2025 38th International Conference on VLSI Design and 2025 24th International Conference on Embedded Systems (VLSID)

2025 38th International Conference on VLSI Design and 2025 24th International Conference on Embedded Systems (VLSID) | 979-8-3315-2244-5/25/$31.00 ©2025 IEEE | DOI: 10.1109/VLSID64188.2025.00014

MERGERS: Multi-access Edge Resource GovErnance for Real-Time SaaS Systems

Aakashjit Bhattacharya
ATDC, IIT Kharagpur, India
aakashjit.bhattacharya021@kgpian.iitkgp.ac.in

Arnab Sarkar
ATDC, IIT Kharagpur, India
arnab@atdc.iitkgp.ac.in

Ansuman Banerjee
ACMU, ISI Kolkata, India
ansuman@isical.ac.in

***Abstract*—The Multi-access Edge computation (MEC) paradigm is being regarded as a viable alternative for handling data and computation requirements of complex real-time and safety-critical applications in upcoming IoT systems. This paper presents a resource arbitration technique called *MERGERS* that utilizes a central adaptive orchestrator called Resource Arbiter (*RA*), to dynamically allocate real-time service requests among heterogeneous MEC servers of a 5G MEC Software-as-a-Service (*SaaS*) provider. The objective of MERGERS is to maximize overall resource utilization of the given set of MEC servers through efficient load balancing. The approach partitions time into uniform intervals known as *epochs*. At the start of each *epoch*, *RA* takes into account the computational, storage, and deadline requirements of new client requests received in the previous epoch. It then uses a strict admission control policy to assign the new requests to suitable MEC edge servers, while ensuring that the hard real-time requirements of already scheduled tasks are not compromised. A newly arrived request is rejected if it's inclusion in the system can lead to deadline misses. Experimental results show *MERGERS* is able to achieve high task acceptance rates over a variety of simulation scenarios.**

***Index Terms*—Multi-access Edge Computing (MEC), Real-Time load balancer for MEC environment, Hard Real-Time Scheduler, Software-as-a-Service provider, Resource Arbitration**

I. Introduction

Multi Access Edge Computing (MEC) is essential for integrating application-centric features into carrier networks. MEC deployment extends beyond 4G+ or 5G+ and is expected to deliver applications as edge services across a range of domains. When used together with the Ultra Reliable Low Latency Communications (URLLC) capability in 5G+, Multi-access Edge Computing (MEC) ushers in the promise of being able to support applications that have strict/ hard latency and QoS requirements [1], [2]. Futuristic MEC systems are expected to cater to ubiquitously connected smart real-time (RT) edge services, especially those that monitor and control Internet of Things (IoT) devices in various domains like e-healthcare, manufacturing, intelligent transportation, smart city and home applications, Internet of Vehicles, etc. [3]–[7]. It is important to emphasize that in the context of service-oriented computing in the cloud/edge in general, and specifically in the case of MECs, efficient resource allocation and careful load balancing can lead to significant enhancements in resource usage efficiencies. In addition, optimizing the use of resources might potentially result in substantial decrease in design expenses [8].

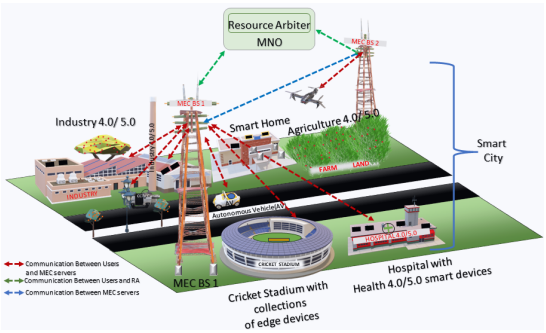

Fig. 1: System Block Diagram of *MERGERS*

Offloading workload on MEC servers necessitate load balancing to address inevitable imbalances and meet specific QoS level requirements [9]–[14]. In [15], the authors proposed a latency-sensitive service placement model to enhance Quality of Experience (QoE). Initially, the model attempts to distribute the load among edge servers associated with a set of coordinating base stations. In case, none of the nearby edge servers are available (possibly due to insufficient capacity or service absence), a user service request is forwarded to the cloud. Although, this strategy can handle services having soft-real time constraints, it lacks mechanisms such as strong admission control and deadline awareness, which are needed to handle hard real-time applications, as discussed above. In [16], an MEC service provider operates a network comprising of multiple heterogeneous MEC base stations having stipulated, possibly overlapping coverage areas. An user task request can be served by an MEC base station if the user's location is within the coverage area of that base station. As the coverage areas of multiple base stations can partially overlap, a user may be potentially served by more than one base station. In this scenario, the objective of this work is to handle a set of user requests using a minimum number of MEC servers. If a request cannot be hosted within the remaining capacities of any eligible edge server, the request is offloaded to the cloud. The authors in [17] have proposed *CooLoad*, a cooperative resource allocation scheme among a set of geographically close edge data centers which are connected by high speed trunks. At any instant of time, each data centre is marked by

2380-6923/25/$31.00 ©2025 IEEE
DOI 10.1109/VLSID64188.2025.00014

a pre-existing workload schedule; a newly arriving request is accommodated within the spare resources available within the schedule. When an edge data centre becomes fully loaded, it transits to *blocking state* which disallows the acceptance of any further requests. A new request arriving at a data centre in *blocking state* is blindly forwarded to any nearby edge data centre which either accepts it if the request can be accommodated, or blindly forwards it to another edge data centre (if it is also in *blocking state*). Ultimately, the request either gets allocated on any one of the cooperating edge data centres, or is dropped. The work in [18] presents a load balancing strategy among a set of fog nodes. The fog nodes are maintained as a set of disjoint clusters such that a request arriving at a fog node can only be executed by a member of the same cluster. Inter-cluster request migration is prohibited. The members of a cluster periodically share their workload status. When a request cannot be accommodated locally, such status information can be used to take better decisions on suitable remote nodes within the cluster which can execute the concerned request. In this regard this policy proves to be more efficient than blind request forwarding as proposed in [17]. Resource usage efficiency may however be pulled down by the blanket prohibition on inter-cluster request migrations. The distributed strategy may also be prone to high intra-cluster network traffic due to status information exchange especially when the cluster sizes are large. Wang, et al. [19] have discussed a scenario which features a collection of base stations (BSs) having specific coverage areas. Each BS serves requests emanating within its coverage area using a single designated edge server (ES) which is associated with the BS. An ES is accessible to a BS if it is within its coverage area, with access times being proportional to the distance between ES and BS. A single ES however can serve multiple BSs. Each BS is associated with a defined workload profile which is given by the distribution of data transmission demands originating from user requests arriving at BS over time. Given a fixed ES capacity and the workload profile associated with each BS, the objective of the work is to determine the optimal number of edge servers and their locations, such that the average data access times experienced by any BS remains within a soft real-time upper bound threshold. A similar work has been presented in [20].

In this work we introduce *MERGERS*, a heuristic real-time resource arbitration mechanism which enables a centralized resource arbiter RA to seamlessly load balance a set of MEC servers, in the face of dynamically arriving real-time service requests. As new service requests arrive for execution to RA in a given epoch, *MERGERS* accumulates and tries to equitably distribute them across the available resources. For each newly arrived service request, *MERGERS* either accepts and schedules the service on a specific server, or rejects it, taking into account multiple factors including: (i) instantaneous workloads on servers, (ii) resource requirements of the service, (iii) deadline, and finally, (iv) the revenue that may be obtained if the request is accepted and executed. The objective of *MERGERS* is to simultaneously maximize resource usage efficiency of the system as well as the aggregate revenue acquired by RA.

The paper is organized as follows. Section II introduces our proposed model, Section III provides a detailed explanation of the algorithms and load balancing method, Section IV describes the experimentation process and outcomes, and Section V concludes the study.

II. Proposed Model

In the proposed model as depicted in Fig. 1, we consider a set $B = \{B_1, B_2, ..., B_n\}$ of n base stations having non-overlapping cellular coverage regions, interconnected via a fully connected backhaul network [21]. Each base station B_i has a co-located edge server $E_i : \langle C_i, S_i^E \rangle$, where C_i represents the computation capacity (unit: #instructions per unit time) and S_i^E, the storage capacity of E_i(unit: #bytes). Real-time service requests (which we also refer to as tasks) which are dynamically generated within a given cell are received by it's local base station and then forwarded to a centralized resource arbiter RA. A client can generate a service request from a stipulated set of l available services $SR = \{sr_1, sr_2, ..., sr_l\}$. Each service $sr_i \in SR$ is characterized by a 3-tuple $\langle I_i, D_i, CR_i \rangle$, where I_i is the number of instructions associated with the service's code, D_i is the relative deadline of the service, and CR_i is the rent that the client has to pay to avail the service. Whenever a client

TABLE I: Table of Notations

Symbol	Meaning
E_i	i^{th} MEC server under *RA*
CL_σ	Client σ
ST_{E+1}	Starting time of $(E+1)^{th}$ epoch boundary
T_σ	Task of CL_σ
D_σ	Relative deadline of sr_σ
TD_σ	Remaining time before deadline at the next epoch boundary for T_σ
T_{sch}	List of already scheduled, partially completed tasks
T_{sch}^c	List of unscheduled tasks
TLST	List of unscheduled tasks arranged in non-increasing order of priority
C_i	Computation speed of E_i
S_i	Storage capacity of E_i
CR_k	Rent for k^{th} service
S_σ	Storage space required by T_σ
I_σ	No:of instructions in T_σ
$MAXTIME$	Maximum till which all the MEC servers are scheduled across all the MEC server.
HRT	Hard Real-Time
EDF	Earliest Deadline First

CL_σ, requests a service say sr_p, a corresponding service request instance T_σ (also referred to as task henceforth) is generated, where I_σ (number of instructions of T_σ) = I_p, D_σ (Relative deadline of the task) = D_p and TD_σ (remaining time before deadline at the epoch boundary) = $D_\sigma - ST_{E+1}$ - latency (here we are considering uniform latency across all task requests). RA maintains a discrete timeline where time is measured as an integral number of time slots or time quanta. Further, RA partitions this timeline into equal length intervals called $epochs$. Requests which arrive dynamically over the t^{th} epoch are scheduled by RA at the beginning of the $(t+1)^{th}$ epoch. At any epoch boundary, there exists a set

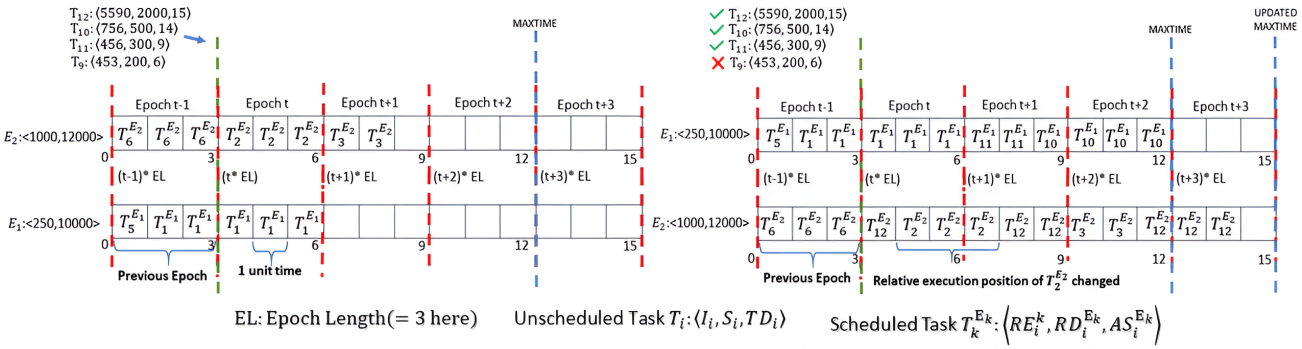

Fig.2a: At the beginning of *Epoch t*: Before allocation of newly arrived tasks

Fig.2b: At the beginning of *Epoch t*: After allocation of newly arrived tasks

Fig. 2: Schedule generation at the beginning of *Epoch t*

of newly arrived unscheduled tasks $T^c_{sch} = \{T_1, T_2, ..., T_\xi\}$, and a set of already scheduled partially completed tasks T_{sch}, where $T_{sch} = \{T^{E_1}_{sch} \cup T^{E_2}_{sch} \cup T^{E_3}_{sch} \cup ... \cup T^{E_N}_{sch}\}$. Each element $T^{E_k}_{sch}$ in T_{sch} contains the list of M_{E_k} partially completed tasks ($T^{E_k}_{sch} = \{T^{E_k}_1, T^{E_k}_2, T^{E_k}_3 ..., T^{E_k}_{M_{E_k}}\}$) in edge server E_k. The remaining execution requirement, the time remaining before deadline, and the allocated server for the i^{th} task in MEC server s (T^s_i) in T^s_{sch}, is denoted by a 3-tuple $T^s_k : \langle RE^s_k, ReD^s_k, AS^s_k\rangle$. Additionally there is a set $T^c_{sch} = \{T_1, T_2, ..., T_\xi\}$ of ξ newly arrived tasks which must be scheduled starting from the ensuing epoch utilizing the spare capacities of the available edge servers. The execution demand (#instructions), storage demand (#bytes), and the remaining time before deadline for the i^{th} task in T^c_{sch} is also represented by a 3-tuple $T_i : \langle I_i, S_i, TD_i\rangle$, where TD_i is the remaining time before deadline at the epoch boundary.

Fig. 2 depicts an example scenario in which at the beginning of the t^{th} epoch, there exists a set of partially completed tasks, $T_{sch} = \{T^{E_1}_1 : \langle 1000, 4, E_1\rangle, T^{E_2}_2 : \langle 4950, 7, E_2\rangle, T^{E_2}_3 : \langle 2000, 9, E_2\rangle\}$, and a set of newly arrived tasks, $T^c_{sch} = \{T_{10} : \langle 756, 12, E_1\rangle, T_{11} : \langle 456, 7, E_1\rangle, T_{12} : \langle 4590, 12, E_2\rangle\}$. The MEC system consists of two base stations (having edge servers $E_1 : \langle 250, 10000\rangle$, and $E_2 : \langle 1000, 12000\rangle$). Fig. 2a shows the execution Gnatt charts on E_1 and E_2 before the new tasks in T^c_{sch} have been scheduled and allocated. On the other hand, Fig. 2b shows the Gnatt charts after the new tasks have been allocated and added to T_{sch}. It may be noted that the allocated slots for the existing tasks (in this case $T^{E_2}_2$) have been modified in Fig. 2b. However, Fig. 2b still represents a deadline meeting schedule for all the tasks comprising both the existing and the new ones.

III. Resource Allocation Methodology

A. MERGERS: Algorithm Overview

The *MERGERS* algorithm is periodically invoked at each epoch boundary in order to allocate new service requests which have arrived over the duration of the preceding epoch. The steps followed by *MERGERS* for allocating the new requests have been shown in Fig. 3. As shown, the newly

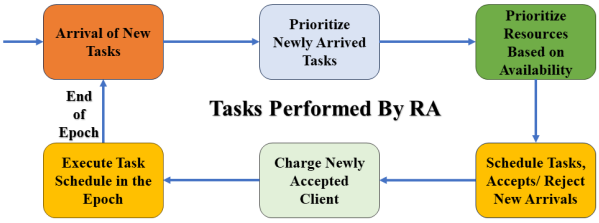

Fig. 3: Tasks Performed By *RA* as per *MERGERS*

arrived requests are first arranged in a task list ordered based on their priority values. The priority value (P^{RA}_k) of a request (T_σ) depends on its service type (sr_k) which determines the resource demand (number of instructions I_k, and relative deadline D_k) for the request as well as the rent (CR_k) that may be obtained through its execution. The relative resource demand η_k for service demand sr_k is calculated with respect to a hypothetical system $E_{ref} : \langle C_{ref}, S_{ref}\rangle$. The actual resource capacity C_p, S^E_p of any resource E_p is proportional to capacity of E_{ref} that is, $C_p = C_{ref} * \nabla^p_{cpu}$, and $S^E_p = S_{ref} * \nabla^p_{sto}$, where ∇^p_{sto} and ∇^p_{cpu} are constants of proportionality. η_k is obtained as a weighted summation over sr_k's relative storage demand and computation demand as shown in Eq 1.

$$\eta_k = \lambda(S^E_\sigma/S_{ref}) + (1-\lambda)(I_\sigma/C_{ref})/TD_\sigma \quad (1)$$

Here, λ is a normalization constant; $0 \leq \lambda \leq 1$. Given CR_k and η_k, the priority ρ^{RA}_k of sr_k is obtained as shown in Eq 2.

$$\rho^{RA}_k = CR_k/\eta_k \quad (2)$$

After task prioritization, the edge servers are also arranged in a priority queue based on their processing speeds as well as the discrete spare processing capacities available on them. After prioritization of both the service requests and the edge servers, *MERGERS* conducts request-to-server mapping in a sequential manner through an efficient but low-overhead scheduling strategy. A request is accepted if *MERGERS* can feasibly schedule it within the deadline; otherwise, the request is rejected. At each iteration during partial schedule generation, if a task is accepted and allocated on a given server, the priority of that server is recomputed and the sorted priority list of servers

is updated accordingly. Then a new iteration commences to allocate the next unscheduled task. This process continues until all unscheduled tasks in T^c_{sch} have been considered for allocation. After allocation, the final schedule for the ensuing epoch (comprising of both the *previously allocated still-to-be-completed tasks* as well as the *newly accepted requests*) are generated and dispatched. *MERGERS* uses a strict admission control policy to assign the new requests to available MEC edge servers, while ensuring that the hard real-time requirements of already scheduled tasks are not compromised. If at the beginning of any epoch, the remaining time available before deadline for a new request is not sufficient to generate a valid schedule on any of the available edge servers, then the request is rejected.

B. MERGERS: Detailed Algorithm

Algorithm 1 *MERGERS*

Require: T^c_{sch}, T_{sch}, mecServers[], SR, $MAXTIME$
Ensure: Fair load balancing across resources
1: **while** $epoch! = \phi$ **do**
2: /*Prioritizing Tasks*/
3: Generate a list $TLST$ of tasks in T^c_{sch}
4: Sort $TLST$ in non-increasing order of the tasks' priority values
5: $MAXTIME \leftarrow$ highest deadline among tasks in $T_{sch} \cup T^c_{sch}$
6: /*Prioritizing MEC Servers*/
7: **for all** E_i in $mecServers$ **do**
8: $P_{E_i} \leftarrow$ total duration of idle time slots in E_i before $MAXTIME$
9: $\varphi_{E_i} \leftarrow$ remaining storage of E_i
10: $CAP_{E_i} \leftarrow \alpha * P_{E_i} * C_{E_i} + (1-\alpha) * (\varphi_{E_i}/S^{E_i})$
11: **end for**
12: Sort $mecServers$ in non-increasing order of cap_{E_i} values
13: **for all** T_σ in $TLST$ **do**
14: **for all** E_i in $mecServers$ **do**
15: $flag \leftarrow Schedule(T_\sigma, E_i, currentSystemTime)$
16: **if** $flag$ == true **then**
17: $P_{E_i} \leftarrow P_{E_i} - (I_\sigma/C_{E_i})$
18: $\varphi_{E_i} \leftarrow$ remaining storage of E_i
19: $CAP_{E_i} \leftarrow \alpha * P_{E_i} * C_{E_i} + (1-\alpha) * (\varphi_{E_i}/S^{E_i})$
20: insert E_i at it's appropriate location in $mecServers$
21: break
22: **end if**
23: **end for**
24: **end for**
25: **end while**

Algorithm 1 is employed by *RA* in order to fairly distribute newly arrived tasks in T^c_{sch} at a given epoch boundary, across all the available resources (refer *while* loop, steps 1 - 23). At first, the tasks in T^c_{sch} are arranged in a task list *TLST* in non-increasing order of their priority values (steps 3 - 4). In order to conduct task allocation, the total number of available idle time slots (P_{ES}) in each MEC server (ES) up to a given instant of time in the future (which we refer to as *MAXTIME*) is determined (step 8). Here, *MAXTIME* points to the maximum absolute deadline among all tasks in $T^c_{sch} \cup T_{sch}$ (step 5). Given P_{ES}, C_{ES} (server speed), and φ_{ES} (available server space), server priority (CAP_{ES}) is obtained in step 10. Step 12 generates a priority order of the servers by sorting $mecServers$ in non-increasing order of server capacities (CAP_{ES}). The nested for loops in steps 13-24 attempts to sequentially allocate and schedule the newly arrived tasks to appropriate servers, in the order of task and server priorities. If a valid deadline meeting schedule for T_σ can be generated on a server *ES*, the $flag$ in step 15 is set to $true$, available idle slots (P_{ES}), server capacity (C_{ES}), and available server space (φ_{ES}) of *ES* are updated (steps 17, 18, and 19), and *mecServers* is re-ordered based on updated CAP_{ES} (step 19). On the other hand if *flag* is $false$, T_σ is iteratively attempted for allocation on the next server in $mecServers$ until all servers have been tested. If none of the servers are successful in scheduling T_σ, it is rejected.

Algorithm 2 Schedule

Require: T_ξ, ES, CT
Ensure: Return true or false based on the task allocation status
1: **if** $S_\xi <$remaining storage of *ES* **then**
2: return false
3: **end if**
4: Sort partially completed tasks (T^{ES}_{sch}) along with the new task T_ξ in non-increasing order of remaining time before deadline: $T_{tmp} \leftarrow T^{ES}_{sch} \cup \{T_\xi\}$; Sort T_{tmp}
5: /*Determine Latest Start Times of Tasks*/
6: $ST[T^1_{tmp}] \leftarrow ReD^{ES}_1 - RE^{ES}_1$
7: **for all** i from 2 to $M^{ES}+1$ **do**
8: **if** $ReD^{ES}_i < ST[T^{i-1}_{tmp}]$ **then**
9: $ST[T^{i-1}_{tmp}] \leftarrow ReD^{tmp}_i - RE^{tmp}_i$
10: **else**
11: $ST[T^i_{tmp}] \leftarrow ST[T^{i-1}_{tmp}] - RE^{tmp}_i$
12: **end if**
13: **if** $ST[T^i_{tmp}] < CT$ **then**
14: return false
15: **end if**
16: **end for**
17: /*Left shift Task*/
18: $ST[T^{M^{ES}+1}_{tmp}] \leftarrow CT$
19: $FT[T^{M^{ES}+1}_{tmp}] \leftarrow ST[T^{M^{ES}+1}_{tmp}] + Re^{ES}_{M^{ES}+1}$
20: **for** $k = M^{ES}$ to 1 **do**
21: $ST[T^k_{tmp}] \leftarrow FT[T^{k-1}_{tmp}]$ + 1
22: $FT[T^k_{tmp}] \leftarrow ST[T^k_{tmp}] + Re^{ES}_k$
23: **end for**
24: Generate updated schedule for ES with $[ST[T^j_{tmp}], FT[T^j_{tmp}]]$ as the execution slot for task T^j_{tmp}
25: return true

Function *Schedule()*: Algorithm 2 attempts to schedule a new task T_ξ in MEC server ES, without compromising the schedulability of any partially completed task in T^{ES}_{sch}. So it at first checks if the required amount of storage space is available in *ES* for T_ξ to execute. If not, *Schedule()* returns *false*. A set of tasks $T_{tmp} = \{T^1_{tmp}, T^2_{tmp}, ..., T^{M^{ES}+1}_{tmp}\}$, sorted in non-increasing order of *remaining time before deadline* (ReD) is created (step 4). T_{tmp} includes all tasks in T^{ES}_{sch} along with T_ξ. After that, the latest start times of all tasks in T_{tmp} are iteratively determined starting with the task having the farthest relative deadline (loop steps 7 - 16). At any iteration of the loop (say, the i^{th} iteration), schedule generation fails if the start time $ST[T^i_{tmp}]$ of task T^i_{tmp} is less than the current time CT (steps 13 - 15). Otherwise, the loop in steps 7 to 16 executes successfully and a feasible schedule can be generated. Now the actual start and finish times of all tasks are calculated so that the work-conserving schedule can be generated (steps 18 - 23).

C. MERGERS: Complexity

The *MERGERS* algorithm is called at the beginning of every epoch in order to schedule a set of K newly arrived unscheduled tasks on M servers. To check for schedulability or generate the schedule of any given task on a specific resource, *MERGERS* calls function *Schedule()*. So, before

discussing the complexity of *MERGERS*, we first analyze the complexity of the *Schedule()* function by assuming that there are N partially completed tasks already executing on the MEC servers. Without loss of generality, we assume that these existing tasks are equally distributed on the M servers. Hence both the for loops in steps 7 - 16 and steps 20 - 23 execute N/M times, also all statements within these loops as well as other statements in the function are constant time operations. Thus complexity of the *Schedule()* function becomes $O(N/M)$.

In the *MERGERS()* function, the sort operation in step 4 consumes $O(KlogK)$ time. Determination of $MAXTIME$ in step 5 has $O(K)$ complexity. Determination of total idle slot in step 8 in a given server consumes $O(N/M)$ time, step 9 has $O(1)$ overhead, therefore the complexity of the *for* loop in steps 7 - 10 is $O(M)*O(N/M) = O(N)$. The sort operation in step 11 has $O(MlogM)$ complexity. The inner *for* loop in steps 13 - 21 executes M times for each newly arrived task. The complexity of each iteration of this loop is dominated by the *Schedule()* function which has $O(N/M)$ overhead. Thus complexity associated with the for loop in steps 7 - 22 is $O(K*M)*O(N/M) = O(K*N)$. Assuming $K, M << N$, the complexity of the $MERGERS$ algorithm becomes $O(K*N) = O(N)$.

IV. Experimental Evaluation

This section evaluates the proposed *MERGERS* scheduler through extensive experiments. We detail the experimental setup, evaluation metrics, and obtain results. All experiments have been conducted on a 9^{th} Gen Intel(R) Core(TM) i7 processor and 32GB RAM.

A. Experimental Setup

A series of simulation based experiments have been conducted in a controlled environment using a Java based simulator called *FLASH*, developed by us. The total simulation duration and epoch length in all experiments have been taken as 30,000 time units and 10 time units, respectively. Datasets for our experiments are generated as follows:-

Serverset Generation: Experiments have been conducted for 25 or 75 heterogeneous servers. The CPU and storage capacities of the servers are generated from Normal Distributions using the corresponding fixed capacities $\langle C_{ref}, S_{ref} \rangle$ of a reference server as mean. In this work we have assumed $C_{ref} = 90,000$ *instructions per unit time*, and $S_{ref} = 5,00,000$ *units of storage*. While the standard deviation of the Normal Distribution for obtaining CPU capacities are taken as $5\% \times C_{ref}$ = 4500, that for obtaining storage capacities are taken as $5\% \times S_{ref}$ = 25,000.

Taskset Generation: The performance of the proposed algorithm has been conducted under controlled system environment settings. Thus, throughout the simulation duration of any particular experiment, aperiodic tasks have been generated randomly while ensuring that the approximate total active workload in the system at any instant, remains bounded within a stipulated upper and lower cap. In order to evaluate the efficiency of *MERGERS* in terms of it's ability to allocate a server to a newly arrived task under diverse conditions, we have conducted experiments for various CPU workload bands, viz. $50\% \pm 5\%$, $60\% \pm 5\%$, $70\% \pm 5\%$, $80\% \pm 5\%$, and $90\% \pm 5\%$. For all these experiments, it is also ensured that the total storage demand of the active tasks at any instant never causes storage related overload. The data associated with each task T_i namely, number of instructions I_i, storage demand S_i, and relative deadline TD_i, are generated from Uniform Distributions having ranges [52×10^4, 42×10^5], [25×10^3, 30×10^3], and [10, 60] respectively. The approximate CPU utilization (U_i) of T_i, assuming execution on the reference server, is obtained as $U_i = I_i/(C_{ref} * TD_i)$. Now, the following mechanism is used to ensure that a task generation process always respects the constraints associated with a given system workload band $WL \pm x$. Let, the current total workload at an instance t be $WL(t)$, and a decision needs to be made on whether a new task T_i having utilization U_i can be spawned. In this scenario, T_i is generated only if the following condition is satisfied: $WL(t) + U_i \leq WL + x$. Similarly let, the execution of a task T_i terminates at time t, and the resulting workload $WL(t) - U_i$ becomes $< WL - x$. In this case one or more new tasks are forcefully spawned at t, so that $WL(t)$ becomes $\geq WL - x$.

B. Experimental Results

The performance of *MERGERS* has been evaluated by measuring:- i) the achieved *acceptance rates* and ii) the *corresponding solution generation times*, against variation in average system workloads.

i. Acceptance Rate Measurement:- Acceptance rate (%) has been obtained as the percentage ratio of the number of tasks accepted, to the number of tasks that arrive for execution in a given simulation run. Each data point in the plots is an average result over 100 simulation runs. The results are presented in Fig. 4. Firstly, it may be observed that even at 50% average system workload, acceptance rates (although very high; above $\sim 94\%$), do not reach 100%, this is because of two main reasons: a) the set of servers are heterogeneous, b) each task is restricted to fully execute on a single server, that is, inter server migration of tasks are not allowed. As expected, for any given number of servers (say, 25 servers), acceptance rates are seen to progressively decrease (from 94% to 82%) with increase in average system workload from 50% to 90%. Further results for 75 servers may be observed to be consistently better than the results for 25 servers for any given average system workload. This may be directly attributed to the higher number of resource allocation options when the number of servers are significantly more.

ii. Solution generation times:- For any simulation run, the average scheduling overhead associated with a given task has been measured by dividing the total runtime of *MERGERS*, by the total number of tasks that arrived for execution during the simulation. Fig. 5 shows the results. Again, each data point in Fig. 5 is obtained as an average over 100 simulation runs. For any given number of servers (say, 25 servers), it

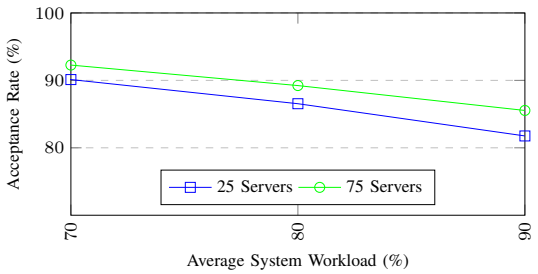

Fig. 4: Acceptance Rates Vs. System Workloads

may be seen that the per task scheduling time increases from $\sim 155ns$ to $\sim 245ns$, as the average system workload is increased from 50% to 90%. This increase in overhead can be attributed to: a) higher time required to search a free slot within any given server (at higher workloads), and b) higher time required to find a server that can accommodate a new task, when searched in server-priority order. Additionally, per task scheduling overheads are significantly higher for systems with 75 servers compared to 25 server systems.

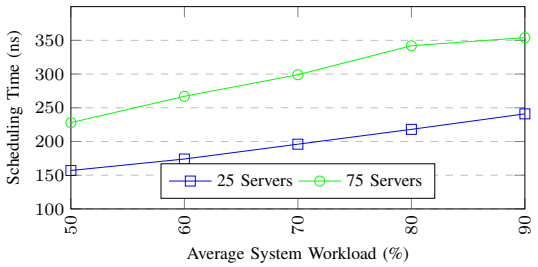

Fig. 5: Scheduling Time Comparison for Constant Storage

V. Conclusion

This paper proposes an efficient load balancing strategy for heterogeneous MEC server systems, considering services/ tasks having hard real-time constraints. The presented scheme *MERGERS* employs efficient but low overhead task and resource prioritization policies to fairly distribute dynamically arriving task requests across all the available resources. *MERGERS* is able to achieve handsome resource usage efficiency leading to high task acceptance rates. The performance of *MERGERS* has been evaluated using extensive simulation based experiments. Going forward, we wish to experiment on real workloads to see how our methods work.

References

[1] Pham, Quoc-Viet, Fang Fang, Vu Nguyen Ha, Md Jalil Piran, Mai Le, Long Bao Le, Won-Joo Hwang, and Zhiguo Ding. "A survey of multi-access edge computing in 5G and beyond: Fundamentals, technology integration, and state-of-the-art." IEEE access 8 (2020): 116974-117017.

[2] Liu, Yaqiong, Mugen Peng, Guochu Shou, Yudong Chen, and Siyu Chen. "Toward edge intelligence: Multiaccess edge computing for 5G and Internet of Things." IEEE Internet of Things Journal 7, no. 8 (2020): 6722-6747.

[3] Siriwardhana, Yushan, Pawani Porambage, Madhusanka Liyanage, and Mika Ylianttila. "A survey on mobile augmented reality with 5G mobile edge computing: Architectures, applications, and technical aspects." IEEE Communications Surveys Tutorials 23, no. 2 (2021): 1160-1192.

[4] Shi, Weisong, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. "Edge computing: Vision and challenges." IEEE internet of things journal 3, no. 5 (2016): 637-646.

[5] Yousefpour, Ashkan, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. "All one needs to know about fog computing and related edge computing paradigms: A complete survey." Journal of Systems Architecture 98 (2019): 289-330.

[6] Mao, Yuyi, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B. Letaief. "A survey on mobile edge computing: The communication perspective." IEEE communications surveys tutorials 19, no. 4 (2017): 2322-2358.

[7] Ning, Zhaolong, Jun Huang, Xiaojie Wang, Joel JPC Rodrigues, and Lei Guo. "Mobile edge computing-enabled internet of vehicles: Toward energy-efficient scheduling." IEEE Network 33, no. 5 (2019): 198-205.

[8] Wang, Pengfei, Chao Yao, Zijie Zheng, Guangyu Sun, and Lingyang Song. "Joint task assignment, transmission, and computing resource allocation in multilayer mobile edge computing systems." IEEE Internet of Things Journal 6, no. 2 (2018): 2872-2884.

[9] Mach, Pavel, and Zdenek Becvar. "Mobile edge computing: A survey on architecture and computation offloading." IEEE communications surveys tutorials 19, no. 3 (2017): 1628-1656.

[10] Porambage, Pawani, Jude Okwuibe, Madhusanka Liyanage, Mika Ylianttila, and Tarik Taleb. "Survey on multi-access edge computing for internet of things realization." IEEE Communications Surveys Tutorials 20, no. 4 (2018): 2961-2991.

[11] Zhao, Junhui, Qiuping Li, Yi Gong, and Ke Zhang. "Computation offloading and resource allocation for cloud assisted mobile edge computing in vehicular networks." IEEE Transactions on Vehicular Technology 68, no. 8 (2019): 7944-7956.

[12] El Haber, Elie, Tri Minh Nguyen, and Chadi Assi. "Joint optimization of computational cost and devices energy for task offloading in multi-tier edge-clouds." IEEE Transactions on Communications 67, no. 5 (2019): 3407-3421.

[13] Zhang, Ke, Yuming Mao, Supeng Leng, Quanxin Zhao, Longjiang Li, Xin Peng, Li Pan, Sabita Maharjan, and Yan Zhang. "Energy-efficient offloading for mobile edge computing in 5G heterogeneous networks." IEEE access 4 (2016): 5896-5907.

[14] Liu, Chubo, Kenli Li, and Keqin Li. "A game approach to multi-servers load balancing with load-dependent server availability consideration." IEEE Transactions on Cloud Computing 9, no. 1 (2018): 1-13.

[15] Somesula, Manoj Kumar, Sai Krishna Mothku, and Sudarshan Chakravarthy Annadanam. "Cooperative Service Placement and Request Routing in Mobile Edge Networks for Latency-Sensitive Applications." IEEE Systems Journal (2023).

[16] Lai, Phu, Qiang He, Mohamed Abdelrazek, Feifei Chen, John Hosking, John Grundy, and Yun Yang. "Optimal edge user allocation in edge computing with variable sized vector bin packing." In Service-Oriented Computing: 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings 16, pp. 230-245. Springer International Publishing, 2018.

[17] Beraldi, Roberto, Abderrahmen Mtibaa, and Hussein Alnuweiri. "Cooperative load balancing scheme for edge computing resources." In 2017 Second International Conference on Fog and Mobile Edge Computing (FMEC), pp. 94-100. IEEE, 2017.

[18] Al-Khafajiy, Mohammed, Thar Baker, Hilal Al-Libawy, Zakaria Maamar, Moayad Aloqaily, and Yaser Jararweh. "Improving fog computing performance via fog-2-fog collaboration." Future Generation Computer Systems 100 (2019): 266-280.

[19] Wang, Shangguang, Yali Zhao, Jinlinag Xu, Jie Yuan, and Ching-Hsien Hsu. "Edge server placement in mobile edge computing." Journal of Parallel and Distributed Computing 127 (2019): 160-168.

[20] Li, Xingcun, Feng Zeng, Guanyun Fang, Yinan Huang, and Xunlin Tao. "Load balancing edge server placement method with QoS requirements in wireless metropolitan area networks." IET Communications 14, no. 21 (2020): 3907-3916.

[21] Polese, Michele, Marco Giordani, Tommaso Zugno, Arnab Roy, Sanjay Goyal, Douglas Castor, and Michele Zorzi. "Integrated access and backhaul in 5G mmWave networks: Potential and challenges." IEEE Communications Magazine 58, no. 3 (2020): 62-68.