Dispatching Deadline Constrained Jobs in Edge Computing Systems

Shaik Mohammed Salman*, Alessandro Vittorio Papadopoulos*, Saad Mubeen*, and Thomas Nolte*
*Mälardalen University,, Västerås, Sweden

***Abstract*—The edge computing paradigm extends the architectural space of real-time systems by bringing the capabilities of the cloud to the edge. Unlike cloud-native systems designed for mean response times, real-time industrial embedded systems are designed to control a single physical system, such as a manipulator arm or a mobile robot, that requires temporal predictability. We consider the problem of dispatching and scheduling of jobs with deadlines that can be offloaded to the edge and propose DAL, a deadline-aware load balancing and scheduling framework that leverages the availability of on-demand computing resources along with an on-arrival dispatching scheme to manage temporal requirements of such offloaded applications. The evaluation indicates that DAL can achieve reasonably good performance even when execution times, arrival times, and deadlines vary.**

I. Introduction

Complex real-time systems such as mobile robots and industrial robots traditionally follow an embedded deployment model with software functions running on a shared hardware platform to control a single physical system such as the robot [1]. Several studies have investigated the benefits of a cloud-oriented design that depends on the elastic availability of resources for real-time applications [2], [3], [4], [5]. One approach to realize a cloud-oriented design is to package much of the application software, including the operating system, as a Virtual Machine (VM) or container and deploy it to the cloud or edge infrastructure [6], [7] where each VM or container is responsible for a single physical system. An alternative model is a microservice design, where individual functions have their own resources, such as CPU and memory, and handle requests from multiple physical systems rather than a single system. Several approaches have been proposed for scheduling microservices requests in such a deployment model, which take into account latency requirements but have limited consideration for issues such as satisfying deadline constraints [8], [9], [10]. Within queuing theory, several works have addressed deadline-based scheduling policies such as Earliest-Deadline-First (EDF) for systems with a single queue [11], [12], [13] and for the scenario where the number of available processors is fixed [14], [15], [16].

Additionally, several algorithms that can be offloaded exhibit variability in execution times [17], inter-arrival times [18] as well as in terms of their relative deadlines [19]. For example, while most real-time task models assume that the relative deadline of a real-time task is fixed, for some tasks this deadline may also vary per job [20] as well as during the execution of a job [19]. The inter-arrival time between successive jobs can also vary significantly [18], [21]. For systems such as mobile robots, the inter-arrival time can be a function of the current velocity and the distance they are expected to cover before new data is needed [22].

To address this, we consider an edge architecture that relies on the concept of on-demand processing commonly used in cloud-computing designs [23] and uses existing low-complexity dispatching and scheduling policies. Specifically, we present DAL, a deadline-aware load balancing and scheduling framework that integrates an on-arrival dispatcher along with an EDF scheduling policy by considering the availability of a processor pool with reserved processors and on-demand processors to reduce the number of jobs that miss their deadlines even with variable execution times, arrival times, and deadlines. To keep the design simple, each DAL instance manages a single microservice and assumes that the computing infrastructure can be viewed as a processor pool with a set of reserved processors and a set of on-demand processors. In the context of controlling real-time physical systems, if we consider the physical system as a client that makes a request for the service offered by the microservice task, a response must be sent to this client within a certain time duration. DAL uses an on-arrival dispatcher that assigns incoming requests to the processor that most likely meets their timing requirements. The dispatcher estimates the time a request will take on a processor before dispatching it. Based on recent work related to redundant designs [24], and overload management in data centers [25], for requests that are deemed to miss their timing requirements, DAL includes a feedback mechanism that notifies the requester of a possible violation of its timing requirement so that it can take remedial action locally. Concretely, we show via simulations that using on-demand processors and a low-complexity admission policy can provide significantly improved performance compared to using only reserved processors for jobs with deadline constraints and variable execution and inter-arrival times.

II. Motivation

To further motivate DAL's design choices, we consider path planning [26] and trajectory generation [27] as examples of a microservice and highlight the challenges imposed by such services.

a) Variable Arrival Times: In a multi-user scenario, requests may come from different clients. Even for periodic requests coming from the same clients, each client may have

different periods. For example, if the service computes a trajectory for a robot, a robot moving at a higher velocity can send requests at a higher frequency. To address this issue, we designed DAL to manage variable arrivals with an incoming request dispatcher that can decide whether the incoming request can be processed within its deadline given the pending requests and the constraints on available resources.

b) Variable Execution Times: Similar to varying arrival times, the execution time for each request can also vary, even for a service that provides the same functionality for all requests. For instance, a path planning service needs to compute paths where execution times vary depending on the required accuracy, segment lengths, and number of collision checks. For example, Alcon et al. [17] analyzed the variability in execution times of prediction and planning modules of an autonomous driving stack, and found that the variations ranged from a minimum of 25 milliseconds to a maximum of 350 milliseconds, and between 175 and 250 milliseconds respectively. To manage such variability, we design DAL to take advantage of the on-demand availability of resources.

c) Variable Deadlines: For a service like trajectory planning, each request can have its own deadline. For example, a request may have a shorter deadline if the current velocity of the client robot is higher for the same distance when compared to another request where the client robot is moving slowly. Gog et al. [20] highlighted this in the context of an autonomous driving system while Shih et al. [19] considered such state-dependent deadlines. DAL addresses deadline variability by considering EDF as its scheduling policy as it sorts jobs according to their deadlines.

d) Latency Violation Feedback: Many real-time systems can tolerate missing deadlines [28], [29], [30] and services like the path planner are no different. If a request misses its deadline, a local planner running on the robot can take over and take corrective action, such as running a local instance of the planner [24], or it can reduce its speed and send a new request with a relaxed deadline. For this reason, DAL is designed to notify a requester if its request cannot be satisfied as estimated by the dispatcher, and subsequently, if it misses the deadline while waiting in the queue. Additionally, DAL deletes the requests that have missed their deadlines to service pending requests and possibly new requests from the same client robot, similar to the analysis in [15].

III. Related Work

Kargahi [31] provided an analytical method to show the performance of parallel EDF queues for join-shortest-queue(JSQ) dispatching and a threshold-based dispatching strategy but without any arrival time acceptance or rejection. Wang et al. [23] considered the problem of dispatching and scheduling requests with heterogeneous reserved and on-demand processors where individual requests have maximum waiting time described by an exponential distribution. They provided a mathematical model as well as the multi-queue request scheduling framework that assigns incoming jobs to different queues depending on the type of the processors, followed by the allocation of jobs in queues to specific processors. They also provide a mechanism to decide the number of on-demand processors to be provisioned. Here the on-demand processors are utilized when the queue is full or when the waiting time exceeds the maximum waiting time.Similarly, Meng et al.[32] considered the problem of dispatching and scheduling jobs with arbitrary deadlines and bounded worst-case execution time on a set of reserved processors along with network transmission delays. However, these algorithms may be unsuitable for applications with low latency requirements due to their complexity.

Gao et al. [33] proposed strategies for scheduling firm semi-periodic real-time jobs in single-processor environments. The jobs are released periodically and share the same relative deadline, but their execution times can have arbitrary probability distributions. The researchers explored several optimization criteria, including the Deadline Miss Ratio (DMR). To determine whether a job should be interrupted before its deadline, they introduced three new control parameters at runtime. These parameters include an upper bound on completion times, which is used to drop a job if it cannot be completed by a certain time between periodic inter-arrival time and relative deadline; an upper bound on job execution times, which is used to reject jobs with execution times exceeding a certain value; and an upper bound on waiting time, which is used to drop a job that has waited until a certain bound. They also considered four admission policies, which involve admitting all jobs, admitting jobs until a fixed number of jobs are in the queue, admitting jobs with a fixed probability, and admitting jobs following a repeating pattern. The evaluation of their work revealed that the most critical control parameter for achieving the best DMR is the upper bound on the waiting time of each job. In contrast to this work, our research utilizes admission policies that estimate the response times based on the job execution time distribution and the number of pending jobs on a particular server. Admitted jobs are allowed to remain in the queue until they are completed or until their deadline.

IV. System Model

a) Task Model: We consider a microservice as a task, and each task releases a job of that task when a request arrives. A task is specified by its execution time distribution[1] E, a poisson arrival process with rate λ, and a deadline distribution D. Each job i of the task takes an unknown amount of time E_i from the distribution E and is expected to be completed before a relative deadline d_i drawn from a uniform distribution. The relative deadline d_i is revealed when the request arrives at the dispatcher (Fig.1). We assume that the time to decode the deadline information is zero[2]. Each request is assigned an absolute deadline D_i^a by the request decoder at the time it arrives according to eq.(1).

$$D_i^a = t_c + d_i, \tag{1}$$

[1]Same as service time distribution in queuing theory.

[2]In practice, a request arriving at the NIC is processed in FIFO order and may take a non-negligible amount of time before its relative deadline is known.

where t_c is the time at which the request is decoded.

b) Processor Pool: We assume that an arbitrary but fixed number of processors are reserved to execute requests of the microservice task. Each job of the task can be executed on any reserved processor. Each of the reserved processors has its own queue with pending jobs ordered according to the EDF scheduling policy. In addition to reserved processors, we also assume that a microservice is deployed on a set of on-demand processors. An on-demand processor can be released if it has no pending jobs of the considered task in its queue. Furthermore, we assume that there is no setup cost associated with on-demand processors, i.e., when the dispatcher sends a job to an idle on-demand processor, it immediately starts executing the dispatched job.

c) Dispatch-on-Arrival and Scheduling Policy: Once a request is processed, it is immediately dispatched to the queue of a reserved processor. The processor selection strategy is described in Section V. The jobs in a processor queue are ordered by their absolute deadlines which are calculated using Eq. (1). If a new job has a shorter deadline than the currently executing job, the scheduler preempts that job, adds it to its own queue, and starts executing the new job.

V. DAL

Scheduling jobs with stochastic parameters with static resource reservations may not provide useful performance unless the reservations are made based on worst-case behavior. For instance, when we evaluated join the shortest queue dispatch policy for exponential arrival and service times with only reserved processors and no on-demand processors, 20 percent of requests missed their deadlines even when another 20 percent of requests were discarded by the dispatcher (see Table V and Table I). We designed DAL to achieve the goal of successfully completing jobs before their deadlines for microservices with stochastic arrival times, execution times, and deadlines by considering the availability of on-demand resources consistent with the computing model of fog and cloud architectures. DAL's architecture is shown in Fig. 1. In the following sections, we describe the various components and policies that DAL employs to achieve this goal.

A. Processor Pool

DAL's design relies on the concept of a processor pool, which is based on the idea of on-demand availability of processors as supported by cloud and fog computing paradigms but can also work at the edge layer where the number of available processors may be limited. DAL assumes that it has access to fixed number of homogeneous processors at any given time in its processor pool including reserved and on-demand processors. Among these processors, reserved processors are available for DAL's exclusive use, while on-demand processors may or may not be always available. An on-demand processor is considered available if it is idle when a request is received by DAL's dispatcher, or if it has pending jobs belonging to DAL's jobs and is unavailable when it is executing jobs of a different microservice of lower priority.

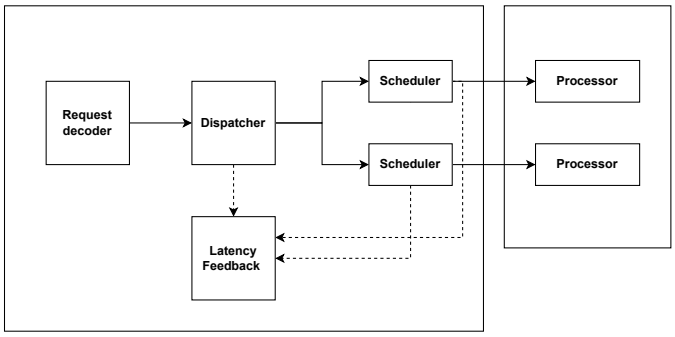


Fig. 1: System architecture of DAL.

a) Processor Allocation: When a request arrives at DAL's dispatcher, DAL first attempts to send the request to one of the reserved processors. If the dispatcher decides that the request cannot meet its deadline on one of the reserved processors, it searches for an available on-demand processor within the group of on-demand processors. If it finds an available processor, the job is immediately sent to that processor.

b) Processor Deallocation: Once a request executing on an on-demand processor completes, DAL is expected to release the processor for use by other services. However, instead of releasing the on-demand processor after the request completes, DAL holds the on-demand processor for a configurable duration by speculating on the arrival of another request within the configured duration. If no job is assigned to the processor within this period, it is released back to the processor pool.

B. Dispatcher

Several dispatch-on-arrival load balancing strategies aim to minimize expected response times for different types of job distributions and assume a fixed number of processors. Common dispatching strategies include join random queue, join shortest queue, and the power-of-d strategy, where d processors are randomly selected and the job is distributed to the processor with the fewest jobs [34], [35]. Since it is known that the performance of the power-of-d policy [34] improves the average response times under different scheduling policies such as first-in-first-out and shortest remaining processing time, DAL combines this policy with a configurable schedulability test to dispatch the jobs. DAL's dispatcher first looks for the reserved processor with the least number of pending jobs and checks whether the incoming job is schedulable. If the job is deemed to be schedulable, it is immediately sent to that processor. If the job fails the schedulability test, the dispatcher searches for an available processor from the set of on-demand processors, and if it finds one, it assigns the request to that processor. If no such processor is found, it signals the latency feedback component to send a response to the request sender about a potential deadline miss and adds the request to the queue of the processor with the least number of pending jobs among the reserved processors.

Online Schedulability Test: A schedulability test decides whether a job can be successfully scheduled on a processor given a scheduling policy and information about the pending jobs on the processor. As jobs in our system are bound by a deadline, we must determine whether the job can meet its deadline on the selected processor. Utilization-based schedulability tests that rely on worst-case execution time values can be used if the service cannot tolerate any deadline miss. Such a test requires us to keep track of all deadlines of pending requests in the queue. However, if over-provisioning is a problem and deadline misses are tolerated, a low-overhead but less accurate test may be useful. DAL dispatches jobs to a processor if the jobs satisfy such a schedulability test. DAL's schedulability test estimates the response time of an incoming job. If the estimated response time is shorter than its deadline, the incoming job is assumed to pass the schedulability test. Estimating the response time requires information about job execution times and the number of pending jobs in the queue. when using EDF policy, the accuracy of the response time estimation depends on how many pending jobs have a lower absolute deadline on the processor and the probability that future jobs will be assigned to that particular processor and that those future jobs will have a lower absolute deadline than the current job. While information about the deadlines of pending requests can be obtained, the information about the number of future jobs that jump ahead of the incoming job is difficult to predict. Moreover, even if such knowledge is available, as the exact execution time is assumed to be unknown, but only its distribution is known, the estimation depends on which value is chosen as representative of this distribution. To be useful under different distributions, DAL's configurable schedulability test introduces a parameter α that decides the proportion of pending jobs that it considers to have deadlines lower than that of the incoming job. Baldwin et al. show in [36] how such a value can be determined. The influence of α becomes relevant as the size of the pending jobs in the queue increases. To account for the dependence on the distribution of execution time, another parameter β decides which execution time value is used to estimate the response time. This can be derived by applying the central limit theorem [35]. The estimated response time R of the incoming job is given by

$$R = \beta * k * E_m, \tag{2}$$

where k is the number of pending requests with a shorter deadline than the incoming job, and E_m is the expected value of the execution time distribution. This is similar to Theorem 4 in [37]. k can be determined either by tracking the deadlines of the pending jobs or from

$$k = \alpha * N, \tag{3}$$

where N is the number of pending jobs in the queue. If R is less than its relative deadline d_i, the dispatcher assigns an absolute deadline value to the job according to Eq. (1) and adds it to the processor's queue.

C. Scheduler

DAL instantiates the preemptive EDF scheduling policy on all processors in its processor pool as DAL is designed to manage requests with deadlines. Whenever a reserved processor receives a request from the dispatcher, it starts executing the job if it has no pending requests. If it is currently executing a job, it checks whether the new job has a lower absolute deadline than the job being executed. If it does, the executing job is preempted and the new job is scheduled. otherwise, it sorts the pending requests including the newly arrived job according to their absolute deadlines. Additionally, DAL's per-processor scheduler keeps track of the waiting times of jobs queued in its queue. Whenever a new job arrives or a job is completed, it checks if any of the pending jobs have waiting times that exceed their deadlines. If such jobs exist, it notifies DAL's latency feedback component and deletes the jobs from its queue.

D. Latency Feedback

As DAL is designed for request-response communication, the requester expects a response from the server. Instead of holding requests that could not be dispatched until a processor is available, DAL notifies the requestor if the request could not be dispatched. It also notifies the requestor when jobs assigned to processors do not complete their execution within the deadline. In the first case, the advantage of early latency violation notification is that it allows the requester to take remedial actions such as locally computing the result on possibly slower computers, while still managing to get the result before the deadline, which would not be possible if it received the notification after the deadline. In addition, such early notification may also allow the requester to modify its requested deadline and send a new request. For example, if the requestor is a system such as a mobile robot, it can reduce its velocity and send an updated request with a new relaxed deadline. The significance of such a feedback mechanism becomes even more apparent when DAL cannot access any of the on-demand processors.

VI. Evaluation

We conducted simulations in various scenarios to evaluate the performance of DAL. The assessment criteria were missed deadlines, dropped jobs, slowdown ratio, and the number of processors utilized during the simulation period. A job is deemed to have missed its deadline if the sum of its waiting time and executed time exceeds the deadline, including jobs deleted before receiving any processing time. We consider a job to be dropped if the dispatcher anticipates that it cannot meet its deadline on any processor in its processor pool upon arrival. We define slowdown as the ratio of a job's actual execution time to its response time, considering only jobs that meet their deadlines. The percentage of missed deadlines and dropped jobs was evaluated as the ratio of the number of jobs that missed their deadlines or were dropped over the total number of jobs released during the simulation time. For the evaluation, we assume that all on-demand processors are

TABLE I: Performance of different on-arrival dispatch policies without on-demand processors.

Dispatch Policy	Missed Deadlines %	Dropped Jobs%
JSQ, load =90 percent	19.6209	19.6047
FF, load =90 percent	63.2401	0.0133305
JSQ, load = 50 percent	3.52308	3.50832
FF , load = 50 percent	49.13	0

always available. This assumption enables us to compare the best possible result achievable against the scenario where only reserved processors are utilized.

A. Simulation Methodology

To generate job execution times, we utilized the exponential distribution class template of the C++ library. The generated float values were rounded off to the nearest largest integer using the CEIL function of the C++ standard library. The release times of the jobs were generated following a Poisson process. For most experiments, we adjusted the arrival rate to 90 percent of the service rate, i.e., the inverse of the expected value of the execution time distribution, for different numbers of reserved processors. We set the mean of the execution time distribution to 40 and set α to 1, resulting in all pending requests being treated as those with shorter deadlines. We also let β equal 1, setting the estimated execution time of each job equal to the mean of its execution time distribution. We assigned deadlines to each job by multiplying its actual execution time with a value from a uniform distribution in the range $[2, 10]$. We note that this approach may not be practical since the actual execution time is typically unknown. However, it avoids jobs with deadlines shorter than their execution times. While we varied the number of reserved processors between one and eight, we present the results only for the case where the number of reserved processors was set to four, unless otherwise specified, as the results had similar trends for different numbers of processors. The simulation proceeded step-wise with a tick value of 1 for a duration of 1 million ticks and the execution and arrival times were set as multiples of the tick value. We assumed that the overhead of DAL's implementation was zero, although this assumption was ideal, allowing us to evaluate the approaches under consideration without considering implementation-specific details.

B. Performance with Reserved Processors

We evaluated the performance of DAL's dispatching and scheduling policy by only considering the reserved processors with exponentially distributed execution times and four reserved processors. Using the dispatcher's shortest queue approach, we found that approximately 20 percent of the total released jobs missed their deadline after being admitted, while an additional 20 percent of the jobs were dropped on-arrival by the admission control policy when the system was 90 percent loaded.

As an alternative dispatching solution, we considered a First-Fit (FF) dispatching approach that uses an arbitrary but static processor ordering and assigns jobs to the first processor on which an incoming job is deemed schedulable. With the FF policy, we observed that only 0.01 percent of the jobs were deemed unschedulable by the admission control policy, but about 63 percent of the jobs missed their deadlines, as shown in Table I. Although the JSQ dispatch policy is better than the FF approach, just over 60 percent of the released jobs managed to complete before their deadlines.

For lower loads, i.e., 50 percent load, about 92 percent of the requests managed to meet their deadlines with JSQ policy, while the FF policy achieved a success rate of less than 50 percent. These observations indicate that using only reserved processors may not be sufficient for deploying microservices with real-time requirements at higher load values.

TABLE II: Performance of DAL under low load conditions.

Dispatch Policy	Missed Deadlines%	Dropped Jobs%
JSQ with Release	0.181869	0
JSQ without Release	0.00184169	0

TABLE III: Performance of DAL with and without release of on-demand processors under high load conditions.

Dispatch Policy	Missed Deadlines%	Dropped Jobs%
JSQ without Release	0.0028588	0
JSQ with Release Period = 1	0.00954372	0
JSQ with Release Period = 10	0.00475217	0
JSQ with Release Period = 100	0.00190391	0

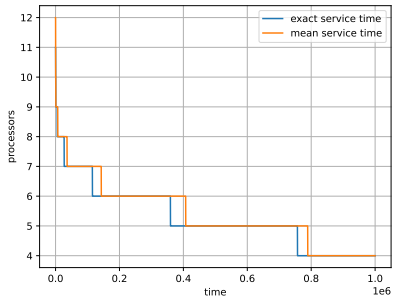
C. Performance with On-Demand Processors

We evaluated the performance of DAL's dispatching and scheduling policy with on-demand processors for exponentially distributed execution times with four reserved processors.

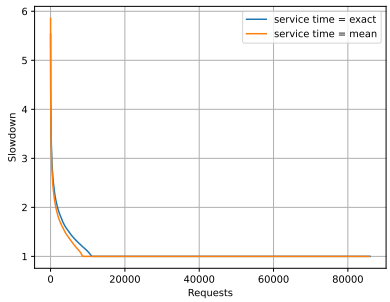
a) Low Load Scenario: With the system load set to 50 percent, we observe that up to 99.92 percent of the jobs are able to meet their deadlines when idle processors are released if they do not have any pending jobs. This is further improved to 99.99 percent when on-demand processors are not released. Table II shows the percentage of deadlines missed when on-demand processors are released if they do not have any pending requests and when on-demand processors are not released back to the processor pool. However, this improvement comes at a cost of increased resource usage as more than four on-demand processors are retained for 90 percent of the simulation duration, as seen in Fig. 4. If idle processors are released, on-demand processors are used for only 20 percent of the simulation time.

Based on these observations, we can conclude that releasing on-demand processors as soon as they become idle can provide reasonable performance in terms of slowdown as well as in successfully completing up to 99.92 percent of the jobs before deadlines, with lower resource usage.

b) High Load Scenario: Under high loads, up to 10 on-demand processors were retained for 80 percent of the time in addition to the 4 reserved processors without release. When processors are released, only 3 additional processors are used for about 80 percent of the time. The slowdown ratio of the successful requests is 1 for more than 50 percent of the requests even when processors are released and can be seen

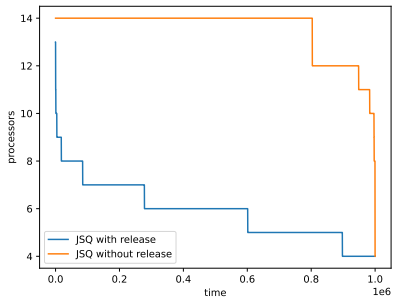


(a) Processor Usage

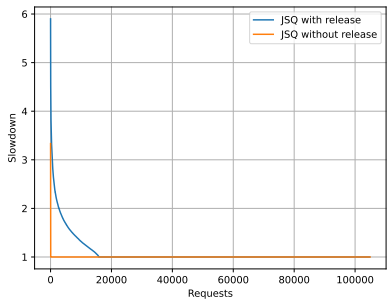


(b) Slowdown

Fig. 2: Comparison of processor usage and slowdown ratio when using approximated execution time values versus exact execution time values in a high load scenario.

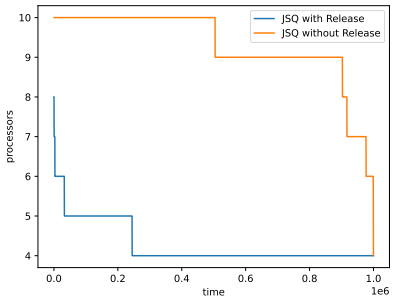


(a) Processor Usage

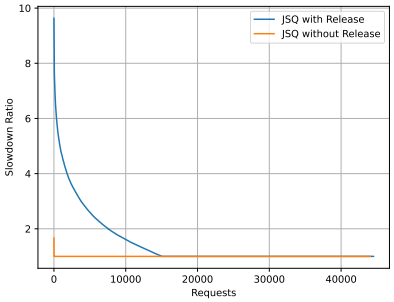


(b) Slowdown

Fig. 3: Comparison of processor usage and slowdown ratio when on-demand processors are not released and when released as soon as they are idle in a high load scenario.

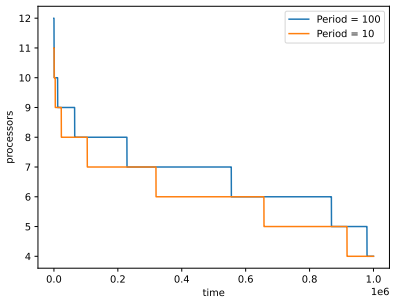


(a) Processor Usage

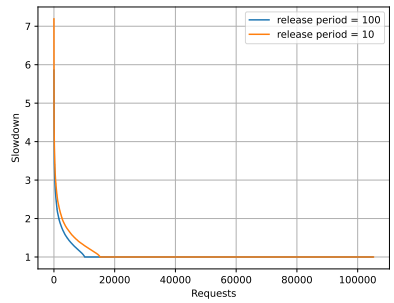


(b) Slowdown

Fig. 4: Comparison of processor usage and slowdown ratio when on-demand processors are not released and when released as soon as they are idle in a low load scenario.



(a) Processor Usage



(b) Slowdown

Fig. 5: Comparison of processor usage and slowdown ratio when on-demand processors are released with various periods in a high load scenario.

TABLE IV: Performance of DAL with and without reserved processor provisioning.

Available Processors	Missed Deadlines%	Dropped Jobs%
No reserved processors	0.0243591	0
Four reserved processors	0.00954372	0

TABLE V: Performance of DAL when using using exact and mean execution time values to estimate response times.

Estimation Type	Missed Deadlines%	Dropped Jobs%
Distribution Mean with no on-demand Processors	19.6209	19.6047
Exact Value with no on-demand Processors	16.3863	16.3805
Distribution Mean with on-demand Processors	0.0058254	0
Exact Value with on-demand Processors	0.00232582	0

in Fig. 3. Additionally, the percentage of deadlines missed is highly reduced (see Table III).

c) No Reserved Processors: We also considered the scenario where an application is deployed only on-demand processors and no reserved processors are provisioned when the system load is set to 90 percent. The percentage of missed deadlines remains low while relatively higher compared to when 4 processors are reserved (Table IV).

d) Impact of Delayed Release: We evaluated DAL's performance by periodically releasing idle on-demand processors. We observed that releasing on-demand processors as soon as they become idle provides similar performance to not releasing them at all. Even with a delay in releasing them, the improvement is insignificant. A success rate of almost 99.99 percent is achieved in all cases, as shown in Table III. We note that there is no statistically significant difference between JSQ with and without release, where the release period is set to 100. In terms of resource usage, Fig. 5 shows the different number of processors held by DAL during the simulation time. We observe that up to 13 processors are utilized for a relatively shorter duration of time even when processors are released as soon as they become idle. However, the average number of processors is significantly lower than the no release scenario while achieving almost identical performance in terms of successful completions. With respect to slowdown, we observe that the performance when processors are released periodically remains quite similar, with a slower release period having a slightly better value.

e) Impact of Execution Time Approximation: We evaluated the performance difference between using approximated execution times and exact execution times for scenarios where only reserved processors are utilized and for scenarios where on-demand processors are released as soon as they become idle. We found that in the former scenario, using approximated execution times results in approximately 6 percent higher deadline misses compared to using exact execution times, as seen in the first two rows of Table V. However, this difference is almost insignificant when on-demand processors are utilized, with almost 99.99 percent of the jobs successfully meeting their deadlines.

Regarding processor usage, we observed that the use of approximation results in a slight increase in the average number of processors being used compared to the usage of exact execution times, as shown in Fig. 2. We also noticed an almost indiscernible difference in the slowdown ratio, with more than 85 percent of the jobs having a slowdown ratio of one for both cases. Furthermore, we found that using exact execution times of pending jobs to estimate response times only marginally improves performance.

VII. Conclusion

We considered the problem of dispatching and scheduling jobs that have variable execution times, arrival times as well as deadlines in an edge computing architecture. By assuming the availability of on-demand processors, we showed that DAL's dispatch-on-arrival policy along with per processor EDF scheduling policy, can achieve significantly better performance in terms of jobs that complete by their deadlines when jobs that miss deadlines are deleted from the queue. In terms of slowdown ratio, a significant percentage of the requests have a minimum achievable slowdown of 1. When on-demand processors are released periodically, both low and relatively high periods achieve similar performance in terms of missed deadlines although the slower release periods hold onto on-demand processors for a longer duration providing very little benefit.

References

[1] S. M. Salman, V. Struhar, A. V. Papadopoulos, M. Behnam, and T. Nolte, "Fogification of industrial robotic systems: Research challenges," in *Proceedings of the Workshop on Fog Computing and the IoT*, 2019, pp. 41–45.

[2] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg, "A survey of research on cloud robotics and automation," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 2, pp. 398–409, 2015.

[3] M.-L. Lam and K.-Y. Lam, "Path planning as a service ppaas: Cloud-based robotic path planning," in *2014 IEEE International Conference on Robotics and Biomimetics (ROBIO 2014)*, 2014, pp. 1839–1844.

[4] K. E. Chen, Y. Liang, N. Jha, J. Ichnowski, M. Danielczuk, J. Gonzalez, J. Kubiatowicz, and K. Goldberg, "Fogros: An adaptive framework for automating fog robotics deployment," in *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, 2021, pp. 2035–2042.

[5] Z. Du, L. He, Y. Chen, Y. Xiao, P. Gao, and T. Wang, "Robot cloud: Bridging the power of robotics and cloud computing," *Future Generation Computer Systems*, vol. 74, pp. 337–348, 2017.

[6] V. Struhár, M. Behnam, M. Ashjaei, and A. V. Papadopoulos, "Real-time containers: A survey," in *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[7] S. Fiori, L. Abeni, and T. Cucinotta, "Rt-kubernetes: Containerized real-time cloud computing," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 36–39. [Online]. Available: https://doi.org/10.1145/3477314.3507216

[8] A. Sriraman and T. F. Wenisch, "{μTune}:{Auto-Tuned} threading for {OLDI} microservices," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 177–194.

[9] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 281–297.

[10] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 361–378.

[11] J. P. Lehoczky, "Real-time queueing theory," in *17th IEEE Real-Time Systems Symposium*. IEEE, 1996, pp. 186–195.

[12] ——, "Using real-time queueing theory to control lateness in real-time systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 25, no. 1, pp. 158–168, 1997.

[13] B. Doytchinov, J. Lehoczky, and S. Shreve, "Real-time queues in heavy traffic with earliest-deadline-first queue discipline," *Annals of Applied Probability*, pp. 332–378, 2001.

[14] A. Movaghar, "On queueing with customer impatience until the beginning of service," *Queueing Systems*, vol. 29, no. 2, pp. 337–350, 1998.

[15] M. Kargahi and A. Movaghar, "A method for performance analysis of earliest-deadline-first scheduling policy," *Journal of Supercomputing*, vol. 37, no. 2, pp. 197–222, 2006.

[16] M. Ahmadi, M. Golkarifard, A. Movaghar, and H. Yousefi, "Processor sharing queues with impatient customers and state-dependent rates," *IEEE/ACM Transactions on Networking*, vol. 29, no. 6, pp. 2467–2477, 2021.

[17] M. Alcon, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, and F. J. Cazorla, "Timing of autonomous driving software: Problem analysis and prospects for future solutions," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 267–280.

[18] G. C. Buttazzo, G. Lipari, and L. Abeni, "Elastic task model for adaptive rate control," in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*. IEEE Comput. Soc, 2-4 Dec. 1998, pp. 286–295.

[19] C.-S. Shih and J. Liu, "State-dependent deadline scheduling," in *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, 2002, pp. 3–14.

[20] I. Gog, S. Kalra, P. Schafhalter, J. E. Gonzalez, and I. Stoica, "D3: A dynamic deadline-driven approach for building autonomous vehicles," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 453–471. [Online]. Available: https://doi.org/10.1145/3492321.3519576

[21] R. I. Davis and L. Cucu-Grosjean, "A survey of probabilistic schedulability analysis techniques for real-time systems," *Leibniz Transactions on Embedded Systems*, vol. 6, no. 1, p. 04:1–04:53, May 2019. [Online]. Available: https://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v006-i001-a004

[22] A. Qadi, S. Goddard, J. Huang, and S. Farritor, "Modelling computational requirements of mobile robotic systems using zones and processing windows," *Real-Time Systems*, vol. 42, pp. 1–33, 2009.

[23] S. Wang, X. Li, Q. Z. Sheng, R. Ruiz, J. Zhang, and A. Beheshti, "Multi-queue request scheduling for profit maximization in iaas clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 11, pp. 2838–2851, 2021.

[24] J. Nouruzi-Pur, J. Lambrecht, T. D. Nguyen, A. Vick, and J. Krüger, "Redundancy concepts for real-time cloud- and edge-based control of autonomous mobile robots," in *2022 IEEE 18th International Conference on Factory Communication Systems (WFCS)*, 2022, pp. 1–8.

[25] I. Cho, A. Saeed, J. Fried, S. J. Park, M. Alizadeh, and A. Belay, "Overload control for {μs-scale}{RPCs} with breakwater," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 299–314.

[26] S. M. LaValle, *Planning algorithms*. Cambridge university press, 2006.

[27] L. Biagiotti and C. Melchiorri, *Trajectory planning for automatic machines and robots*. Springer Science & Business Media, 2008.

[28] 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.

[29] N. Vreman, A. Cervin, and M. Maggio, "Stability and performance analysis of control systems subject to bursts of deadline misses," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[30] N. Vreman, C. Mandrioli, and A. Cervin, "Deadline-miss-adaptive controller implementation for real-time control systems," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 13–26.

[31] M. Kargahi and A. Movaghar, "Dynamic routing of real-time jobs among parallel edf queues: A performance study," *Computers & Electrical Engineering*, vol. 36, no. 5, pp. 835–849, 2010.

[32] J. Meng, H. Tan, X.-Y. Li, Z. Han, and B. Li, "Online deadline-aware task dispatching and scheduling in edge computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1270–1286, 2019.

[33] Y. Gao, G. Pallez, Y. Robert, and F. Vivien, "Dynamic scheduling strategies for firm semi-periodic real-time tasks," *IEEE Transactions on Computers*, vol. 72, pp. 55–68, 1 2023.

[34] M. Mitzenmacher and M. Dell'Amico, "The supermarket model with known and predicted service times," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2740–2751, 2022.

[35] M. Harchol-Balter, *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[36] R. O. Baldwin, N. J. Davis Iv, J. E. Kobza, and S. F. Midkiff, "Real-time queueing theory: A tutorial presentation with an admission control application," *Queueing Systems*, vol. 35, no. 1, pp. 1–21, 2000.

[37] A. Biondi and Y. Sun, "On the ineffectiveness of 1/m-based interference bounds in the analysis of global edf and fifo scheduling," *Real-Time Systems*, vol. 54, no. 3, pp. 515–536, 2018.