Hindawi
Mathematical Problems in Engineering
Volume 2022, Article ID 2379643, 12 pages
https://doi.org/10.1155/2022/2379643

Research Article

Intelligent Fault-Tolerant Mechanism for Data Centers of Cloud Infrastructure

Satish Kumar T,[1] **Madhusudhan H S**,[2] **S. M. F. D. Syed Mustapha**,[3] **Punit Gupta**,[4] **and Rajan Prasad Tripathi**[5]

[1]*Department of Computer Science & Engineering, BMS Institute of Technology & Management, Bengaluru, Karnataka, India*
[2]*Department of Computer Science & Engineering, NIE Institute of Technology, Mysuru, Karnataka, India*
[3]*College of Technological Innovation, Zayed University, Dubai, UAE*
[4]*Department of Computer and Communication Engineering, Manipal University Jaipur, Jaipur, India*
[5]*Department of Electronics and Communication, Amity University Tashkent, Tashkent, Uzbekistan*

Correspondence should be addressed to Punit Gupta; punitg07@gmail.com

Received 9 November 2021; Accepted 6 December 2021; Published 8 February 2022

Academic Editor: Naeem Jan

Fault tolerance in cloud computing is considered as one of the most vital issues to deliver reliable services. Checkpoint/restart is one of the methods used to enhance the reliability of the cloud services. However, many existing methods do not focus on virtual machine (VM) failure that occurs due to the higher response time of a node, byzantine fault, and performance fault, and existing methods also ignore the optimization during the recovery phase. This paper proposes a checkpoint/restart mechanism to enhance reliability of cloud services. Our work is threefold: (1) we design an algorithm to identify virtual machine failure due to several faults; (2) an algorithm to optimize the checkpoint interval time is designed; (3) lastly, the asynchronous checkpoint/restart with log-based recovery mechanism is used to restart the failed tasks. The valuation results obtained using a real-time dataset shows that the proposed model reduces power consumption and improves the performance with a better fault tolerance solution compared to the nonoptimization method.

1. Introduction

Cloud computing has emerged as a prominent paradigm over the past decade and its use has seen substantial growth [1]. Not only small scale users but also large scale commercial business and scientific applications are getting benefited by the use of cloud. With minimal effort, users can get services from the cloud as it enables ubiquitous, on demand access to a shared pool of computing resources. Resources like software, hardware, and apps are shared resources. The three main layers of cloud architecture are Software as a Service, Infrastructure as a Service, and Platform as a Service. Fault may occur on all these three layers; nevertheless, software based algorithms are identified and applied to recover from faults.

Fault tolerance is described as a system's capacity to continue executing its intended purpose in the face of errors or faults [2, 3]. Even a well-designed system with the greatest components and services cannot be called dependable without fault tolerance capabilities [4]. Because a large number of delay-sensitive (real-time) applications must be run, reliability is a critical aspect of cloud computing. Furthermore, service dependability is critical to the cloud's wider acceptance. As a result, fault tolerance has gotten a lot of attention in research. There are various fault tolerance mechanisms—replication, checkpointing, Self-Healing, Task Migration, Retry, Safety-Bag Checks, Reconfiguration, Task Resubmission, Masking, etc. [5–8]—to tackle faults at various levels either in reactive or proactive fashion.

Cloud computing entails the dynamic allocation of resources and the use of data centers that are often dispersed geographically. The hypervisor, also known as the virtual machine monitor (VMM), is a high-level monitoring unit that splits the server's available resources into virtual machines (VMs) or virtual nodes (VNs) and monitors their performance and availability. Single or many VMs are assigned to run the submitted application based on the user's request. The benefit of utilizing a virtual machine is that it allows users to run applications on a variety of operating systems, IDEs, and software environments. In most cases, the virtual infrastructure management (VIM) module of cloud computing manages resource pooling, physical and virtual resource management, and other tasks [7].

A cluster is formed using a group of different hosts or servers. Here, we consider clusters as assets of servers for better generalization. Cluster allows cloud service providers to assign VMs to virtual clusters in a dynamic mode based on SLA or user request. Such prior knowledge for cloud service providers is very much necessary to handle dynamic allocation of virtual machines.

In this work, we propose an intelligent fault-tolerant mechanism that performs the following tasks: (a) detecting VM failure due to the higher response time of a node, byzantine fault, and performance fault; (b) optimizing checkpoint interval time; and (c) using asynchronous checkpoint/restart method to model the cloud service execution. In our cloud model, fault tolerance procedure is illustrated in Figure 1. At the beginning, tasks are submitted by the users. The cloud supervisor forms the virtual clusters of hosts and performs allocation of tasks to virtual machines (VM) along with monitoring of VMs and hosts. Virtual machines will start executing the allotted tasks along with checkpointing it at the optimized regular interval of time that is derived from the optimization algorithm.

If a node response time exceeds the response time defined in the QoS requirement, it is halted and all the tasks are restarted on another host. If a virtual machine fails, all the tasks running on the virtual machines will be restarted on other virtual machines from their most consistent checkpoints. Byzantine faults are detected as described in Section 3.1.1. The node in which byzantine fault is detected will be halted, and another virtual machine is launched. Log-based recovery mechanism is implemented to optimize the restart process of the tasks. It is noted that there will be overhead in identifying different types of faults and finding the most consistent checkpoint to restart the tasks.

The rest of the paper is organized as follows: Section 2 presents literature survey, the proposed method is discussed in Section 3, Section 4 gives evaluation of the proposed method with experimental setup and results, and lastly conclusion is provided in Section 5.

2. Literature Review

This section presents some of the work done by researchers.

Authors in [9] proposed a fault-tolerant VM placement, where fault tolerance is implemented using VM replication technique. Here, based on VM requirements, different numbers of replicated copies are used. Each physical machine has its own requirement or constraint, and the replicated copies of the same VM cannot be placed on the same physical machine. The integer linear programing method is used here to handle VM replica placement. In [10], to increase the reliability of the system, a checkpointing/restart mechanism was proposed along with a replication scheme. The development of a fault-tolerant system assures the reliability and continuity of services. Checkpointing is the most susceptible in the event of a higher failure rate since the checkpointing file will become inaccessible if the computer that stores it fails, rendering the failed job unrecoverable. Hence, a replica of the checkpointing file is maintained to improve the reliability. A checkpoint and replication based fault tolerance technique was developed [11]. The work focuses on MapReduce framework in cloud, where proactive based fault tolerance is used to recover from the fault.

Cloud service reliability enhancement through optimization of VMP was developed by Zhou et al. Three algorithms are used in this method. Based on the network topology, the first algorithm chooses an acceptable selection of VM-hosting servers from a potentially large collection of possible host servers. With K-fault-tolerance assurance, the second algorithm develops an appropriate strategy for placing the primary and backup VMs on the specified host servers. Finally, to solve the task-to-VM reassignment optimization issue, which is defined as finding the greatest weight matching in bipartite graphs, a heuristic is utilized. In [13], an (m, n)-fault tolerance virtual machine placement for cloud data center was proposed. m represents the number of edge switches, and n denotes the host servers. K-fault-tolerant replication strategy was used to enhance reliability of the application or services. The first step is to recast the issue as an integer linear programming problem and demonstrate that it is NP-hard. Second, to address the integer linear programming issue, the differential evolution (DE) technique is implemented. Authors in [14] proposed a unique execution time prediction model that takes into account execution events that other multilevel checkpointing models did not include. The relationship between the system failure rates, checkpoint/restart overhead, and time between consecutive checkpoints is complicated, and determining the ideal time between checkpoints is a difficult task. The work explains how the proposed model can be used to set checkpoint intervals and why these execution events are essential to consider.

In [16], a fault-tolerant cloud computing service based on checkpointing is proposed. The fault tolerance service employs semicoordinated checkpointing, which reduces the time spent in the coordination phase and thereby reduces the amount of energy consumed and overhead. Results showed that the proposed approach also lowers the expense of a rollback. Bansal et al. [17] introduced the WQR-FT fault-tolerant WQR method, which employs a group manager to guarantee the existence of a certain number of copies in the system. Checkpointing adds overhead, which might lengthen the execution time [18, 19]. The checkpointing method (protocol), checkpointing storage, or recovery process can contribute to this cost [20].

3. Proposed Work

Many existing methods do not focus on virtual machine (VM) failure that occurs due to multiple factors like higher

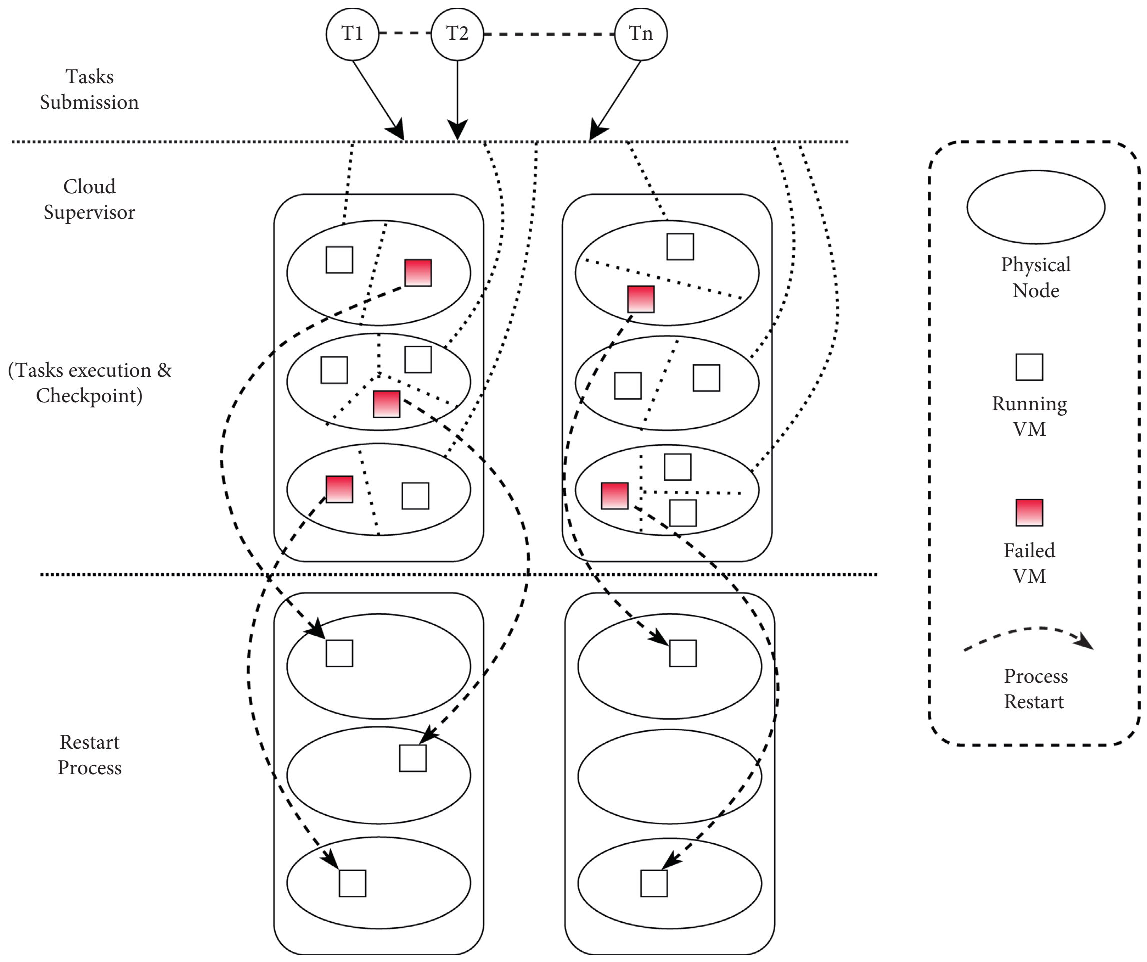

Figure 1: Fault tolerance procedure.

response time of a node, byzantine fault, and performance fault, and existing methods also ignore the optimization during the recovery phase. The proposed approach for fault tolerance in the cloud data center involves three phases. Phase 1 focuses on finding VM failure. Here, few algorithms are proposed to detect VM failure due to higher response time that occurs at the virtualization layer of the cloud, and even the byzantine faults are also detected. Phase 2 also describes the proposed algorithm for intelligent fault-tolerant mechanism cloud data center which also involves the checkpoint interval time calculation process. In phase 3, asynchronous checkpoint and optimized recovery process using log-based mechanism is discussed. Figure 2 shows the working principle of the proposed model.

3.1. Phase 1: Detection of Different Types of Faults

3.1.1. Byzantine Fault Detection Using Checksum Validation. To detect byzantine faults, we have used the SHA-2 algorithm. SHA-2 is one of the novel hash functions used in different fields. SHA-256 uses a 256-bit hash value. Hash value is computed using eight 32-bit words. SHA-256 checksum can also be used in cloud platforms.

In cloud environment, when a node uses the TCP/IP protocol to connect to another node, it is expected to produce a checksum, so such nodes are automatically equipped with SHA-256. The nodes which are connected to other nodes through IP protocol are termed as internodes.

In this work, every node in the cloud environment performs the checksum. The checksum of a particular data block is always unique and does not clash with the result of another data block. As a result, when a node is provided with a message and fails to produce the necessary checksum, the node can be identified as erroneous and compromised. Malicious nodes are discouraged from altering the checksum findings because reconstructing the original data from the checksum or conducting collision analysis is generally a time, space, and cost constrained task. SHA-256 checksum computation on arbitrary datasets is simple, easy, and feasible. Byzantine nodes frequently produce genuine-looking output that is incorrect owing to byzantine fault-induced miscalculation.

3.1.2. Checksum Prerequisites. In the cloud environment, a cloud monitoring (supervisor) node is expected to send the message M to k number of internodes automatically and

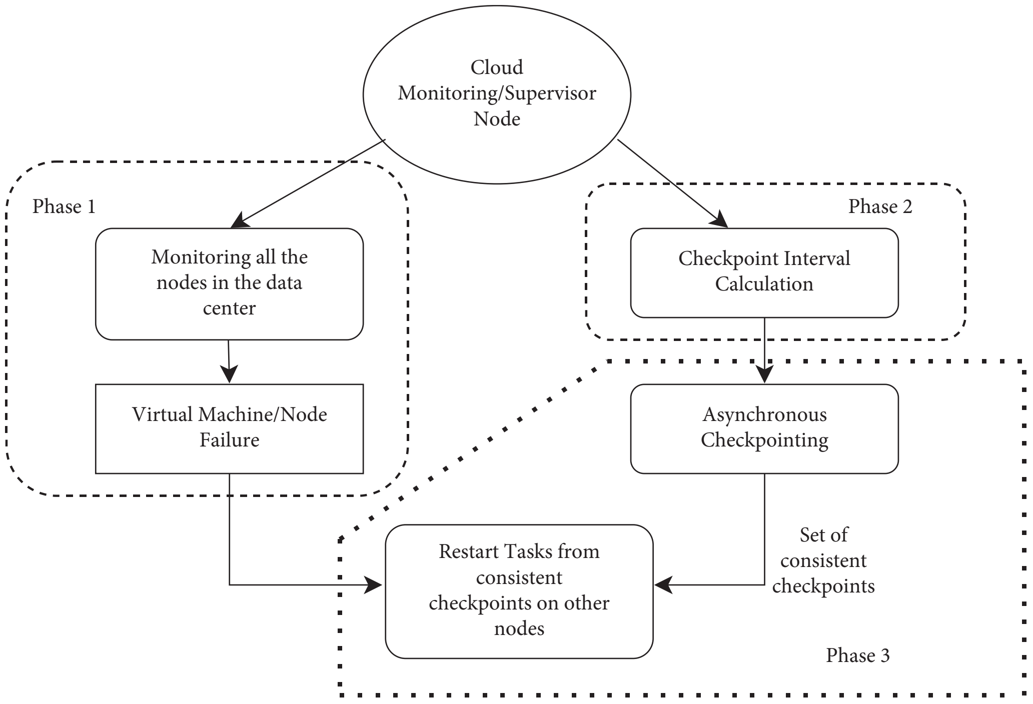
Cloud Monitoring/Supervisor Node
Phase 1
Monitoring all the nodes in the data center
Virtual Machine/Node Failure
Phase 2
Checkpoint Interval Calculation
Asynchronous Checkpointing
Set of consistent checkpoints
Restart Tasks from consistent checkpoints on other nodes
Phase 3

Figure 2: Work Process of the proposed model.

receive the checksum $\{C_1, C_2, \ldots, C_k\}$ in time $\{T_1, T_2, \ldots, T_k\}$. 512 bits is the size of the standard message for SHA-256, and the resultant checksum is 256 bits. We consider a supervisor node that has a precomputed checksum C in time T. Next, we compare set $P\{C\}$ with $Q\{C_1, C_2, \ldots, C_k\}$.

To compare checksum, if P and Q are the sets and Q's every element is also P's element, then $Q \subseteq P$; i.e., Q is a subset of P; hence,

$$Q \subseteq P, \quad \text{if } \forall y \left(y \in Q \longrightarrow y \in P \right). \tag{1}$$

If Q is not a subset of P, then one or more elements of Q exhibit processing error, hence the difference in the checksum.

If the set Q contains no element of P, then it is a null set $\{\}$ represented by $\varnothing$; i.e., $P \cap Q = \varnothing$. This means that the entire set of observed checksums is incorrect, so the entire set of observed nodes is compromised. This also may reflect that the supervisor node itself is compromised.

If there exist set of checksums in set Q which is produced by erroneous nodes, then

$$\frac{P}{Q} = \{y\colon\ y \in P | y \notin Q\} \longrightarrow \text{set of wrong checksums}. \tag{2}$$

Before any application begins execution in cloud, supervisor node selects message M, generates checksum $P\{C\}$, sends it to k nodes automatically, and receives the checksum $Q\{C_1, C_2, \ldots, C_k\}$ in time $\{T_1, T_2, \ldots, T_k\}$.

If $Q \subseteq P$, if $\forall y (y \in Q \longrightarrow y \in P)$, then we record the response time, i.e., transit time + processing time as the set R $\{T_1, T_2, \ldots, T_k\}$.

3.1.3. Algorithms for Detection of Different Types of Faults. Cloud computing delivers services to users maintaining QoS as mentioned in the SLA. Response time and QoS delay are among the QoS metrics which are associated with all the cloud nodes. Supervisor nodes monitor the set of nodes that meets the SLA.

In Algorithm 1, if the response time for any node exceeds QoS response time, then node is checkpointed and Algorithm 2 is called.

Algorithm 2 submits the message M to the operating node. If the operating node produces a checksum which is not matching the C, then it shows a checksum error denoting byzantine fault. If the fault is detected, the algorithm will shut down the node and start a new virtual machine. If no fault is detected, then the set $S\{T_1, T_2, \ldots, T_j\}$ is compared with R $\{T_1, T_2, \ldots, T_k\}$.

Consider a function f with set R and partially ordered set S as subset, an element s of S is upper bound of f if $S \geq f(R)$ for each r in R. If this holds good for at least one value of r, then it indicates that variation in the delay experienced is high or extreme, and it indicates performance fault; hence, the node is shut down after the transfer of workload at previous checkpoint as depicted in Algorithm 3.

3.1.4. State Transition for Checksum. Figure 3 shows the state transition diagram for the virtual node. A node, after receiving the message M from the supervisor, calculates the checksum. Here, the initial state of the node is considered as 0. If the node fails to compute the expected checksum after receiving message M, there is an error and it enters a byzantine state (i.e., 1). From the byzantine state, it reaches state 2 with probability $p = 1$ where the node is shut down

```
**Input**: N operating nodes $\{N_1, N_2, \ldots, N_k\}$
**Output**: faulty node or normal node
for all operating nodes ($N$)
  if response_time of $Nj \geq$ response time in QoS
    then
      take the checkpoint
      call checksum_compare()
  else
    continue supervise
  end if
end for
```

ALGORITHM 1: Node failure due to higher response time.

```
**Input**: Message $M$ to all operating nodes $N$
for each $Nj$ in $N$
  **if $Cj \neq C$//byzantine fault**
  then
    halt $Nj$
    start new node as $Nj$ from recent
    consistent checkpoint
  else
  call delay_deflection_compare()
  end if
end for
```

ALGORITHM 2: checksum_compare() for byzantine fault detection.

```
for each operating node $Nj$
  Choose $Ti$ in $S$
  Copy corresponding $Ti$ in $R$
    if Ti in $S < Ti$ in $R$
      no fault//minimal delay variation
      call checkpoint_optimization()
    else if Ti in $S = Ti$ in $R$
      no fault//call checkpoint_optimization()
    else if $Ti$ in $S \geq$ upper bound in $R$
      shut down $Nj$
      start new node as $Nj$ from recent
      consistent checkpoint
    else
      call checkpoint_optimization()
    end if
    end if
    end if
end for
```

ALGORITHM 3: delay_deflection_compare().

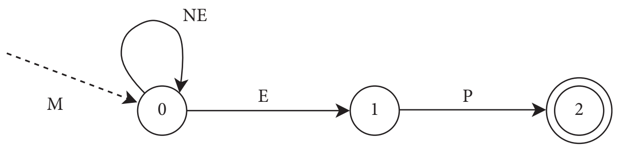

FIGURE 3: State transition for checksum.

and a new VM is started. If no error is detected, it remains in the same state. Here, *E* indicates the error, and *NE* indicates no error.

3.1.5. State Transition with Delay Variation. We consider three delay variations: (Δ) normal, high, and extreme. As shown in Figure 4, the initial state of the node is 0. If the delay variation is normal (*N*), the node remains in the same state; if the delay variation is high (H) or extreme (Ex), it denotes byzantine or performance failure, so the node takes transition to state 1. After this, the node is transited to state 2 where a shutdown of the node takes place.

3.1.6. Delay-Sensitive Server Scheduling (DSSS). The DSSS algorithm's goal is to maintain track of all of the servers that make up the virtual cluster. It is a lightweight model and can be integrated into cloud supervisor.

DSSS keeps track of the number of failed delay-sensitive tasks that surpass the QoS delays, as well as faults caused by VM failures, resource contention, and other factors. The count is then used to rank the server after each state interval, with the server with the fewest fault counts being at the top of the list. As a result, DSSS can help with dynamic job placement based on the server's performance. It may be also used to rate servers based on their prior performance and to keep track of the status of previous cluster implementation. Having such knowledge of prior performance can aid the management model in selecting the server for forming clusters to execute sensitive applications in an appropriate and dynamic manner. Notations used in DSSS algorithm are depicted in Table 1.

After the selection of the suitable server for processing the job, the next step is to apply an appropriate fault tolerance mechanism.

3.2. Phase 2: Checkpoint Interval Optimization. Checkpoint/restart optimization is a challenging task keeping checkpoint intervals at the optimal value. It aims at finding the time interval that is necessary to take checkpoints for the tasks. Let α represent the preset initial state monitoring interval. The optimization algorithm works in the following way: if a node does not exhibit delay variation or checksum error that happens when a node stays in the same state (0) as shown in Figure 5, then the interval value is incremented. If a node exhibits high or extreme delay variation and checksum error, the state interval is reset to initial.

3.2.1. Proposed Algorithm. To execute the tasks generated by users, our proposed algorithm (intelligent fault-tolerant mechanism, IFTM) uses several algorithms that have been discussed in Sections 3.1, 3.2, and 3.3 of this paper. The proposed algorithm identifies the VM failure due to higher response time, byzantine fault, and performance fault. It also calculates the optimal checkpoint interval time and restarts

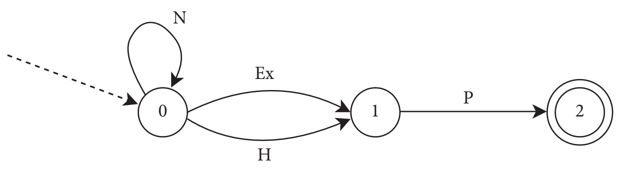

Figure 4: State transition for delay variation.

Table 1: Notations used in DSSS.

Notation	Meaning
R	User request/application
S	List of available servers
J_i	Task or job
F_{VM}	Failed virtual machine
C	Count of failed task
VM	Virtual machine
L_{DSSS}	List of servers sorted in ascending order
SI	State interval for fault tolerance

```
Input: $R$, $S$
Output: $L_{\text{DSSS}}$
Divide $R = \{J_1, J_2, \ldots, J_n\}$
for all $s$ is $S$ do
        if $s_j$ is assigned to $J_i$ then
        if $s_j$ not in $L_{\text{DSS}}$ then
            $s_j \longrightarrow L_{\text{DSS}}$
            $L_{\text{DSSS}} = L_{\text{DSSS}} + 1$
        end if
        end if
end for
for each $s_j$ in $L_{\text{DSSS}}$
    if $VM = F_T$
        $C = C + 1$
    else if $VM = F_{VM}$
        $C = C + 1$
    else $VM = F_{VM}$
        $C = C + 1$
        end if
    end if
end for
sort $L_{\text{DSSS}}(s, C)$
    for $j = 0$ to $n-1$
        if $s_j.c \leq s_{j-1}.c$ then
            swap ($L_{\text{DSSS}}[s_{j-1}]$, $L_{\text{DSSS}}[s_j]$)
        end if
    end for
```

Algorithm 4: DSSS.

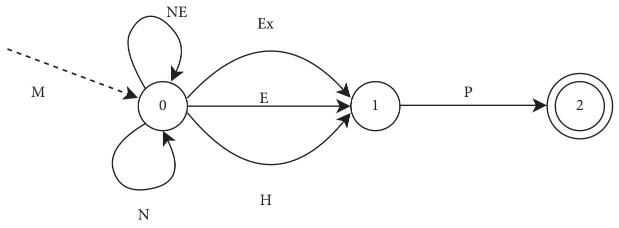

Figure 5: State transition for checksum and delay variation.

```
Set $I = \alpha$
Set $s = 0$
for each node $Nj$ in state 0
    if $\Delta C = \{NE$ or $N\}$
        $\alpha \leftarrow \alpha + I$
        Call delay_deflection_compare()
        Call checksum()
    else if $\Delta C = \{$ Ex or $H$ or $E\}$
        $\alpha = I$
        shut down $Nj$
        start new node as $Nj$
    end if
end for
```

Algorithm 5: Checkpoint_Interval_Optimization().

```
for each $Ji$
    for each $VMi$
        do
            supervise (Checksum, Delay Variation)
            DSSS()
            Checkpoint_Interval_Optimization()
            Asynchronous_checkpoint()
            Checksum_compare()
            delay_deflection_compare()
            if ($F_{VM}$)
                recovery_algorithm()
            end if
    end for
end for
```

Algorithm 6: IFTM.

the failed tasks using an asynchronous checkpoint/restart mechanism.

3.3. Phase 3: Asynchronous Checkpointing and Recovery.

There are two types of VM fault-tolerant methods that are often utilized. One is based on checkpoint-based and log-based rollback techniques. The other is based on the primary-backup paradigm, with incremental checkpoints as a feature [15].

In this work, fault tolerance is modeled using the asynchronous checkpoint and log-based rollback. The applications or the processes/tasks getting executed on the allocated VMs running concurrently were checkpointed independently. These checkpoints are taken independently without any synchronization among the processes, hence the lower runtime overhead during normal execution. If VM failure is detected or some of the tasks fail, then the recovery process is activated. Recovery process needs to iterate to find a consistent set of checkpoints, which is one of the limitations of this method. Figure 6 shows an example of checkpoints and global consistent recovery points for different processes. The recovery algorithm must search for the most recent consistent set of checkpoints before it initiates recovery.

As shown in Figure 4, three processes, Pi, Pj, and Pz, take checkpoint at {{*Ci*, 0}, {*Ci*, 1}}; {{*Cy*, 0}, {*Cy*, 1}}; and {{*Cz*, 0}, {*Cz*, 1}} respectively. When the process Pi fails, it rolls back to the previous consistent checkpoint {*Ci*, 1}. Rollback of process Pi to {*Ci*, 1} creates an orphan message M7, and it forces Pj to roll back to checkpoint {*Cy*, 1}. Since asynchronous checkpoints face a domino effect during recovery, to overcome that effect and to optimize recovery, we have used a log-based recovery mechanism.

During checkpoint and recovery, few assumptions are taken into account. Communication channels are considered to be reliable, having infinite buffers, and deliver messages in FIFO order.ff Triplet (S, *M*, MSG_SENT) represents the state of P. Process at state S receives the message *M*, and it moves to the state S1 and sends the message out. Two types of log storages, volatile and stable log, are used. After the execution of an event, the triplet is recorded without any synchronization with other processes. Local checkpoints consist of a set of records that are first stored in volatile log and then moved to stable log. During recovery, Algorithm 7 is used.

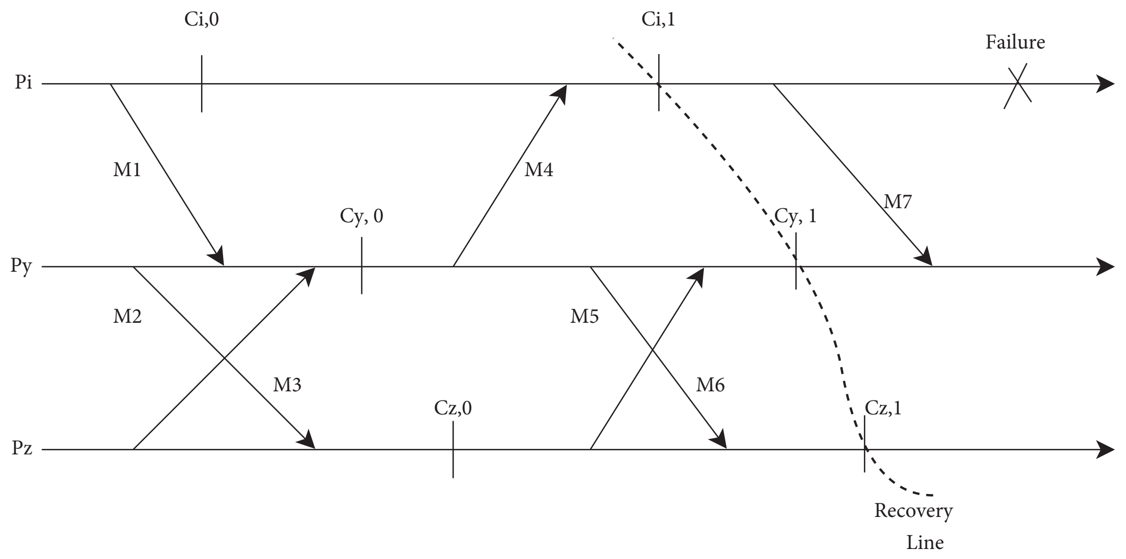

Figure 6: Asynchronous checkpointing and recovery.

```
Process $P_a$ accomplishes the following:
part 1
  if $R_a$ then
    $CP_a$: = latest event logged in stable storage
else
    $CP_a$: = latest event that took place in $P_a$ {can be in volatile storage or stable storage}
    end if
part 2
for $i = 1$ to $K$
  do
    for each neighbor process $q$ do
      calculate $SD_a \longrightarrow_b (CP_a)$
      send a ROLLBACK($a$, $SD_a \longrightarrow_b (CP_a)$)
      message to $P_b$
    end for
  for every ROLLBACK ($b$, $O_c$) message
    received from a neighbor $b$ do
    if $RC_a \leftarrow_b (CP_a) > O_c$ // indicates presence // of orphan message
      then
        find the latest event $e$ such that
          $RC_a \leftarrow_b (e) = O_c$
        $CP_a$: = $e$
      end if
    end for
end for
```

Algorithm 7: Rollback recovery.

Notations used in the algorithm are as follows:

$RC_a \leftarrow_b (CP_a)$ indicates the number of messages received by process P_a from P_b, from the beginning of the computation to checkpoint CP_a.

$SD_a \longrightarrow_b (CP_a)$ indicates the number of messages sent by process P_a to P_b, from the beginning of the computation to checkpoint CP_a.

R is the number of process recovered after failure.

K is the number of processes.

O_c is orphan message.

Here, a set of consistent checkpoints are selected from the set of checkpoints based on the number of messages sent and received.

3.4. Model Execution with Checkpoint Mechanism. The checkpoint procedure is regarded as deterministic, and the cost of a checkpoint is solely determined by the amount of work already completed. Let W be the workload and v be the number of checkpoints. $W_1, W_2, W_3, \ldots, W_v$ are the amount of work between each checkpoint such that $\sum_{q=1}^{v} W_q = w/\beta m$, where $ß$ represents the overhead factor $(0 \leq \beta \leq 1)$ and m denotes the number of virtual machines. W_q is the amount of work done between checkpoint number $q-1$ and q. Let $C(F_q)$ represent the checkpoint cost after quantity of work F_q, where $F_q = \sum_{i=1}^{q} W_i$, where W_i denotes the quantity of work that must be completed prior to each checkpoint. R represents the restart process cost before q^{th} checkpoint, denoted as $R(F_{q\text{-}1})$, where $F_{q-1} = \sum_{i=1}^{q-1} W_i$.

It is assumed that no failure happens during the rollback recovery process. The total execution time can be represented as

$$E(P_k) = E(P_{wf}) + \sum_{t=1}^{k} \sum_{q=1}^{v} u_q \cdot E(P_{tf}), \tag{3}$$

where

k is the number of processes.

P_{wf} is a process without failure.

P_{tf} is a process with failure and recovery.

$u_q = W_q + C(F_q) + R(F_{q-1})$.

4. Simulation Results

Experimental setup, performance metrics, and experimental results are discussed in this section.

4.1. Experimental Setup. CloudSim toolkit simulator is used to evaluate the proposed method. We have used real workload traces (log) files from PlanerLab which is part of CoMon project having CPU utilization from more than 1000 VMs running on different hosts in more than 500 locations across the world. We have used 4 types of VMs, micro, small, medium, and large instances. 800 heterogeneous hosts, which belong to HP ProLiant G4 and HP ProLiant G5 category, are used. The number of tasks generated is between 100 and 1000.

Faults are generated using the FaultGenerator class in CloudSim. VM fault is induced by shutting down the VM, resource contention fault is simulated by reducing the resource capacity, and a modified FaultGenerator class is used to simulate byzantine fault or performance fault.

4.2. Performance Metrics and Results. In the proposed method (intelligent fault-tolerant mechanism), BFD is used as a VM placement technique. Here, the active hosts are categorized according to their power efficiency, and the most efficient ones are favored. In BFD, the host is better than other host if its power efficiency is greater than the other host and lesser in fault counts. The proposed method is compared with checkpointing technique without FCFS..

The following metrics are used to evaluate the performance of the proposed and other methods.

4.2.1. Power Consumption. It represents the total amount of energy utilized by all of the data center's physical machines (PMs). The linear cubicle power consumption model is used to calculate the energy consumption of PMs. In this power paradigm, the physical host's power consumption climbs linearly as CPU use rises. For the power model, we consider the following parameters.

$P_k^{\max}$: maximum power consumed when the host k is completely utilized.

P_k^{idle}: idle power value of the host k.

U_k : current CPU utilization of the host k.

T : total number of hosts in the data center.

The power consumption of host P_k can be expressed as

$$P_k = P_k^{\text{idle}} + \left(P_k^{\max} + P_k^{\text{idle}} * U_k^3\right). \tag{4}$$

Our goal is to reduce data center power usage, and subsequently we aim to minimize

$$\sum_{k=1}^{T} P_k = \sum_{k=1}^{T} \left[P_k^{\text{idle}} + \left(P_k^{\max} - P_k^{\text{idle}}\right) * U_k^3\right]. \tag{5}$$

Figure 7 shows the power consumption of both the methods. Here, the average power consumption of the proposed method is lesser compared to the nonoptimization method for the dataset planetlab/20110303 to planetlab/20110420.

4.2.2. Makespan. It is the total execution time required to process all the tasks. Since the faults are simulated, few tasks may fail and get restarted from the identified checkpoint, causing the completion of the tasks to take more time than expected. Makespan is one of the key performance metrics to evaluate the algorithms/methods. Figure 8 shows the execution time of the proposed method and nonoptimization method. As shown in the figure, average execution time of the proposed method using optimization technique is less by 25% compared to nonoptimization method.

Figure 9 and 10 show standard deviation of execution time of the proposed method and nonoptimization method with VM selection. The standard deviation value falls within the range of 0.005 to 0.012 seconds in the proposed method, and in the nonoptimization method it ranges from 0.009 to 0.021 seconds.

The comparison of number of tasks completed by the proposed method and nonoptimization method is replesented in Figure 11 by varying the number of tasks from 100 to 1000. Total number of tasks completed by the proposed method is higher compared to nonoptimization method. Consequently, reliability is high because of a lesser number of failed tasks in the proposed method. Reliability can be measured as the inverse of the failure probability. The more number of tasks completed signifies that the reliability of the system is high.

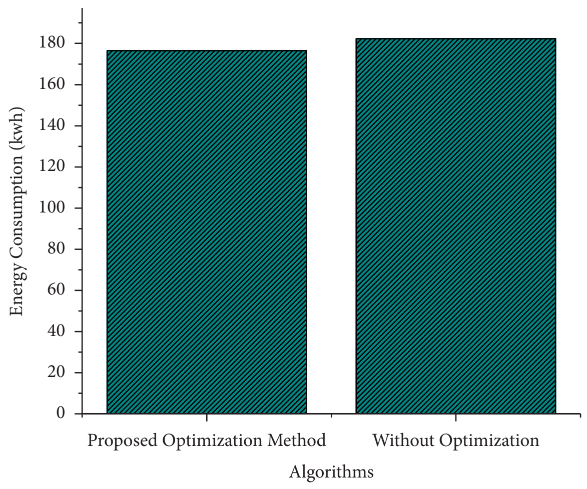

Figure 7: Average power consumption.

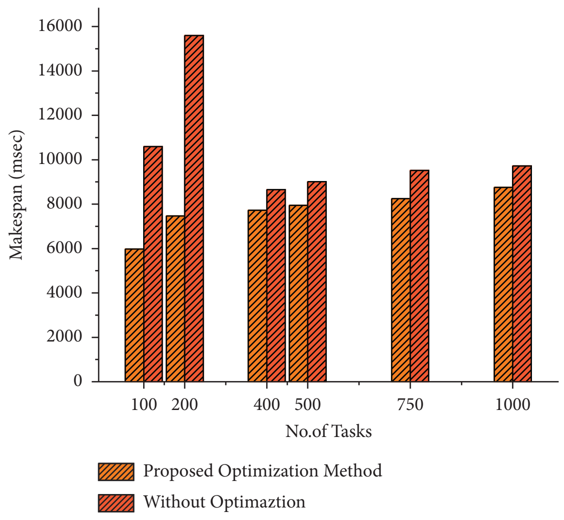

Figure 8: Makespan of different methods.

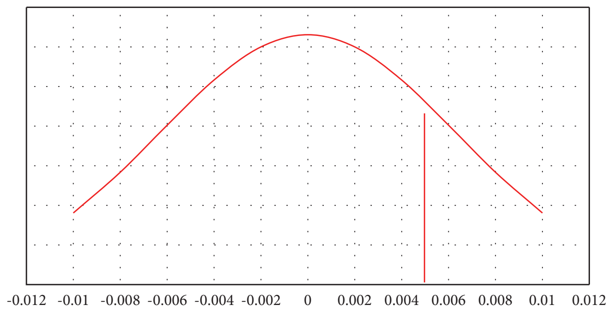

Figure 9: Proposed method execution time, VM selection standard deviation.

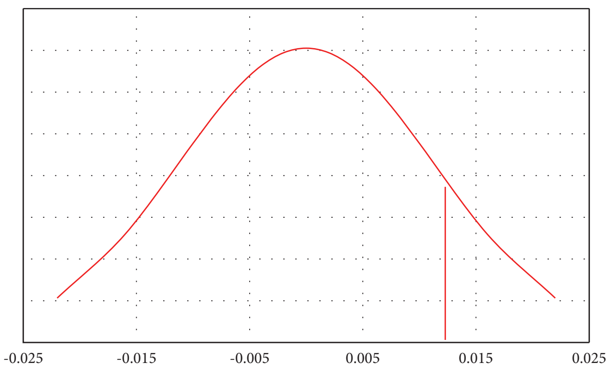

Figure 10: Nonoptimization method execution time, VM selection standard deviation.

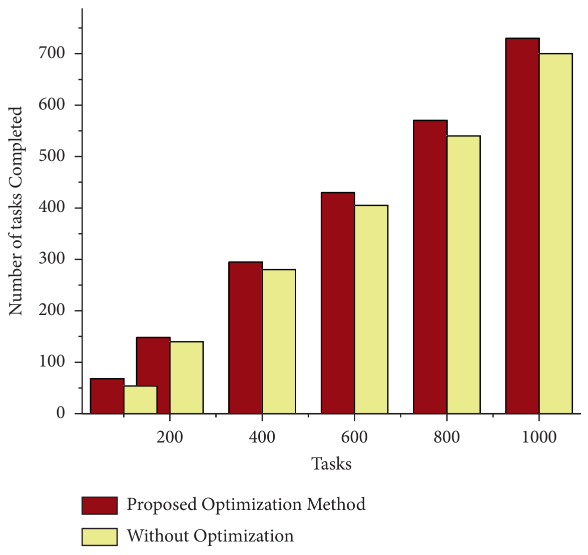

Figure 11: Number of tasks completed.

5. Conclusion

This paper aims at enhancing the reliability of cloud services through fault-based mechanism. The proposed approach is a three-phase process: phase 1 is the detection of virtual machine (VM) failure due to the higher response time of a node, byzantine fault, and performance fault. The checkpoint optimization algorithm in phase 2 finds the suitable time to mark checkpoints periodically while executing the tasks. Finally, in the checkpoint and recovery phase, in case of failure, the backup and recovery algorithm finds the optimal global checkpoint to restart the failed tasks. The evaluation result using a real-time dataset shows that the proposed method gives a better fault-tolerant solution decreasing the execution time and energy consumption and increasing reliability compared to the non-optimization method. Our future work includes developing fault tolerance mechanism using other reactive techniques and testing on different workload traces [12].

Data Availability

No dataset is required.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Authors' Contributions

Madhusudhan H. S., Satish Kumar T., and Punit Gupta developed the theory and proposed the model. Dr. S. M. F. D. Syed Mustapha and Rajan Prasad Tripathi worked on data preparation and model for the dataset and verified the analytical methods. All authors discussed the results, contributed to the final manuscript, and agreed to the submitted version. All authors confirm sole responsibility for the following: study conception and design, data collection, analysis and interpretation of results, and manuscript preparation.

References

[1] M. A Mukwevho and T. Celik, "Toward a smart cloud: a review of fault-tolerance methods in cloud systems," *IEEE Transactions on Services Computing*, vol. 14, no. 2, pp. 589–605, 2018.

[2] W. Ahmed and Y. W. Wu, "A survey on reliability in distributed systems," *Journal of Computer and System Sciences*, vol. 79, no. 8, pp. 1243–1255, 2013.

[3] S. Hernández, J. Fabra, P. Alvarez, and J. Ezpeleta, "Using cloud-based resources to improve availability and reliability in a scientific workflow execution framework," in *Proceedings of the Fourth International Conference on Cloud Computing, GRIDs, and Virtualization, Cloud Computing*, pp. 230–237, International Academy, Research, and Industry Association IARIA, Valencia, Spain, May 2013.

[4] M. Nazari Cheraghlou, A. Khadem-Zadeh, and M. Haghparast, "A survey of fault tolerance architecture in cloud computing," *Journal of Network and Computer Applications*, vol. 61, pp. 81–92, 2016.

[5] A. Bala and I. Chana, "Fault tolerance-challenges, techniques and implementation in cloud computing," *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 1, Article ID 288, 2012.

[6] L. P. Saikia and L. D. Yumnam, "Fault tolerance techniques and algorithms in cloud computing," *International Journal of Computer Science & Communication Networks*, vol. 4, no. 1, pp. 01–08, 2014.

[7] Y. M. Essa, "A survey of cloud computing fault tolerance: techniques and implementation," *International Journal of Computer Applications*, vol. 138, 13 pages, 2016.

[8] T. Zaidi, "Modeling for fault tolerance in cloud computing environment," *Journal of Computer Sciences and Applications*, vol. 4, no. 1, pp. 9–13, 2016.

[9] C. Gonzalez and B. Tang, "FT-VMP: fault-Tolerant virtual machine placement in cloud data centers," in *Proceedings of the 2020 29th International Conference on Computer Communications and Networks (ICCCN)*, pp. 1–9, Honolulu, HI, USA, August, 2020.

[10] B. Meroufel and G. Belalem, "Optimization of checkpointing/recovery strategy in cloud computing with adaptive storage management," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 24, Article ID e4906, 2018.

[11] M. Raghuwanshi and G. Deepak Kumar, "Checkpoint and replication based fault tolerance for map reduce framework in

cloud environment," *International Journal of Engineering Science*, vol. 12, no. 2, pp. 1029–1036, 2018.

[12] A. Zhou, S. Wang, B Cheng et al., "Cloud service reliability enhancement via virtual machine placement optimization," *IEEE Transactions on Services Computing*, vol. 10, no. 6, pp. 902–913, 2016.

[13] A Zhou, S. Wang, C. H. Hsu, M. H Kim, and K. S. Wong, "Virtual Machine Placement with (m,n)-Fault tolerance in cloud data center," *Cluster Computing*, vol. 22, no. 5, pp. 11619–11631, 2019.

[14] D. Dauwe, S. Pasricha, A. Anthony, and S. Howard Jay, "An analysis of multilevel checkpoint performance models," in *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 783–792, Vancouver, BC, Canada, May, 2018.

[15] W. Zhang, X. Chen, and J. Jiang, "A multi-objective optimization method of initial virtual machine fault-tolerant placement for star topological data centers of cloud systems," *Tsinghua Science and Technology*, vol. 26, no. 1, pp. 95–111, 2020.

[16] B. Meroufel and G. Belalem, "Service to fault tolerance in cloud computing environment," *WSEAS Transactions on Computers*, vol. 14, no. 1, pp. 782–791, 2015.

[17] J. Bansal, S. Rani, and P. Singh, "A fault tolerant scheduler with dynamic replication in desktop grid environment," *Int J Emerg Trends Technol Comput Sci*, vol. 3, no. 1, pp. 170–175, 2014.

[18] B. Yang, F. Tan, and Y.-S. Dai, "Performance evaluation of cloud service considering fault recovery," *The Journal of Supercomputing*, vol. 65, no. 1, pp. 426–444, 2013.

[19] S. Di, Y. Robert, F. Vivien, D. Kondo, C.-Li Wang, and F. Cappello, "Optimization of cloud task processing with checkpoint-restart mechanism," *Networking, Storage and Analysis*, in *Proceedings of the International Conference on High Performance Computing*, pp. 1–12, Atlanta, Georgia, November, 2013.

[20] S. Al-Kiswany, M. Ripeanu, S. S. Vazhkudai, and A. Gharaibeh, "STDCHK: A checkpoint storage system for desktop grid computing," in *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, pp. 613–624, Beijing, China, June, 2008.