is the number of consecutive jobs that miss their deadline or are dismissed. Fig. 8 reports the response times obtained for the first 1000 jobs in the experiment using the MPC light trace 0, without any dismissal policy (blue dots), versus using JAMS with a statically configured 95% percentile (red dots), where job dismissals are highlighted by red crosses. It is clear that without a dismissal policy, transient overloads lead to long periods with no jobs producing timely results.

D. Experiment 2 – MPC Traces With Real-Time Load

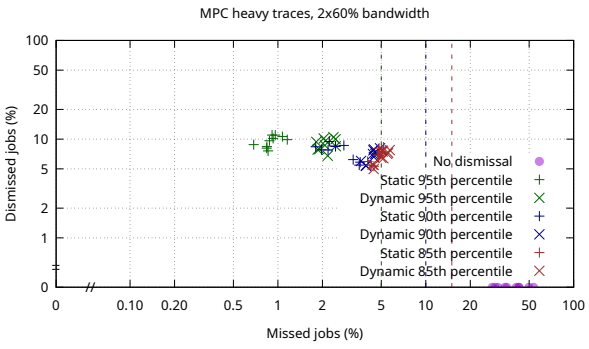

Fig. 9. Missed jobs (X axis) in % over the processed jobs, compared to the dismissed jobs % (Y axis). The board was hosting an additional 15% of per-CPU real-time reservations, spread across 3 real-time tasks on each CPU.

We also experimented with the JAMS mechanism running alongside other real-time workload on the platform. As JAMS is designed around using CBS and partitioned EDF for temporal isolation among multiple real-time tasks, the additional load on the platform was also run through CBS reservations. More specifically, we performed a run of the

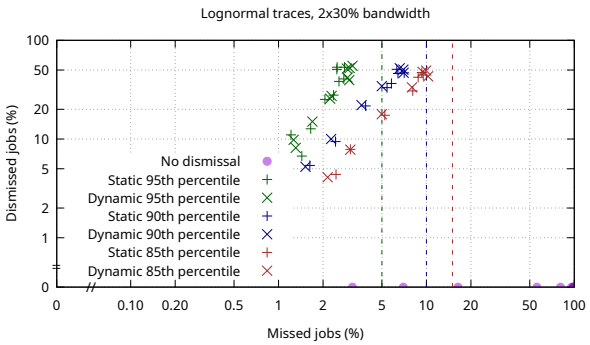

Fig. 10. Missed jobs (on the X axis) in % over the processed jobs, compared to the dismissed jobs % (on the Y axis).

heavy traces described above, served by 2 CBS servers occupying 60% of the CPUs 2 and 3, deploying on each of these 2 CPUs also additional 3 real-time tasks, attached to CBS reservations with runtime and period=deadline of (3ms,60ms), (5ms,100ms) and (7ms,140ms), for a total of additional 15% of real-time CPU workload.

JAMS was made aware of the total utilization of the CPUs where its CBS servers were deployed, so to correctly compute the available budget to deadline $g_{i,j}$ in Eq. (4). The obtained results are summarized in Fig. 9, where, alongside the usual "No dismissal" points visible at the bottom right of the plot, we can see the results from JAMS configured with 95th, 90th and 85th percentiles of the computation times distributions (in green, blue and brown dots, respectively). Results are reported with statically configured percentiles (orthogonal crosses) and dynamically estimated ones (oblique crosses). The expected percentile bounds on the deadline misses are represented with vertical dashed lines with the same color as the points corresponding to that configured percentile (green, blue, and brown for 95th, 90th, and 85th, respectively).

E. Experiment 3 - Lognormal I.I.D. Traces

We performed another experiment using the lognormal traces shown in Fig. 6. In this case, we used JAMS configured with 2 threads, 30% of per-CBS bandwidth, a relative task deadline of 480 ms, and the usual three percentile configurations. The obtained results are shown in Fig. 10. In this case, the lognormal traces were generated using different parameters drawn at random, as described earlier. Therefore, with the configured CBS bandwidth assignment, we obtained a wide range of different results, evident from the "No dismissal" violet dots scattered throughout the X axis, from 2.5% to 100% of deadline miss rates. Interestingly, applying the JAMS mechanism to these cases, we consistently obtain a deadline-miss ratio below the configured target percentile (95th, 90th, and 85th, highlighted as dashed vertical lines as usual), obtained at the cost of applying a dismissal rate that also varies greatly across the cases, from 2% to nearly 50% of dismissals.

We also performed a run designed to validate the dismissal probability bound in Theorem 3. This was done by running our experimentation using the last trace shown as a dashed curve in Fig. 6, with JAMS configured as usual with 2 CBS servers with a 60% bandwidth and a job deadline of 240ms. Over the run, we experienced 0.04% of job dismissals. The theoretical dismissal rate bound from Eq. (6) is derived, calculating $\mathcal{B}+\mathcal{L}$ as a convolution of an increasing number of bounded lognormal distributions for several Δ as multiples of 80ms. The dismissal probability bound is 1.2%, observed for the shortest Δ.

VIII. Conclusion and Future Work

This paper introduced the Job Acceptance Multi-Server (JAMS) mechanism as an extension of the Constant Bandwidth Server (CBS) for efficient resource allocation in real-time systems. By leveraging a shared job queue across multiple CBS instances, JAMS enables dynamic resource sharing among tasks with varying computational demands and deadlines. Its targeted job acceptance and dismissal strategy prioritizes jobs with a high likelihood of meeting their deadline, limiting the impact of job tardiness and overloading. Experimental evaluation with synthetic and realistic workloads showed that JAMS significantly reduces the deadline miss rate compared to a baseline system that lacks a dismissal policy. More specifically, JAMS consistently meets design-bound deadline-miss ratios while managing resource demands through adaptive dismissal rates, showing robust performance under several diverse workload conditions.

In the future, we plan to extend the present work in various directions. On the analysis side, the theoretical bounds on dismissal probability presented in the paper may be pessimistic in certain scenarios. Refining these bounds to achieve tighter estimates would improve the applicability of the approach in a broader set of scenarios. From an experimental perspective, more extensive experimentation is needed to assess possible scalability limitations of the proposed technique in the presence of several CBS servers pulling from the same shared queue. Albeit JAMS is proposed in the context of real-time embedded platforms, where the number of available CPUs might be limited, future embedded platforms seem to trend into featuring dozens of cores easily. Therefore, further testing on diverse architectures, including distributed and cloud/edge-based real-time systems, would provide insights into JAMS's scalability and resilience in a wide range of use-cases, such as real-time cloud computing [52]–[55]. Finally, on the implementation side, our current JAMS prototype can certainly be improved by moving the queue management logic into kernel space. This would have the advantage of being able to easily access the CBS scheduling parameters of the `SCHED_DEADLINE` threads participating in a JAMS queue, with the ability to wake them up only when jobs can certainly be accepted.

References

[1] X. Yang and L. T. Biegler, "Advanced-multi-step nonlinear model predictive control," *Journal of process control*, vol. 23, no. 8, pp. 1116–1128, 2013.

[2] M. Abouzahir, A. Elouardi, S. Bouaziz, R. Latif, and A. Tajer, "FastSLAM 2.0 running on a low-cost embedded architecture," in *2014 13th International Conference on Control Automation Robotics & Vision (ICARCV)*. IEEE, 2014, pp. 1421–1426.

[3] F. Gustafsson, "Particle filter theory and practice with positioning applications," *IEEE Aerospace and Electronic Systems Magazine*, vol. 25, no. 7, pp. 53–82, 2010.

[4] D. Fox, "Adapting the sample size in particle filters through KLD-sampling," *The international Journal of robotics research*, vol. 22, no. 12, pp. 985–1003, 2003.

[5] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*. IEEE, 1998, pp. 4–13.

[6] S. Baruah, J. Goossens, and G. Lipari, "Implementing constant-bandwidth servers upon multiprocessor platforms," in *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2002, pp. 154–163.

[7] Ł. Kruk, J. Lehoczky, K. Ramanan, S. Shreve *et al.*, "Heavy traffic analysis for EDF queues with reneging," *The Annals of Applied Probability*, vol. 21, no. 2, pp. 484–545, 2011.

[8] S. Manolache, P. Eles, and Z. Peng, "Schedulability analysis of applications with stochastic task execution times," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 4, pp. 706–735, 2004.

[9] S. Natarajan and D. Broman, "Timed C: An extension to the C programming language for real-time systems," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018, pp. 227–239.

[10] T. Cucinotta and D. Faggioli, "An exception based approach to timing constraints violations in real-time and multimedia applications," in *International Symposium on Industrial Embedded System (SIES)*, July 2010, pp. 136–145.

[11] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *2010 31st IEEE Real-Time Systems Symposium*. IEEE, 2010, pp. 259–268.

[12] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," *Real-Time Systems*, vol. 49, pp. 404–435, 2013.

[13] I. I. Lupu and J. Goossens, "Scheduling of hard real-time multi-thread periodic tasks," in *19th International Conference on Real-Time and Network Systems, RTNS '11, Nantes, France, September 29-30, 2011. Proceedings*, S. Faucou, A. Burns, and L. George, Eds., 2011, pp. 35–44. [Online]. Available: http://rtns2011.irccyn.ec-nantes.fr/files/rtns2011.pdf

[14] J. K. Ousterhout *et al.*, "Scheduling techniques for concurrent systems." in *ICDCS*, vol. 82, 1982, pp. 22–30.

[15] W. Ali and H. Yun, "RT-Gang: Real-time gang scheduling framework for safety-critical systems," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019, pp. 143–155.

[16] S. Ahmed and J. H. Anderson, "Soft real-time gang scheduling," in *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2023, pp. 331–343.

[17] A. Parri, M. Marinoni, J. Lelli, G. Lipari *et al.*, "An implementation of a multiprocessor bandwidth reservation mechanism for groups of tasks," in *Proceedings of the 16th Real Time Linux Workshop, OSADL, Ed., Dusseldorf, Germany*, 2014.

[18] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the linux kernel," *ACM SIGBED Review*, vol. 16, no. 3, pp. 33–38, 2019.

[19] J. L. Díaz, D. F. García, K. Kim, C.-G. Lee, L. L. Bello, J. M. López, S. L. Min, and O. Mirabella, "Stochastic analysis of periodic real-time systems," in *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.* IEEE, 2002, pp. 289–300.

[20] K. Zagalo, Y. Abdeddaïm, A. Bar-Hen, and L. Cucu-Grosjean, "Response time stochastic analysis for fixed-priority stable real-time systems," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 3–14, 2022.

[21] J.-J. Chen, M. Günzel, P. Bella, G. von der Brüggen, and K.-H. Chen, "Dawn of the dead (line misses): Impact of job dismiss on the deadline miss rate," *arXiv preprint arXiv:2401.15503*, 2024.

[22] P. Pazzaglia, C. Mandrioli, M. Maggio, and A. Cervin, "DMAC: Deadline-miss-aware control," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019, p. 1.

[23] A. Burns and R. I. Davis, "Mixed criticality systems-a review," Department of Computer Science, University of York, Tech. Rep., Feb. 2022.

[24] S. K. Baruah, A. Burns, and R. I. Davis, "Response-time analysis for mixed criticality systems," in *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE, 2011, pp. 34–43.

[25] H.-M. Huang, C. Gill, and C. Lu, "Implementation and evaluation of mixed-criticality scheduling approaches for sporadic tasks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, pp. 1–25, 2014.

[26] I. Bate, A. Burns, and R. I. Davis, "An enhanced bailout protocol for mixed criticality embedded software," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 298–320, 2016.

[27] A. Burns, R. I. Davis, S. Baruah, and I. Bate, "Robust mixed-criticality systems," *IEEE Transactions on Computers*, vol. 67, no. 10, pp. 1478–1491, 2018.

[28] A. Kritikakou, C. Pagetti, O. Baldellon, M. Roy, and C. Rochange, "Run-time control to increase task parallelism in mixed-critical systems," in *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 2014, pp. 119–128.

[29] G. C. Buttazzo, J. A. Stankovic *et al.*, "RED: Robust earliest deadline scheduling," in *Proc. of 3rd International Workshop on Resonsive Computing Systems*, 1993.

[30] M. Spuri, G. Buttazzo, and F. Sensini, "Robust aperiodic scheduling under dynamic priority systems," in *Proceedings 16th IEEE Real-Time Systems Symposium*. IEEE, 1995, pp. 210–219.

[31] L. Abeni and G. Buttazzo, "Resource reservation in dynamic real-time systems," *Real-Time Systems*, vol. 27, pp. 123–167, 2004.

[32] G. Bernat, A. Burns, and A. Liamosi, "Weakly hard real-time systems," *IEEE transactions on Computers*, vol. 50, no. 4, pp. 308–321, 2001.

[33] Z. Tong, S. Ahmed, and J. H. Anderson, "Holistically budgeting processing graphs," in *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2023, pp. 27–39.

[34] A. R. Ward, "Asymptotic analysis of queueing systems with reneging: A survey of results for FIFO, single class models," *Surveys in Operations Research and Management Science*, vol. 17, no. 1, pp. 1–14, 2012.

[35] D. Towsley and S. Panwar, *Optimality of the stochastic earliest deadline policy for the G/M/c queue serving customers with deadlines*. Citeseer, 1991.

[36] W. Whitt, "Fluid models for multiserver queues with abandonments," *Operations research*, vol. 54, no. 1, pp. 37–54, 2006.

[37] ——, "Time-varying queues," *Queueing models and service management*, vol. 1, no. 2, 2018.

[38] ——, "Stabilizing performance in a single-server queue with time-varying arrival rate," *Queueing Systems*, vol. 81, pp. 341–378, 2015.

[39] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on networking*, vol. 1, no. 4, pp. 397–413, 1993.

[40] K. Nichols and V. Jacobson, "Controlling queue delay," *Communications of the ACM*, vol. 55, no. 7, pp. 42–50, 2012.

[41] L. Abeni, G. Lipari, and J. Lelli, "Constant bandwidth server revisited," *Acm Sigbed Review*, vol. 11, no. 4, pp. 19–24, 2015.

[42] R. Jain and I. Chlamtac, "The P2 algorithm for dynamic calculation of quantiles and histograms without storing observations," *Communications of the ACM*, vol. 28, no. 10, pp. 1076–1085, 1985.

[43] R. I. Davis and L. Cucu-Grosjean, "A survey of probabilistic schedulability analysis techniques for Real-Time systems," *LITES: Leibniz Transactions on Embedded Systems*, pp. 1–53, 2019.

[44] F. Marković, P. Roux, S. Bozhko, A. V. Papadopoulos, and B. B. Brandenburg, "CTA: A correlation-tolerant analysis of the deadline-failure probability of dependent tasks," in *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2023, pp. 317–330.

[45] K.-H. Chen, N. Ueter, G. von der Brüggen, and J.-J. Chen, "Efficient computation of deadline-miss probability and potential pitfalls," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 896–901.

[46] B. V. Frias, L. Palopoli, L. Abeni, and D. Fontanelli, "Probabilistic real-time guarantees: There is life beyond the iid assumption (outstanding paper)," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2017, pp. 175–186.

[47] A. Friebe, F. Marković, A. V. Papadopoulos, and T. Nolte, "Efficiently bounding deadline miss probabilities of Markov chain real-time tasks," *Real-Time Systems*, pp. 1–48, 2024.

[48] N. Piccinelli, "Libmpc++: A library to solve linear and non-linear MPC," https://github.com/nicolapiccinelli/libmpc.

[49] D. Kraft, "Algorithm 733: TOMP–fortran modules for optimal control calculations," *ACM Transactions on Mathematical Software*, vol. 20, pp. 262–281, 1994.

[50] S. G. Johnson, "The NLopt nonlinear-optimization package," http://github.com/stevengj/nlopt.

[51] P. Skarin, "Control over the cloud: Offloading, elastic computing, and predictive control," Ph.D. dissertation, Lund University, 2021.

[52] M. García-Valls, T. Cucinotta, and C. Lu, "Challenges in real-time virtualization and predictable cloud computing," *Journal of Systems Architecture*, vol. 60, no. 9, pp. 726–740, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762114001015

[53] S. Xi, C. Li, C. Lu, C. D. Gill, M. Xu, L. T. Phan, I. Lee, and O. Sokolsky, "RT-Open Stack: CPU resource management for real-time cloud computing," in *2015 IEEE 8th International Conference on Cloud Computing*, June 2015, pp. 179–186.

[54] R. Andreoli, H. Gustafsson, L. Abeni, R. Mini, and T. Cucinotta, "Optimal deployment of cloud-native applications with fault-tolerance and time-critical end-to-end constraints," in *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing*, ser. UCC '23. ACM, Dec. 2023. [Online]. Available: http://dx.doi.org/10.1145/3603166.3632139

[55] V. Struhár, S. S. Craciunas, M. Ashjaei, M. Behnam, and A. V. Papadopoulos, "Hierarchical resource orchestration framework for real-time containers," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 1, Jan. 2024.