RESEARCH ARTICLE OPEN ACCESS

Investigating the Schedulability of PeriodicReal-Time Tasks in Virtualized Cloud Environment

R.NAVIN KUMAR MCA., M.Phil.*, B.NEELAMPARI**
* Assistant Professor, Department of MCA, Nandha Engineering College (Autonomous), Erode, Tamilnadu, India.
Email: navinsoccer07@gmail.com
** Final MCA, Department of MCA, Nandha Engineering College (Autonomous), Erode, Tamilnadu, India.
Email: neelambari1022@gmail.com

****

ABSTRACT:

In this paper, we developed a computing architecture and algorithms for supporting soft real-time task scheduling during a cloud computing environment through the dynamic provisioning of virtual machines. The architecture integrated three modified soft real-time task scheduling algorithms, namely Earliest Deadline First, Earliest Deadline until Zero-Laxity, and Unfair Semi-Greedy. A deadline look-ahead module was incorporated into each of the algorithms to fire deadline exceptions and avoid the missing deadlines, and to maintain the system criticality. The results of the implementation of the proposed algorithms are presented during this paper in terms of the typical deadline exceptions, the additional resources consumed by each algorithm in handling deadline exceptions, and therefore the average reaction time . The results not only suggest the feasibility of the soft real-time scheduling of periodic real-time tasks in cloud computing but that the method can also be scaled up to handle the near-hard real-time task scheduling.

Index terms - **Real-time, cloud computing, virtual machine, deadline, laxity.**

****

I.INTRODUCTION

With the exponential growth of big data induced by the proliferation of Internet technologies, there has been an increase in the demand for cloud computing over the last few years. Cloud computing is a type of distributed parallel computing system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements between the service provider and consumers [1], [2]. Cloud computing facilitates flexible and dynamic outsourcing of applications while improving cost-effectiveness.

Cloud Computing offers three different types of service models: Software-as-a-Service (SaaS), Platform- as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS). In SaaS model, the consumer is offered to use provider's applications hosted in a cloud infrastructure. In PaaS, the consumer is provided with required software and hardware tools to develop and deploy cloud applications that are hosted in the provider's cloud infrastructure. In IaaS, the consumer is provisioned the use of virtualized computing, storage and network resources that are delivered on demand basis. In IaaS model, the consumer will not have control on the cloud infrastructure, however, he can control the operation system, the storage and deployed applications beside the possibility of controlling limited network components such as firewalls [1], [2]. Cloud computing is enabling the event of subsequent generation of computing services, which might be heavily geared

toward massively distributed on-line computing. It also affords a replacement model of worldwide accessible on-demand high-performance computing (HPC) services. However, such benefits are presently not available to real-time safety-critical applications [3]. this might flow from to the lack of current cloud infrastructures to support timing constraints. Compared with the case of existing real-time scheduling platforms like multiprocessors and multicores [4], the handling of real-time constraints on cloud platforms is more complex thanks to the problem of predicting system performance in virtualized and heterogeneous environments [5]. However, real-time applications are gradually progressing unto cloud computing platforms, driven by the tremendous possibilities afforded by such platforms. samples of hard and soft real-time system applications of cloud computing are military distributed control systems for remote surveillance, early warning and response systems, sensor-driven unmanned vehicles with augmented intelligence, and cloud gaming [5]–[7]. Cloud computing technology isn't actually geared toward hard real-time applications in closed environments, but soft real-time applications that don't require direct exposure to the hardware bypassing system software [6], [7]. Incidentally, soft real-time scheduling policies are often integrated invirtualzation platforms, enabling the system to deliver hierarchical real-time performance [6].

In this paper, we propose and analyze the likelihood of applying real-time task scheduling, or deadline-constrained task scheduling, on IaaS cloud computing service model, for the support of sentimental and near-hard real time applications. samples of such applications are cloud-based gaming, online video streaming, and telecommunication management [5]–[7]. Such applications may benefit significantly from cloud computing despite the restrictions related to the start-up time and communication of virtual machines. this is often due to the power of cloud computing to support dynamic workloads, thereby enabling the elastic allocation of resources [6], [7].

The rest of the paper is organized as follows. Section 2 introduces the task model and therefore the proposed cloud architecture, and also defines some terms utilized in this paper. Section 3 briefly reviews related works. Section 4 further describes the proposed cloud architecture and its underlying scheduling algorithms. Section 5 presents and discusses the results of the demonstrative implementation of the architecture. Finally, the conclusions drawn from the study are presented in Section 6.

II. MODEL AND TERMS DEFINITION

This paper considers the problem of scheduling independent periodic real-time tasks with implicit deadlines (i.e. task deadlines are equal to task periods, i.e. di pi) on a virtualized cloud environment that offers IaaS services to cloud subscribers

In real-time systems, a periodic real-time task is one that is 'released' periodically at constant intervals. Such periodic real-time tasks can be found in a broad range of real-time system applications. In fact, many of the tasks carried out by these systems are typically periodic in nature. For-example, monitoring certain conditions in a plant or in a flight control system with different set of sensors sending their data periodically at constant rates, i.e. at every specific period. Another example is changing the track in a radar system over specified period according to the target movement. Yet another example is polling information from sensors periodically and respond accordingly (e.g. driving some actuators).

A periodic real-time task Ti, i 1, 2, . . . , n, is usually described by three parameters, namely, its worst-case execution time ei, its deadline di, and its period pi.1 An instance i.e. release of a periodic task is referred to as a job and is denoted by Tij eij, pij , j 1, 2, 3, . . . ,. eij refers to the worst-case execution requirement of job Tij, and pij refers to its period. The deadline of a job is the arrival time of its successor. For example, the deadline of job Tij is the arrival time of job Ti(j+1), namely, at time (j 1)pi. The laxity of a job

Tij at time t, denoted by lij,t , is that the duration over which Tij can remain idle before its execution is commenced, as denoted by equation 1.

$$l_{ij,t} = p_{ij} - e_{ij,t} - t \quad (1)$$

where eij,t denotes the remaining time in the execution of job Tij at time t.Another important parameter that is used to describe a task Ti is its utilization, denoted by equation 2, and refers to the proportion of the time required for its execution relative to the entire duration between its release and deadline.

$$u_i = e_i / p_i \quad (2)$$

We use the term, Usum to denote the total utilization of a given taskset T , and Umax to denote the maximum utilization. A periodic real-time taskset T T1, T2, . . . , Tn is said to be schedulable on N nodes each with m identical multiprocessors if and only if Usum (T) < N m and Umax (T) 1. Table 1 summarizes the notations and terms used in this paper.

In IaaS cloud service model, Virtual Machines (VMs) are used to deliver computing, memory, and other resources to cloud subscribers. We assume that the Master Node (MN) in the cloud platform has access to a pool of virtual machine nodes. Initially, the pool contains NVM virtual machine nodes, each of which contains m symmetric shared-memory multi-processors (SMPs). The total number of available processors in the cloud is denoted by M NVM m.

Considering the target of a virtualized environment, α is used to denote the VM instantiation time, and β the delay introduced by the communication between the MN and native 1In the case of real-time tasks with implicit deadlines, i.e. di pi, the term di is typically omitted, with the term pi wont to ask both the task deadline and period.

TABLE 1. NotationS and terms.

Notation	Definition
N, N_{VM}	Number of nodes
M	The total number of processors
n	Number of tasks
T	Set of tasks $\{T_1, T_2, \ldots, T_n\}$
T_i	ith task
T_{ij}	jth job of task T_i
p_i	Period (minimum inter-arrival time) of task T_i
p_{ij}	Period (minimum inter-arrival time) of jth job of task T_i
e_i	Worst-case execution requirement of task T_i
e_{ij}	Worst-case execution requirement of jth job of task T_i
	Remaining worst-case execution requirement of jth job of task T_i at time t
	Laxity of task T_i

VM nodes.2 Hence, if an instance of a task Ti, i.e. Tij, is migrated and suffering from the VM instantiation time, its new worst-case execution time is given by equation

$$erij=eij+\alpha+\beta \quad (3)$$

Accordingly, the new laxity of job Tij at time t is given byequation4.

$$lirj(t)=pij-erij-t \quad (4)$$

III. LITERATUREREVIEW

A real-time system is defined by Burns and Welling [1] as an 'information processing system which has got to answer externally generated input stimuli within a finite and specified period: the correctness depends not only on the logical result but also on the time it was delivered; the failure to respond is as bad as the wrong response.' This means that a real-time system has a dual notion of correctness, namely, logical and temporal. Another important feature of a real-time system is that it must be predictable. Predictability implies the possibility to show, demonstrate, or prove that the real-time constraints are always met based on any assumptions such as the workloads [2]–[7].Real-time systems are often classified into two main classes, hard and soft, supported the value of the failure related to missing or violating the timing constraints [2], [7]–[9], [12]. There have been many proposals of optimal real-time algorithms for multiprocessors [8]–[12]. Such algorithms are utilized in hard real-time systems to ensure that no deadline is missed during a schedulable task set.How ever ,they are inflexible 2In this paper, the delay caused by the communication between the VM nodes and the master node is considered zero, given the relatively low communication level, which is also supported by current high-speed networks.

Another important feature of a real-time system is that it must be predictable. Predictability implies the possibility to show, demonstrate, or prove that the real-time constraints are always met based on any assumptions such as the workloads [2]–[7].

Real-time systems can be classified into two main classes, hard and soft, based on the cost of the failure associated with missing or violating the timing constraints [2], [7]–[9], [12]. There have been many proposals of optimal real-time algorithms for multiprocessors [8]–[12]. Such algorithms are used in hard real-time systems to guarantee that no deadline is missed in a schedulable taskset

IV. METHODOLOGY

The following subsections discuss in detail the proposed cloud computing architecture and its underlying algorithms for real-time job scheduling on a cloud platform.

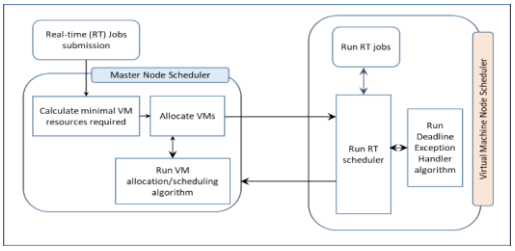

FIGURE 1. Proposed real-time cloud computing architecture.

A.PROPOSED ARCHITECTURE

The proposed cloud computing architecture, shown in Fig. 1, is composed of two main parts: the MN and the VM nodes. As mentioned previously, it was assumed that the MN had access to the VM pool. Hence, when submitted real-time jobs arrive at the MN, the master scheduler selects a VM resource from the pool and assigns tasks to it until equation 5 is violated, i.e. until the VM is fully utilized.

$$\Sigma \, ei \leq mVM \quad (5)$$

```
1.  Receive real-time tasks
2.  Calculate minimum computing resources (VMs)
3.  Allocate VMs from VM pool
4.  For every VM node in VMs
5.      Assign tasks to the VM node.
6.      Run the VM node.
7.  Wait for deadline exceptions raised by VMs.
8.  If a deadline exception is received
9.      look for a node with an idle processor
10.     If such a node (VM) exists,
11          assign the task to that node
12.     else
13.         start a new VM node
14.         assign the task to that VM node
```

FIGURE 2. Pseudocode of the MN scheduler.

Once the minimum required number of VM resources is calculated, the MN scheduler starts by assigning real- time tasks to the VM node and begin it immediately. Figure 2 shows the pseudo-code of the master node MN scheduler.

```
1.  Initialize scheduling queues
2.  Run the tasks according to proposed algorithm (USG,
    EDZL, EDF)
3.  Check for any deadline exceptions.
4.  If a deadline exception is suspected.
5.      remove the suspected task from the waiting queue
6.      raise a deadline exception event
7.      and send the suspected task to the master
```

FIGURE 3. Pseudo code of VM node scheduler

B. IMPLEMENTATION OF VM NODE SCHEDULERS

The VM node schedulers are implemented using Each of these algorithms was modified to incorporate a deadline look-a-head module to intercept any possible deadline miss. The following lemma shows exactly when a running task i.e. job is expected to miss its deadline.

Lemma 1: A job Tij scheduled for execution on a virtual node VMNi is expected to fire a deadline exception at time t tx when its laxity reaches α β (i.e. lij (t) α β)and the nearest running job to completion still has remaining execution units greater than α β (i.e eij(t)>αβ).

Proof: Suppose that job Tij is being selected for execution at time t< tx. Suppose also that, while task Ti is running it misses its deadline at time t =tz, where tz=>tx. This means that at time t= <tz,Tij reached its deadline while still having some remaining units to execute. However, we know that when a task is being executed, its remaining units decrease while its laxity remains constant. Nevertheless, job Tij misses its deadline during its execution. This could only happen if, when job Tij is being selected for execution at time t tx, its remaining time to deadline (i.e. laxity) is less than its remaining execution units, i.e. pi (t =x) < eri (t= x) > pi (t=x) eri (t=x) < 0. This means that lri (t =x) < 0. Hence, the last means of avoiding a deadline miss for job Tij is when lri (t) 0.

C.DEADLINE LOOK-A-HEAD SCHEDULERS

With the above lemma in mind, we modified the above- mentioned algorithms, namely, USG, EDZL, and EDF, by adding a deadline look-a-head module to every . This was achieved by maintaining a queue of waiting jobs so as of accelerating laxity. The deadline look-a-head scheduling module constituted a further queue in EDF because the roles in its original waiting jobs queue are so as of accelerating deadline, and not laxity.3 Conversely, both USG and EDZL already maintain a queue of waiting jobs so as of increasing laxity. All the algorithms were thus appropriately updated with a deadline look-a-head handler to handle any deadline exception raised by the algorithm.

```
1. while ((deadline_Look_A_HeadQ.size()>0) &&
   (deadline_Look_A_HeadQ.peek().key == tCur))
2.   Task tU = deadline_Look_A_HeadQ.poll();
3.   waitingQ.remove(tU);//remove the correspondent
4.   tU.key = tU.p - tU.key%tU.p+ tCur;//change to key
5.   MasterScheduler.urgentQ.add(tU);
```

FIGURE 4. Deadline exception handler.

Figure 4 shows the proposed deadline exception handler, which is named upon whenever a deadline exception is fired.

V. RESULTS AND DISCUSSION

To test the performance of the modified algorithms in the pro- posed cloud computing architecture, we used a standard prcedure to generate the real-time tasksets [8], [9]. The task periods pi were generated using the Uniform Integer Distribution, which produces random integers within the range [a, b], with all the possible values having the same likelihood of being produced. To generate the task worst-case execution times ei, Uniform Real Distribution was first used to uniformly generate a random real number x within the range (0, 1]. The worst-case execution time of the tasks was then calculated using equation 6

$$ei=bx\times pi] \qquad (6)$$

The task periods and the worst-case execution requirements were subsequently uniformly chosen with in the period[1,1000].With regard to the VM instantiation time, it was assumed that the worst-case delay caused by the VM startup time was 100 s. This was based on the performance evaluation of the VM startup time for cloud computing in . In this previous work, it was shown that the average VM startup times for Linux and Rackspace machines were 96.9 and 44.2 s, respectively. The delay was divided by the number of real- time tasks to be scheduled, n, and the result was added to the worst-case execution time of each task. To ease the simulation, the delay caused by the communication between the VM nodes and the master node was neglected, given the relatively low communication level, which is also supported by current highspeed networks.

It was further assumed that the number of real-time tasks to be executed was twice the total number of processors on the targeted cloud computing platform. Table 2 enumerates the details of the simulation test bed with regard to the generated tasksets and their corresponding cloud computing architecture. For example, 10000 samples of real-time tasks of size 16, i.e. 16 tasks per sample, were generated for execution on a cloud platform containing four nodes, each of which was equipped with two processors, and another two nodes, each equipped with four processors. This configuration enabled the tracing and investigation of the relevant factors, namely, the numbers of nodes and processors that more significantly impacted the response time and deadline exceptions reported by each algorithm.

The results were presented in terms of the average number of deadline exceptions, the average number of additional resources required to handle the fired deadline exceptions, and the average response time.

The following subsections discuss the results. The simulation has been developed using Java SE 8 development kit installed on an Oracle Solaris 10 operating sys- tem running on a Core I7 machine equipped with 8 GB of RAM.

TABLE 2. Details of the simulation test bed.

Tasksets	No. of	No. of Per Nodes
10000 tasksets, , each taskset contains 16 tasks	4 2	2 4
10000 tasksets, each taskset contains 32 tasks	8 4 2	2 4 8
10000 tasksets of size 64	16 8 4 2	2 4 8 16
10000 tasksets of size 128	32 16 8 4 2	2 4 8 16 32
10000 tasksets of size 256	64 32 16 8 4 2	2 4 8 16 32 64

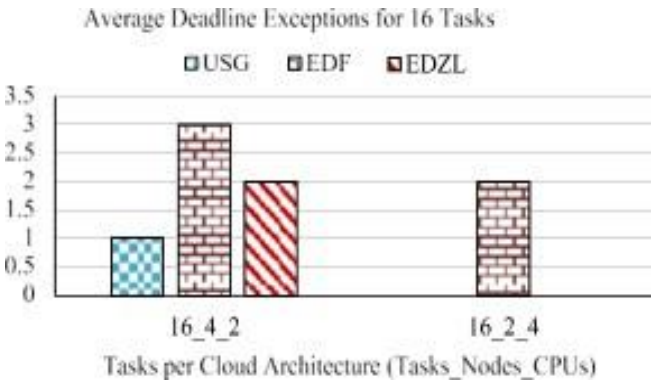

FIGURE 5. Average deadline exceptions of 16 task

A.DEADLINE EXCEPTIONS

Figures 5-9 show the typical deadline exceptions fired by each algorithm. It are often clearly seen that the USG fired the smallest amount number of deadline exceptions, followed by EDZL, then EDF. This agrees with the findings of previous studies which showed that the performance of Least Laxity First (LLF)-based algorithms surpassed that of EDF-based algorithms. For example, as indicated in Fig. 5, when 16 tasks are scheduled on four nodes with two processors each, USG fires one deadline exception, EDZL fires two deadline exceptions, and EDF fires three deadline exceptions. However, when 16 tasks are scheduled on two nodes with four processors each, both USG and EDZL fire no deadline exception, while EDF fires two

deadline exceptions. The superior performance of USG and EDZL could be attributed to the integrated Zero Laxity (ZL) policy.

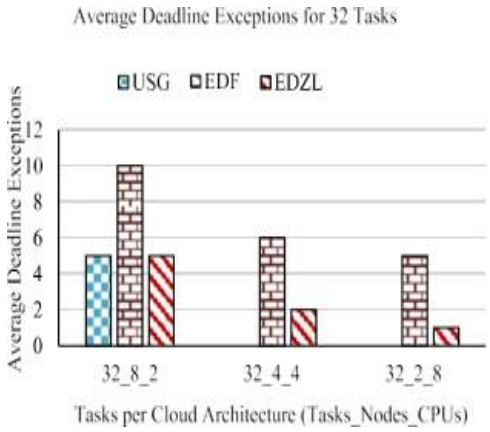

FIGURE 6. Average deadline exceptions of 32 task

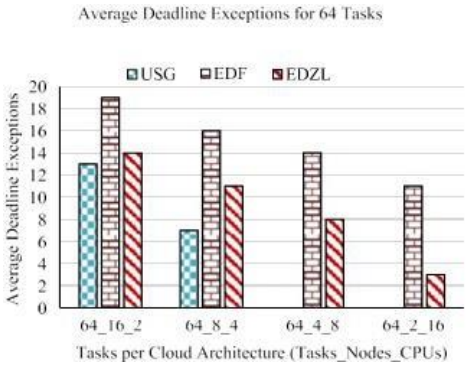

FIGURE 7. Average deadline exceptions of 64 task.

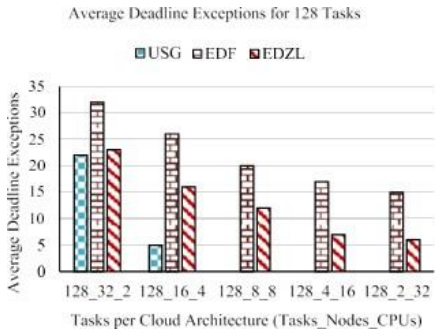

FIGURE 8. Average deadline exceptions of 128 tasks.

B.EXTRA REQUIRED RESOURCES

Based on the fired deadline exceptions, the MN scheduler allocates new VMs and assigns the urgent tasks to them within the absence of a VM node(s) with idle processors. Hence, the number of additional required resources is directly proportional to the number of fired exceptions. Figures 10 show the additional resources that the different algorithms require to handle the deadline exceptions It can be clearly.

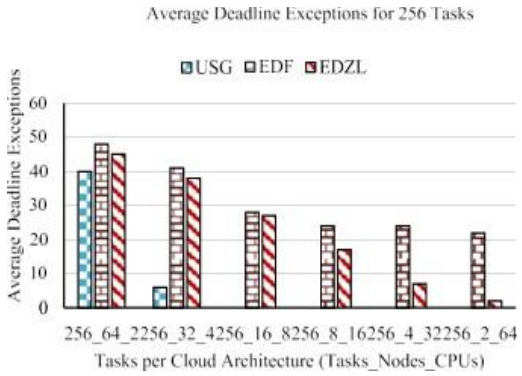

FIGURE 9. Average deadline exceptions of 256 tasks

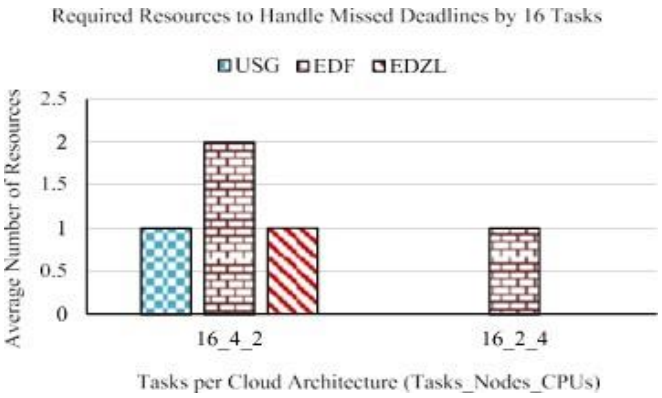

FIGURE 10. Extra resources of 16 task.

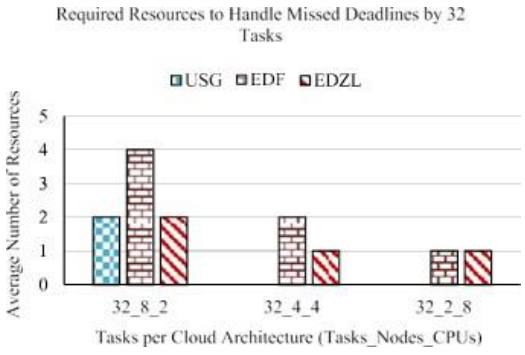

FIGURE 11. Extra resources of 32 task.

seen from the figures that USG requires the least amount of additional resources, which is because it fires the least number of deadline exceptions. It is followed by EDZL, and then EDF. For example, in Fig. 10, for 16 tasks scheduled on four nodes with two

processors each, USG is provisioned with one extra resource, and each of EDZL and EDF with two extra resources. Conversely, for the same number of tasks scheduled on two nodes with four processors each, both

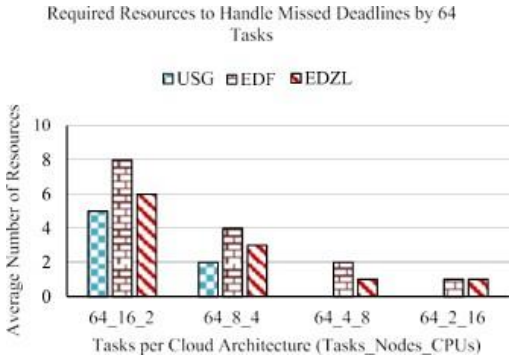

FIGURE 12. Extra resources of 64 task

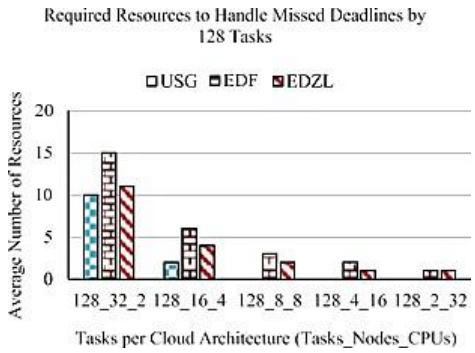

FIGURE 13. Extra resources of 128 tasks.

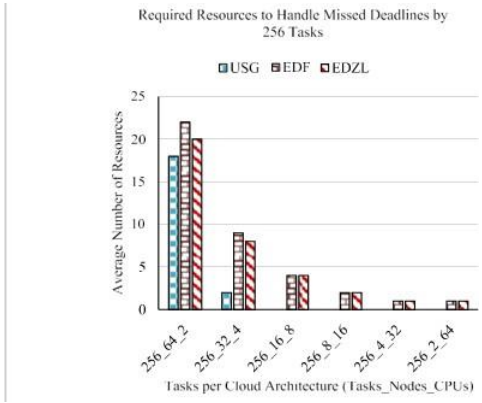

FIGURE 14. Extra resources of 256 tasks.

USG and EDZL require no additional resource, while EDF is provisioned with an extra resource.

C.RESPONSE TIME ANALYSIS

With regard to the response time, EDF produced theshortest among the three algorithms in the proposed

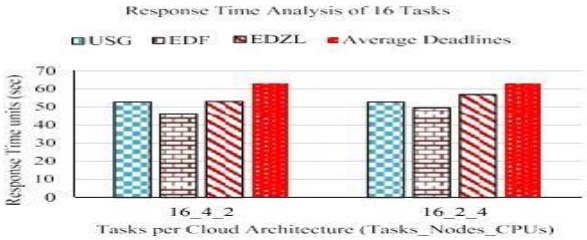

FIGURE 15. Average Response Time Analysis of 16 Tasks.

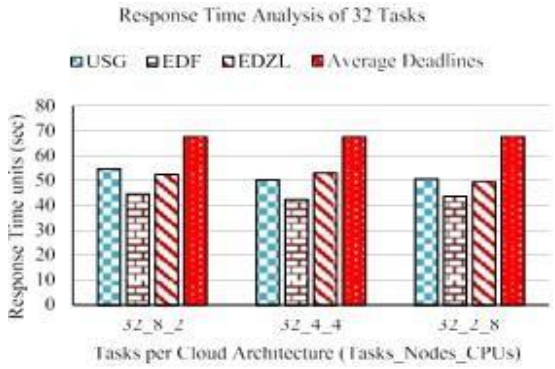

FIGURE 16. Average Response Time Analysis of 16 Tasks.

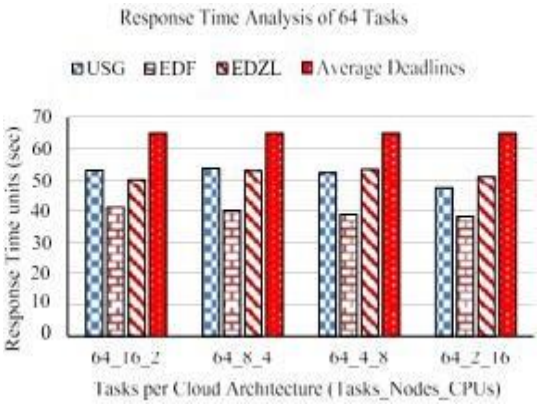

FIGURE 17. Average Response Time Analysis of 16 Tasks.

cloud architecture. This was, however, mainly because of the extra resources that the algorithm was provisioned with to handle the deadline exceptions. For example, as indicated in Figs. 15, when 16 tasks are scheduled on four nodes with two processors each, EDF produces an average response time of 46.1 s compared with the 52.8 s of USG and 53 s of EDZL. This is mainly because, for example, for the given architecture and number of tasks in Fig. 10, EDF is provisioned with two extra resources, which is twice that of each of

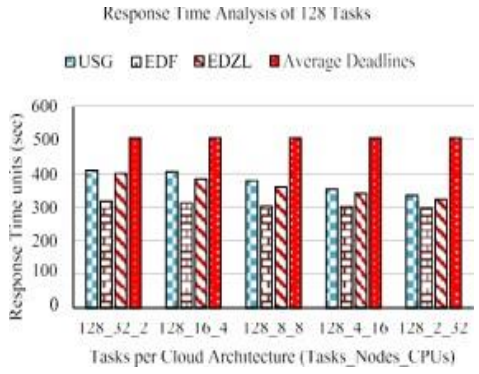

FIGURE 18. Average Response Time Analysis of 16 Tasks.

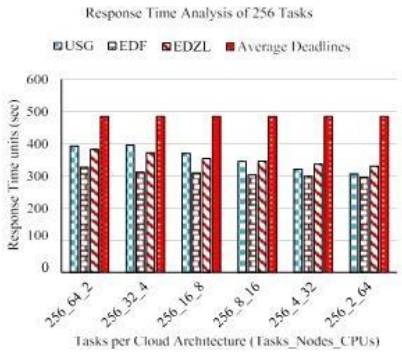

FIGURE 19. Average Response Time Analysis of 16 Tasks.

USG and EDZL. However, the important observation is that all the algorithms produced average response times within the corresponding average deadlines in all the simulations. This implies that all the deadline exceptions were successfully handled by each algorithm and all the tasks could be completed before their deadlines. Further, for the same number of provisioned extra resources, USG outperformed EDZL with respect to the response time. Nevertheless, EDZL often requested for more resources owing to its earliest deadline policy.

VI.CONCLUSION

In this study, we developed a cloud computing architecture and algorithms for soft and near-hard real-time scheduling. The idea was to utilize the features of cloud computing in the dynamic allocation and release of computing resources in response to workload needs. For this purpose, deadline look- a-head schedulers were developed for the firing of deadline exceptions before their occurrence. When such exceptions are detected, the VM node sends the tasks causing them to the MN. The MN then attempts to assign the tasks to an empty processor. In the absence of such a processor, a new VM node is dynamically created and assigned the task. The results of the demonstrative implementation of the proposed architec- ture and algorithms in the present study were expressed in terms of the number of deadline exceptions fired by each algorithm, the number of extra resources provisioned to each algorithm to handle the deadline exceptions, and the average response time of the tasks. The USG algorithm was found to fire the lowest number of deadline exceptions, and require the lowest number of extra resources, followed by the EDZL and EDF algorithms, respectively. In terms of the response time, EDF produced the shortest response time, although it also recorded the highest number of deadline exceptions. USG outperformed EDZL in terms of the reaction time for an equivalent provisioned number of resources. These results suggest the applicability of the scheduling of periodic real- time tasks to cloud computing. This assumes that the start- up time and communication limitations of virtual machines would be overcome in the near future through advancements in virtual machine technologies[12].

The non-consideration of some other important scheduling parameters such as cost and energy consumption constitutes a limitation of the present study. Further experimental study is planned to consider such parameters as secondary priorities in addition to meeting the constrained deadlines of real-time tasks.

REFERENCES

[1] A. Burns and A. J. Wellings, Real-Time Systems and Program- ming Languages, 4th ed. Toronto, ON, Canada: Pearson Education, 2009.
[2] G. C. Buttazzo, Hard Real-Time Computing Systems: Predictable Schedul- ing Algorithms and Applications, vol. 24, 3rd ed. New York, NY, USA: Springer, 2013.
[3] J. W. S. Liu, Real-Time Systems, 1st ed. Upper Saddle River, NJ, USA: Prentice-Hall, 2000.
[4] R. Mall, Real-Time Systems: Theory and Practice. London, U.K.: Pearson, 2009.
[5] H. Kopetz, Real-Time Systems, 2nd ed. New York, NY, USA: Springer, 2013.
[6] J. A. Stankovic and K. Ramamritham, ‘‘What is predictability for real-time systems?’’ Real-Time Syst., vol. 2, no. 4, pp. 247–254, 1990.
[7] I. Lee, J. Y. Leung, and S. H. Son, Handbook of Real-time and Embedded Systems. Boca Raton, FL, USA: CRC Press, 2007.
[8] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, ‘‘Multiprocessor scheduling by reduction to uniprocessor: an ingenious optimal approach,’’ Real-Time Syst., vol. 49, no. 4, pp. 436–474, 2013.
[9] G. Nelissen, V. Berten, V. Nélis, J. Goossens, and D. Milojevic, ‘‘U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic
tasks,’’ in Proc. 24th Euromicro Conf. Real-Time Syst. (ECRTS), Pisa, Italy, Jul. 2012, pp. 13–23.
[10] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, ‘‘DP -FAIR: an easy model for understanding optimal multiprocessor scheduling,’’ in Proc. 22nd Euromicro Conf. Real-Time Syst. (ECRTS), Jul. 2010, pp. 3–13.
[11] S. Funk and V. Nanadur, ‘‘LRE-TL: An optimal multiprocessor scheduling algorithm for sporadic task sets,’’ in Proc. 17th Int. Conf. Real-Time Netw. Syst., 2009, pp. 159–168.
[12] B. Andersson and E. Tovar, ‘‘Multiprocessor scheduling with few preemp-tions,’’ in Proc. 12th IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl., Aug. 2006, pp. 322–334.