ISSN: 2278 – 7798
International Journal of Science, Engineering and Technology Research (IJSETR)
Volume 2, Issue 4, April 2013

Failure Prediction for Scalable Checkpoints in Scientific Workflows Using Replication and Resubmission task in Cloud Computing

P.Palaniammal, R.Santhosh

Abstract— **The workflow scheduling system is to schedule the workflows within the user given deadline to achieve a good success rate. Workflow is a set of tasks processed in a pre-defined order based on its data and control dependency. Scheduling these workflows in a computing environment. To overcome these failures, the workflow scheduling system should be fault tolerant. The fault tolerance by using replication and resubmission of tasks based on priority of the tasks. The replication of tasks depends on a heuristic metric which is calculated by finding the tradeoff between the replication factor and resubmission factor. To achieve the historic lack of success trace data of the target environment. To solve the software support what you think problem unavailability of the resources, The Node Deficit of success Using(Independent Platform Management Interface) IPMI, Predicting Unattainability. Simple-based scheduler which indicates resources based on their predicted contemporary CPU time. This scheduler accomplishes high level of the CPU speed that will complete the application.**

Index Terms—**Cloud computing, Scheduling, Workflows, Replication, Resubmission, Fault tolerance.**

I INTRODUCTION

Cloud computing has emerged as a global infrastructure for applications by providing large scale services through cloud servers. The services can be either storage service or computation service. These service scan be configured dynamically by making use of virtualization. Any application in cloud computing environment can be represented by a workflow. This computing environment still cannot deliver the quality, robustness and reliability that are needed for the execution of various workflows The scientific community has shown increasing interest in researching new high-performance distributed computing platforms for accommodating and meeting the ever increasing computational and storage requirements of grand-challenge e-science applications. Whether they are metacomputers, computational Clouds, or more recent Cloud systems, the unfortunate reality is that all these environments still cannot deliver the quality, robustness, and reliability that are needed for widespread acceptance as tools to be used on a day-to-day basis by scientists from a multitude of scientific fields. The problem lies in the complex and dynamic nature of highly distributed systems, which exhibit high failure rates that the environments running on top have to be able to cope with [1]. With a system that has a low tolerance for faults, users will regularly be confronted with a situation that makes them lose days or even weeks of valuable computation time because the system could not recover from a fault that happened few seconds before the successful completion of their applications. This is of course intolerable for anyone trying to effectively use such environments and, as a consequence, scientists often prefer a slower solution that only uses their own local (and limited) computing resources, but which offer a higher reliability and controllability. We investigate a new method and algorithm to improve the fault tolerance of scientific workflow applications, which emerged in the last decade as one of the most successful paradigms for programming e-science applications in highly distributed environments such as Clouds and Clouds. Currently, there are two fundamental and widely recognized techniques to support fault tolerance in distributed environments: resubmission and replication [3]. Resubmission tries to execute a task after a failure which can significantly delay the overall completion time in case of multiple repeating failures. Replication submits several copies of the same task in parallel on multiple resources which suffers from potentially large resource consumption. To find a compromise balance between these two complementary techniques, we propose in this paper a new algorithm called Resubmission Impact (RI) that tries to establish a metric describing the impact of resubmitting a task to the overall execution time of a workflow application, and to adjust the replication size of each task accordingly. because of different failures link failure, failure providing the service, malicious code in the executing node. In contrast to our approach, other existing fault tolerance methods usually rely on precise prediction of failure the probabilities for a task on a resource in a certain time slot [4], [5].

Manuscript received Feb 10, 2013.
P.Palaniammal, Department of Computer Science and Engineering, Karpagam University, Coimbatore, Tamilnadu.

R.Santhosh, Department of Computer Science and Engineering, Karpagam University, Coimbatore, Tamilnadu.

This prediction is very hard to achieve and requires not only a deep knowledge of the behavior of the target infrastructure, but also years of trace data of the specific system. In contrast, we propose a generic method for fault tolerance which can be immediately applied to any workflow in any distributed computing environment, even if no historic trace data to build an environment-specific fault model is available. Real-life users typically want to know an estimation of the execution time of their application before deciding to have it executed. In many cases, this estimation can be considered to be a soft deadline that shall be satisfied with some probability [6]. In other words, if a soft deadline is not met, the results are still useful, the only difference being in the Quality of Service (QoS) provided. If a system regularly fails to meet soft deadlines by large amounts, users will be dissatisfied with the QoS offered. In contrast to soft deadlines, other applications such as weather forecasting and medical simulations have hard deadlines which, once broken, make the results useless. We propose on top of the RI heuristic a dynamic enactment method that first proposes to the end user a realistic soft deadline and then monitors and dynamically reschedule the workflow to meet the deadline in the absence of failure models. computational clusters that comprise thousands of nodes and the underlying hardware capability to serve applications an order of magnitude larger than could previously be supported. Concurrently [4], P2P, and Public Resource Computing (PRC) [5] efforts are incorporating large numbers of individual machines to create useful wide area networks of resources with very different characteristics. The merger of these two models into clouds that contain both kinds of resources would seem to be a natural next step. Cloud and large-scale cluster computing's most significant impact will be on scalable applications that make use of a significant number of these resources simultaneously, both within a single large-scale cluster, and across organizations. As applications grow to use more resources for longer periods of time, they will inevitably encounter increasing numbers of node failures. Furthermore, differences in node failure characteristics will widen as more different kinds of resources join the same federated collection. Current fault tolerance solutions will fail to support high performance applications in this environment for two important reasons: (1) current check pointing solutions require centralized storage and are therefore inherently not scalable, and (2) schedulers do not consider failure characteristics in making placement decisions. The thesis of this paper is that scalable check pointing solutions, and schedulers that take advantage of information about the unavailability characteristics of different component resources, are necessary for supporting high performance distributed computing in current and future cloud environments. We describe emerging research in both of these directions, at several different levels.

1) Modern processors include interfaces to monitor the behavior and dynamic characteristics hardware components. [6] We believe that these characteristics could be used to predict imminent failure.

2) Cluster-wide check pointing solutions, can be built without reliance on centralized Checkpoint storage. [8]

Past behavior of resources in large-scale clouds can be used to categorize resources into classes, and can help predict their future availability. These predictions can help schedulers map applications onto resources to improve application performance and cloud utilization. [8] We describe our current work in each of these individual areas, and identify ways that they can be used together.

II.ASSOCIATED WORK

Considering faults in executing scientific workflows, we can divide existing techniques into three categories. The first category of techniques builds on extensive and complex failure models that aim to describe the failure probability of a task on a resource in a certain time interval. [7] describe an approach for combined fault tolerance and scheduling of workflow applications in computational Clouds. [9] describe fault tolerance techniques for running meteorological workflows on the TeraCloud taking into account check pointing, migration, and overprovisioning. Building such a model describing the failure probability of a task on a resource can be a very difficult task that often requires years' worth of traces of failure data about the specific target environment. On the other hand, commercial Cloud providers such as Amazon do not disclose any information regarding their infrastructure, and failure traces are often a closely guarded secret. The second category of techniques [8],tackles the problem with the help of Service Level Agreements (SLAs) or advance reservations. Techniques in this category usually rely on agreements with the end user or the resource providers, describing the QoS requirements for the execution of a given workflow. The workflow execution is accepted and started only if binding agreements with the resource providers can be reached. Given the strict criteria of QoS in the SLAs, this shifts the responsibility of dealing with faults to the resource providers. The problem with this approach is the lack of support for negotiating the required fine-grained QoS terms in today's Cloud/Cloud systems. The third category of techniques can often be found in current Cloud workflow systems [10]. To developed an adaptive Cloud middleware capable of recovering applications after a failure by restarting them from checkpoint files automatically. As described in our earlier survey [4], techniques in this category often deal with unreliable environments by resorting to fault tolerance mechanisms such as check pointing, task replication, and task resubmission without considering a failure model of the target system. This method has the advantage of applicability to new and unknown environments, however, it often leads to unnecessarily large resource consumption and to large differences between the expected execution time (as promised to the end user) and the real execution time. Our work belongs to this third category and brings two advantages over existing methods: 1) it reduces the resource consumption, and 2) it offers improved QoS by meeting soft deadlines. If the selected resources fail before the application Completes, however, then the importance of having correctly predicted the node with the lowest future load is diminished; schedulers should therefore also consider the future availability of candidate resources. This observation has led to a small number of efforts at resource availability prediction [10].

III.WORKFLOW FAULT TOLERANCE

The two most widely used fault tolerance mechanisms [4] are task replication and task resubmission. The idea behind task replication is that a replication of size r can tolerate r _ 1 failed tasks while keeping the impact on the execution time minimal. We call r the replication size. While this technique can help to successfully complete time-critical tasks, its downside lies in the large resource consumption. On the other hand, task resubmission with a maximum of s (re)submissions can also tolerate s _ 1 failures while keeping the additional resource consumption very low. The downside of resubmission is the potentially large degradation in total execution time upon a large number of failures. While replication still wastes a lot of resources in the absence of failures, it does not impact the workflow execution time as severely as resubmission does, especially in the case when there are sufficient fast resources available. Even though these two approaches have different drawbacks and advantages, they are complementary and can be used in cooperation: replication is a method suited to the workflow scheduling phase, while resubmission is mostly applicable during execution. Algorithm 1 presents a trivial baseline heuristic called replicate all that schedules a workflow by simply replicating each task a fixed amount of times indicated by a maximum replication size repack input parameter. This heuristic builds first a replication vector RV defining the replication sizes replication 2 RV for each task Ai 2 Assets, and initializes it with the maximum replication size parameter repacks (given by the user) for all tasks in the workflow. Then, it schedules the workflow including the replicas onto the available resources R by invoking heft replication that replicates each task according to the input replication vector RV and schedules the resulting workflow according to an extension of the immensity [9] algorithm. We designed heft replication as a generic function that will be used again. which works in two phases. In the first phase, it computes a rank for all the tasks based on the length of the critical path to the end of the workflow, and sorts them in descending order. In the second phase, it maps the tasks in descending order of their ranks to the resource giving the earliest completion time (ECT). After each task has been mapped, its parent tasks are checked and, if all children tasks have been mapped to a resource already, the previously set number of replicas of the parent are created and mapped to resources according to the ECT. The reason for this delayed creation of replicas is that each task should first be mapped onto the resource that delivers its ECT which is not influenced by the previously created replicas that may steal valuable resources.

a)resubmission impact heuristic

Even though replicate all can achieve good fault tolerance, it has two important drawbacks: 1) it makes intensive use of additional resources during replication, and 2) it increases the overall workflow make span if there are not enough free resources available. Therefore, it is necessary to find a compromise solution that balances the tradeoffs between resource consumption using replication on one hand and increase in execution time when using resubmission on the other hand. To solve this problem, we propose in this paper a new heuristic called RI. The idea behind RI is to establish a metric describing the impact of having to resubmit a task on the execution time of a workflow application. In addition to the workflow application workflow and the set of resources R, the RI heuristic receives as input the maximum replication count repacks and the maximum resubmission count remap, which have to be given by the user

IV.SCHEDULING FOR SOFT DEADLINES

The RI heuristic is able to schedule workflows with a high success rate while consuming a reasonably low amount of resources. However, since RI schedules the full workflow before it is executed, it cannot react to unexpected periods of high failures during execution and is therefore unable to adjust its replication size accordingly. This can lead to situations where the replication can fail and many tasks have to rely on resubmission. Consequently, the workflow make span can exhibit large deviations from the expected value, degrading the QoS for the end user. We therefore propose an extension to the RI heuristic that is able to provide the end user with a realistic estimation of the execution time in the form of a soft deadline. The workflow enactment responsible for submitting the tasks and transferring data according to the control flow and data flow dependencies is enhanced with a monitoring step after the completion of each task, followed by a heuristic that adjusts the RI values and reschedules the workflow remainder if the soft deadline is likely to be missed. In the following, we explain the enactment and

A)Rescheduling

As explained in the previous section, the new rescheduling heuristic is invoked if the enactment engine discovers that the real workflow make span is too far behind the scheduled workflow execution plan, which implies a heightened probability of missing the soft deadline presented to the user. The role of the rescheduling heuristic is to decrease the probability of violating soft deadlines by adjusting the amount of replication rep0 i of each remaining task Ai of a workflow as follows.

where Reid is the RI of task assets and repacks is the maximum replication size. shows the effect of using the resubmission multiplier. The higher the scheduling cycle c is, the higher the replication size becomes for tasks with small RI. In other words, with every scheduling cycle we start replicating tasks that have not yet been replicated and increase the replication size of tasks that are already subject to replication. This rescheduling heuristic, which uses (5) to adjust the replication size repay of each workflow task. Finally, heft replication is used again to schedule the remaining workflow tasks including the replicas.

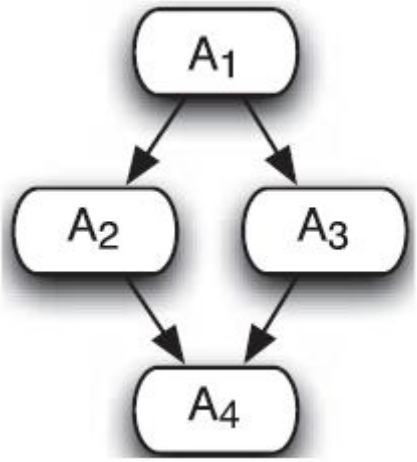

Fig 1.Sample workflow with ETC

b)Resubmission Impact

To evaluate the RI heuristic, we simulated an environment comprising 548 heterogeneous processors distributed over 12 Austrian Cloud clusters. We simulated a set of experiments summarized in Table 3 by varying the following parameters: workflow application, problem size maximum replication and resubmission counts, failure model, and scheduling heuristic. The workflows are created based on existing traces logged from real workflow executions in the Austrian Cloud. The cross product of these parameters result in over 20000 workflow executions accumulating a total of over 1.1 billion of processing hours. Fig. 4a shows the average success rate of the three techniques for the three failure models. We observe that the HEFT algorithm immediately fails after encountering the first failed task and is therefore not able to finish any workflow in the unstable resource environment. In the normal environment with a is able to finish only the shortest workflows, resulting in a success rate of 4 percent. Since many workflows in our setup have expected execution times of about 1 month, even in a stable environment with a HEFT stays at a low success rate of 13 percent. In contrast, the fault tolerant replicate all can increase the workflow success rate to 26 percent in the unstable, 75 percent in the normal, and up to 90 percent in the stable environment. We can further see that RI can reach almost the same average success rates as the replicate all technique: 28 percent for unstable, 73 percent for normal, and 89 percent for stable resources. To compare the workflow make spans in the three approaches, we need a method of normalizing them, since heft finishes only the small workflows with a short make span. Therefore, we utilize a metric called standard length ratio [6] defined as the ratio between the workflow make span and the B-level of the first task, calculated by the heft algorithm as the sum of the computation and communication costs of every task in the workflow critical path, averaged across all resources: Because a good scheduling algorithm chooses “the best” resources for every task, SLR is usually lower than where a low SLR value indicates better performance. We can see in Fig. 4b that RI and replicate all perform worse than heft for two reasons. First, task replication can saturate the resource environment to a point where the scheduler cannot choose the same resource for a certain task anymore, leading to another choice with less performance and higher execution time. Second, the fault tolerant approaches resort to task resubmission in the case of a failure in all task replicas, which directly impacts the workflow execution time and, therefore, the SLR. Replicate all has an average SLR of for unstable, for normal, and for 0 stable resources, while RI shows even better results. Unstable, for normal, and for stable resources. The reason for the better performance of RI compared to replicate all is due to the better resource usage, defined as the sum of the actual processor time used by all executed workflow tasks. Fig. 4c shows that the average resource usage of the RI heuristic over all examined replication and resubmission sizes is considerably lower than for replicate all, especially for the normal and stable resources. The high replication size used by replicate all consumes valuable resources that slowdown other parallel tasks that do not benefit from enough fast resources for being executed. RI, in contrast, tunes the replication size to a lower value based on the RI and leaves more free resources to the parallel tasks. Both approaches exhibit increased resource usage with more stable resources because they are able to successfully complete more workflows, To gain a deeper insight into the advantages of RI resource usage, we focus our attention on the resource waste representing the amount of resources used that took part but did not contribute to the completion of the workflows. We define the resource waste as the total processor time used by the failed tasks and all the tasks of failed workflows, independent of their success or failure. Displays the average resource waste over all examined replication and resubmission sizes for each of the three resource failure models. In the case of unstable

resources, many replicas of a task are needed to cope with the high failure rate. In many cases, even the highest selected maximum replication size will not be enough to overcome the high failure rate. In this case, the system resorts to task resubmission to avoid task failure as much as possible. As the resources get more stable, the ratio of unnecessary replications gets higher because more replicas have the chance to successfully finish and, therefore, the resource waste increases. For stable resources, replicate all shows an increase of over 200 percent in resource waste compared to the unstable case, increasing the success rate of the workflow executions. When analyzing our new RI heuristic, we observe that the resources wasted percent less than replicate all in the unstable case and up to lower in the normal and stable cases.

V.PREDICTING FAILURE

We are exploring two different methods for predicting the failure and unavailability of resources, as described below.

A) Predicting Node Failure Using IPMI

To help reduce the amount of checkpoint data, and to aid in the overall maintenance and monitoring of a large computational cluster, an accurate yet light-weight tool must monitor for failed nodes and provide data useful for determining likely imminent failures.

B)Predicting Unavailability from Past Behavior

We have generated some initial results that indicate that nodes fail differently from one another, and that their failure is somewhat predictable. Our approach analyzes a one month trace of 435 machines in a university Condor pool.

c) Improved scheduling an recovery

Predicting which nodes may fail, and how they fail, can improve cloud and cluster schedulers and recovery techniques.

average execution time is lower for the predication based scheduler than for the CPU-based scheduler, and within of the semi-optimal scheduler. The prediction-based scheduler also obtained better throughput than the CPU-based scheduler, completing of the jobs, compared to the CPU based scheduler. These results illustrate considering resource availability characteristics can yield performance gains.

IV.SCALABLE CHECK POINTING ALGORITHM

we show that fault tolerance can be achieved with low overhead while still providing a high degree of resiliency to node failures. The work focuses on two challenges to periodic distributed check pointing: varying restart topologies, and eliminating the need for centralized checkpoint storage. This solution is ideal for large computational clouds as it does not rely on any form of centralized storage (including SAN, NAS, or centralized servers) for a globally accessible

Algorithm1. The proposed by scalable checkpoint using IMPI

ALGORITHM : scalable checkpoint using IMPI

Wf=(ASet,Dset,Wotk):scientific workflow

R : resource set;

Rep_{max} : maximum replication size;

Reguire: rex_{max} : maximum resubmission count;

C_{max} : maximum rescheduling cycle;

F_r : rescheduling factor;

1: function DYNAMICENACT(Wf,R, $rep_{max,}$ $res_{max,}$fr, $c_{max)}$

2: sched ← RESUBMISSIOMIMPACT(Wf,R ,rep_{max} ,$res_{max)}$

3: c ←1

4: $t_r^{(c)}$ ← tWf . f_r

5: t_d ← twf.(1+f_r . $\sum_{k=1}^{cmax}$)

6: **while** state(Wf)≠ completed **do**

7: for all A€Aset □ state(A)=ready□

(state (A')=completed: UA'€ pred(A)) **do**

8: **SUBMIT(**A,sched(A))

9: **end for**

10: t^{real}← WAIT (A) :state(A) = completed

11: if max(0.t_A^{real} − t_A) □ $t_r^{(c)}$ □ c □ c_{max} **then**

12: Aset' ← Aset' \ {A : state(A) = completed}

13: DSet ← ASet \ { (A_1, A_2, $Data_{ij}$) } :state

{(A_1 completed □ state (A_2)) = completed }

14. W f '= (ASet', DSet' , Work)

15. RI_RESCHEDULING (W f ', R, rep_{max} ,c)

16. C ← C + 1

17: $t_r^{(c)}$ ← t_{Wf} . f_r .$\sum_{k=1}^{c}$

18: **end if**

19: **if** state (A) = completed :U A€ ASet **then**

20: state(Wf) ← completed

21: **end if**

22: **end while**

23: **end function**

VI. SIMULATION RESULTS

Such a monitoring tool would allow an application to decide at runtime whether a failure is likely to occur, and consequently whether a checkpoint is necessary. Describe a cooperative check pointing strategy where in application-level check pointing is used to dynamically determine which checkpoints can be skipped at runtime. However, the nature of this approach requires a programmer to manually instrument all code with the appropriate check pointing primitives. We propose a monitor that will not rely directly on application level periodic check pointing, but can instead leverage hardware level failure indicators such as smart hard disk tools and IPMI [6] to generate checkpoint requests only as needed.

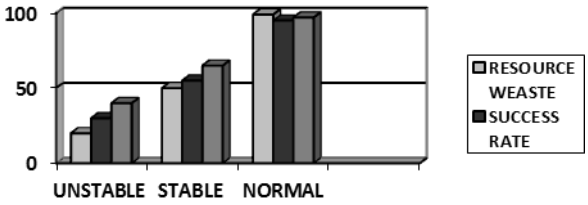

Fig 2.Comparison of RI and IMPI

We track their movement between five different availability states (Available, User Present, CPU Threshold Exceeded, Unavailable, and Becoming Unavailable), and classify resources based on their behavior in terms of these states, over time. A predictor can then anticipate the likelihood of the next state being reached by a particular resource, with significantly improved accuracy over a "random predictor" and over a predictor that does not distinguish between resources.[9]

A)Failure-Aware Scheduling

We have simulated job submission data that includes a mix of jobs that vary according to checkpoint ability half were checkpoint able, half had to start over upon node failure, checkpoint periods, and runtime. We simulated their execution on resources whose availability was determined from the trace data. We designed and tested a simple prediction-based scheduler, which chooses resources based on their predicted failure rate during the application's execution interval, their current CPU load, and their CPU speed.

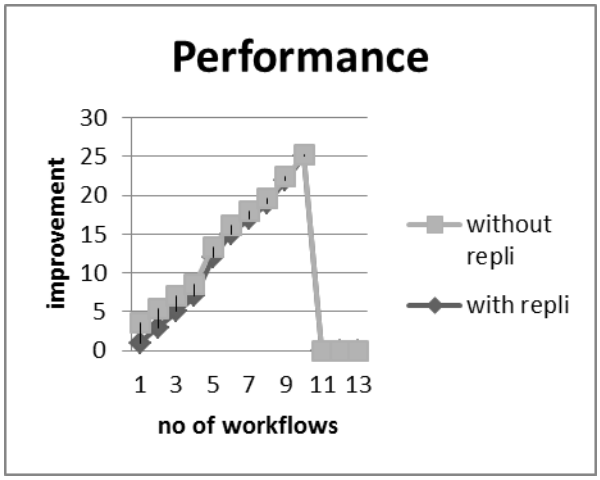

fig 3: failure-aware scheduling performance

We compared results against two other schedulers, (i) a Condor-like scheduler, which chooses the available resource with the highest CPU speed, and (ii) a semi optimal scheduler that given oracle knowledge about the future Availabilities of machines,. The checkpoint repository. chooses the machine with the fastest CPU speed that will complete the application without becoming unavailable We show that exceptionally low overhead can be achieved through storing checkpoints to local disks and replicating the checkpoints to a few "peer" nodes. Coupling this replication strategy with the ability to vary the restart topology allows for improved restart times, and eliminates the need for spare nodes. However, even lower overhead could be achieved by utilizing information captured through system monitoring tools as described.

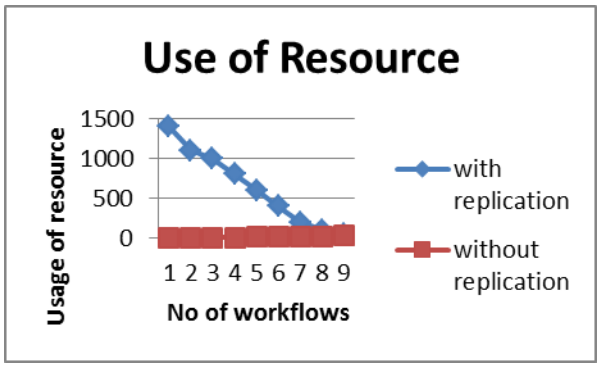

Fig 4: use of resource in predicting failure

This would allow for the reduction or elimination of the periodic nature of the check pointing system. In doing so, checkpoints could be initiated by any node participating in the computation only when needed. Further, data could be replicated in a structured manner to nodes with a low probability of failure, as indicated by the online monitors.

VII. CONCLUSION

The workflow scheduling application should be fault tolerant for failures. The main goal is to schedule workflows and execute these workflows within the deadline addressed for providing fault tolerance but without considering the user given deadline. The proposed work, This showed that FTWS algorithm produced good success rate. The heuristic is based on a combination of task replication and task resubmission using a new RI metric that describes the impact of task resubmission on the overall workflow make span. Predicting Node Failure Using IPMI To help reduce the amount of checkpoint data, and to aid in the overall maintenance and monitoring of a large computational cluster, Predicting Un availability from Past Behavior We track their movement between 5 different availability states (Available, User Present, CPU Threshold Exceeded, Unavailable, and Becoming Unavailable), and classify resources based on their behavior in terms of these states, over time Failure-Aware Scheduling We have simulated job submission data that includes a mix of jobs that vary according to checkpoint ability Scalable Check pointing The work focuses on two challenges to periodic distributed check pointing varying restart topologies, and eliminating the need for centralized checkpoint storage.

REFERENCES

[1] G. Kandaswamy, A.Mandala, and D.A. Reed, Fault Tolerance and Recovery of Scientific Workflows on Computational Grid Proc. IEEE Eighth Int. Cluster Computing (CGDRID ’08), pp. 777-782, 2008.

[2] J. Mickens and B. Noble, “Exploiting Availability Prediction in Distributed Systems,” Proceedings of NSDI’06, pp. 73–86, 2006
.

[3] T. Zheng and M. Woodside, “Heuristic Optimization of Scheduling and Allocation for Distributed Systems with Soft Deadlines,” Proc. 13th Int’l Conf. Computer Performance Evaluations, Modelling Techniques and Tools, pp. 169-181, 2003.

[4] H. Topcuoglu, S. Hariri, and M. Wu, “Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing,” IEEE Trans. Parallel and Distributed Systems, vol. 13, no. 3, pp. 260-274, Mar. 2002 Z. Shi and J.J.Donagarra, “Scheduling workflow applications on processors with different capabilities,” Future Gen. Comupter systems 22, pp.665-675, 2006.

[5] Intel, “The IPMI Specification,” http://www.intel.com/design/servers/ipmi/ spec.htm.

[6] J. P. Walters and V. Chaudhary, “AComprehensive User-level Checkpointing Strategy for MPI Applications,” Technical Report 2007-1, Department of Computer Science and Engineering, University at Buffalo (SUNY), 2007.

[7] Y. Zhang, D. Wong, and W. Zheng, “User-level Checkpoint and Recovery for LAM/MPI,” SIGOPS Oper. Syst. Rev., vol. 39, no. 3, pp. 72–81, 2005.

[8] M. Wieczorek, M. Siddiqui, A. Villazon, R. Prodan, and T. Fahringer, “Applying Advance Reservation to Increase Predictability of Workflow Execution on the Grid,” Proc. IEEE Second Int’l Conf. e-Science and Grid Computing (E-SCIENCE ’06), 2006.

[9] A. Luckow B. Schnor, „“Adaptive Checkpoint Replication for Supporting the Fault Tolerance of Applications in the Grid,“ Seventh IEEE International Symposium on Network Computing and Applications, 2008 IEEE.

[10] H.P. Reiser, M.J. Danel, and F.J. Hauck., “ A flexible replication framework for scalable andreliable .net services.,” In Proc. of the IADIS Int. Conf. on Applied Computing, volume1, pages 161–169, 2005

P.Palaniammal eived her BE degree in Computer Science and Engineerimg from Periyar Maniammai College of Technology for Women Thanjavur in 2011 and currently doing her M.E-Computer Science andEngineering degree in Karpagam University.

R.Santhosh received his B.Tech degree in Information Technology from K.S.R College of Technology in 2006, M.E degree in Software Engineering from Sri Ramakrishna Engineering College in 2009, M.B.A in Education Management from Alagappa University in 2011 and pursuing Ph.D in Computer Science and Engineering at Karpagam University. He is currently working as an Assistant Professor in the department of Computer Science and Engineering at Karpagam University