H-RADIC: The Fault Tolerance Framework for Virtual Clusters on Multi-Cloud Environments

Ambrosio Royo, Jorge Villamayor, Marcela Castro-León, Dolores Rexachs and Emilio Luque

CAOS – Computer Architecture and Operating Systems, Universitat Autònoma de Barcelona, Bellaterra (Cerdanyola del Vallès), Barcelona 08193, Spain
pabloambrosio.royo@e-campus.uab.cat,
{jorgeluis.villamayor,marcela.castro,dolores.rexachs,emilio.luque}@uab.cat

Abstract

Even though the cloud platform promises to be reliable, several availability incidents prove that they are not. How can we be sure that a parallel application finishes the execution even if a site is affected by a failure? This paper presents H-RADIC, an approach based on RADIC architecture, that executes a parallel application in at least 3 different virtual clusters or sites. The execution state of each site is saved periodically in another site and it is recovered in case of failure. The paper details the configuration of the architecture and the experiments results using 3 virtual clusters running NAS parallel applications protected with DMTCP, a very well-known distributed multi-threaded checkpoint tool. Our experiments show that the execution time was increased between a 5% to 36% without failures and 27% to 66% in case of failures.

Keywords: Cloud, Fault-Tolerance, High-Performance Computing, RADIC.

1. Introduction

We know that there aren't any computers small or big safe from failures, we have seen big cloud providers fail, Windows Azure had availability problems for the Olympic Games in 2012 [1], Amazon Web Services had been affected from the extreme weather [2] and by human errors [3], also Google Cloud Platform was attacked by a low-level software [4] or even Oracle Cloud's wide network error issue that was fixed by restarting the network [5].

Since communication with a cloud can be lost due to a wide range of possible errors, also causes loss of computational resources, power consumption and money; different authors have worked to prevent failures when there are parallel applications with message passing running on cloud environments; in the work of Villamayor et. al [6] where they propose Resilience as a Service (RaaS), a Fault Tolerant (FT) framework for High Performance Computing (HPC) applications running in a cloud, and Gomez et. al [7] propose a multi-cloud FT framework that was capable to continue working if any of the cloud sites fail and delivered the results on the due date. We have studied the previous work and took advantage of the elasticity (easy provisioning of "hardware") of cloud computing.

We propose to take the Redundant Array of Distributed Independent Controllers (RADIC) architecture and hierarchically scale it up to be applied to a fully automated, elastic FT framework for virtual clusters running on different cloud environments. Capable to protect applications running in private or public clouds from fatal fails like: loss of nodes during execution and loss of communication between clouds.

The next section presents some related state of the art work and the description of the RADIC architecture. In section 3, we will describe an overall detail of the H-RADIC architecture, fallowed up by section 4, a summary of the experiments done from the implementation of the H-RADIC architecture, after that section 5 with the conclusions, finishing with future work in section 6.

2. Background

In Japan, Bautista-Gomez et. al [8] proposed a low-overhead high-frequency multi-level checkpoint technique, that implement a three level checkpoint scheme that compensates for the overhead of the FT by dedicating a thread of execution per node.

Another group of IEEE members [9] in the USA, set up an online two-level checkpoint model for HPC, one level deals with logical problems such as transient memory errors and the other one deals with hardware crashes like node fails, contributing with an online solution that determines the optimal checkpoint patterns and doesn't required the knowledge of the job length in advance.

Egwutuoaha et. al [10], from Australia, approached the problem by not relaying on spare node prior of a fail, aiming to reduce the time and cost of the execution on the cloud.

And of course, the work of Gomez et. al [7] and Villamayor et. al [6], that we mention in the introduction.

2.1. RADIC Architecture

The RADIC architecture is transparent and automatic, therefore the application doesn't have to be modified to apply it and there is no need of human intervention, also it is elastic since it has the ability to add new nodes whenever one crashes [11]. It consists on implementing FT for message passing applications, by intercepting and masking error which detect and manage errors instead of ending the application. Taking advantage of the hardware redundancy, RADIC needs at least three physical nodes to work so the application doesn't have to stop and start again. It works with two distributed software RADIC components [12]: Observers and Protectors; there is a protector running in each node and one observer for every process running in the application, as shown in Fig. 1. Additionally, a table named RadicTable is used to store the relation between nodes, observers and protectors which is updated by the RADIC Components.

Observer: When the application is launched, a software instance is created and attached to every processor used in the program, its job is to perform checkpoints and intercept the messages of its processor and send it to the Protector to store it.

Protector: It's in charge to request the observers to perform checkpoints and store the checkpoint files in its own Stable Storage (SS), and also to verify that the node that it's protecting is working and, in the case, that it fails launch the latest checkpoint.

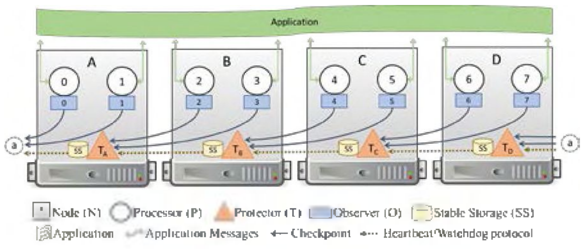

Fig. 1 - RADIC Architecture

RADIC defines four functions *Protection*, *Detection*, *Recovery* and *Error Masking*.

Protection: Observer O_3 is in charge of monitor Process P_3, then O_3 gets a Protector T_A located in a different node N_A. At some point T_A will ask O_3 to get the state of it process and send it to T_A to be stored in T_A´s Stable Storage (SS), this is called a checkpoint; checkpoints will be done periodically to keep on saving the state of the processes during a fault free execution. The moment that every T has its neighbor's checkpoint, the RADIC protection will be in place.

Detection: Node N_B faults when the process P_3 cannot communicate with another process P_2 in the same node, that's when O_3 sends the error message to T_A. Also, a node faults when there is no communication between nodes, that is, each protector (T_A) is in charge of detecting a possible failure in the neighbor Node (N_B). Each protector keeps a heartbeat/watchdog protocol with its neighbors Protectors, in this way a fault detection mechanism is implemented; in one hand, the Protector T_B is periodically sending heartbeats to T_C, and in the other hand, T_B is the watchdog of T_A, so if T_B loses communication with both neighbor protectors, T_B will destroy itself because it has been left alone, if T_A and T_C can't see T_B, then T_A will launch the recovery and error masking functions. Based on this function, RADIC needs at least 3 nodes to work properly.

Recovery: Protector T_A restarts/rolls-back the processes running in N_B, using the data saved in the SS from the last checkpoint, if the system has a spare node, the processes will be restarted in it, otherwise the processes are restarted in the node (N_A) that has the checkpoint.

Error Masking: After the recovery function, when the processes have been restarted, the Observers that got the error message hide the communication error caused from the node failure. The Protector T_A sends a message to the affected Observers (O_{0-1} and O_{4-5}) with the new address where the processes have been recovered, also updates the RadicTable so communications are done as usual.

RADIC has been originally designed to protect a parallel application with message passing, recently it has been leveraged to work in cloud environments. In our work we scale up the architecture to be used for parallel application protection, running in several virtual clusters on multiple cloud environments.

3. H-RADIC Architecture

We propose a new protection level of RADIC, the Hierarchical RADIC (H-RADIC) architecture, an automated and elastic FT framework that protect a parallel application running in virtual clusters on

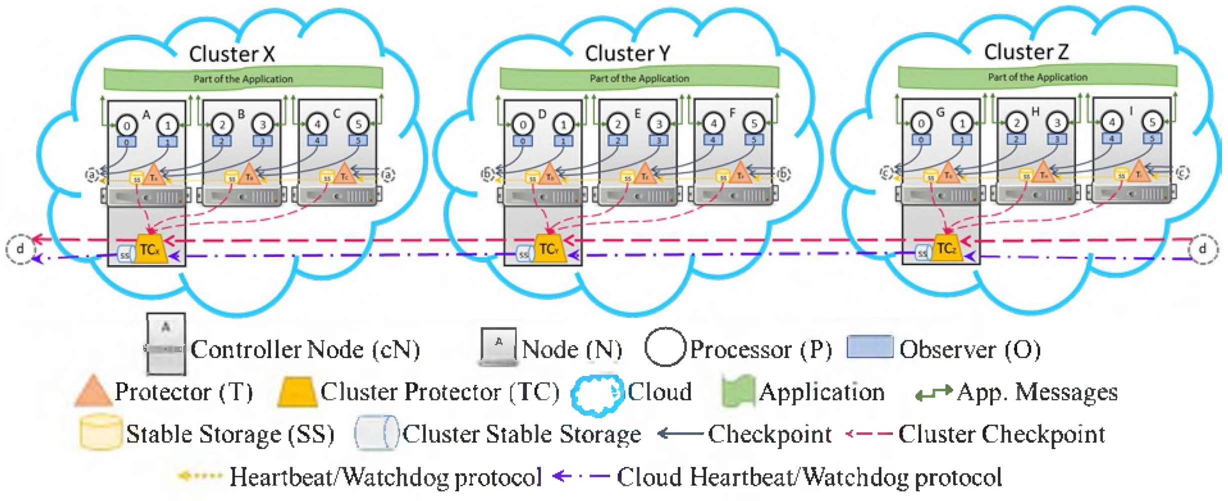

Fig. 2 - H-RADIC Architecture

cloud; requires at least three virtual clusters, each one of them protected with the RADIC architecture and located in different geographical sites so they don't share any of the physical resources, this will allow to identify nodes faults within and between the clusters.

The architecture works for parallel applications with a Message Passing Interface (MPI), that is, besides the regular RADIC architecture designed to protect from node fails, H-RADIC will protect applications from crashing in the event of multiple fails; when the virtual nodes fail, the physical nodes that the virtual are mounted on fail and/or whenever there is loss of communication. H-RADIC will perform a diagnostic and then management of the error.

Additionally to all the RADIC components, H-RADIC has an extra software component, the proTector of the Cluster (TC); on every cluster, a main node will be defined (controller Node - cN) to create the TC´s, which will be in charge of the communications between clusters, as it's shown Fig. 2.

3.1. H-RADIC functions

The H-RADIC architecture implements the RADIC functions in each cluster, that is, whenever there is a fault, the architecture tries to recover from it by applying the RADIC functions. H-RADIC functions will process the failures when the RADIC functions can't process them.

To guaranty the completion of the execution, H-RADIC transported the RADIC functions to be apply into programs running in multi-clusters environments and depict its functionality in the following functions:

Protection: Only while the execution is running fault free, checkpoints will be periodically done. Once that the Protectors (T) have the checkpoint of each of the process, the Cluster Protector (TC_Y) will be in charge of collecting the updates in the Stable Storage (SS) of every T in the Cluster (C_Y). Then, TC_Y must send it to the assigned Cluster Protector TC_X located in a different cloud to store the Cluster Checkpoint (multilevel checkpoint) at the Cluster SS. When every TC has a checkpoint of its neighbor C, the H-RADIC protection will be activated.

Detection: This function works the same way as the RADIC detection function; TC_X is in charge of identifying a possible failure in the neighbor Cluster C_Y, using a heartbeat/watchdog protocol. If C_Y does not answer, TC_X asks TC_Z to verify if there is an error with C_Y, if TC_Z confirms that there is no communication with C_Y, Recovery and Error Masking functions are triggered.

Recovery: Cloud Protector TC_K restarts/rolls-back the process running in C_L, using the data saved in the cluster SS from the last checkpoint, then TC_K checks if there is a spare cluster to lunch the checkpoint in it, otherwise creates new nodes in C_K to restart all the processes previously running in the failed C_L, then updates the RADIC Tables.

Error Masking: After the recovery function, when the processes have been restarted, the Cluster Protector hides the communication errors caused by the cloud failure, the Cluster Protector TC_K sends a message to the affected Protectors with the new address where the processes have been recovered, updating the H-RADIC Table and broadcasting this update to every TC in the architecture, then communications will be done as usual.

3.1.1. H-RADIC Recovery function' options

After a fault is detected, the recovery function has two options to restart a checkpoint; the first one checks if there are spare clusters available in another cloud, viewed in Fig. 3, the second one checks if there are spare nodes in the same cloud, as shown in Fig. 4, then the checkpoint files are sent to the nodes that are in the spare cluster.

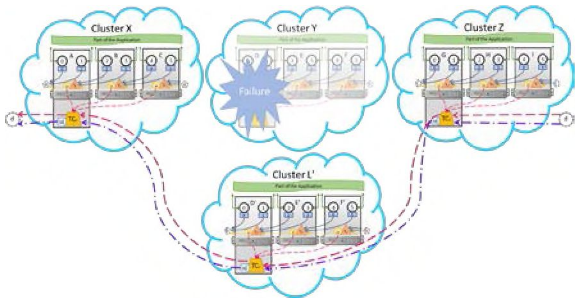

Fig. 3 - H-RADIC Recovery function - spare nodes/cluster in another cloud.

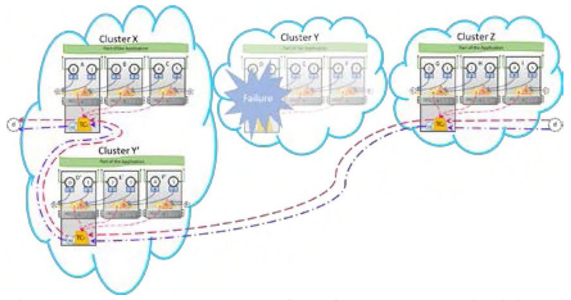

Fig. 4 - H-RADIC Recovery function - spare nodes/cluster in the same cloud.

The main difference between the two restart options is the way to store the checkpoints. In the case that there is a spare cluster available in other cloud (Fig. 3), H-RADIC will be working as usual, but if there are no spare nodes available in a third cloud, the checkpoint will be restarted in the spare nodes of the cloud that has the checkpoint. After the restart, the two clusters (X and Y' in Fig. 4) will send their checkpoints to Cluster Z and vice versa.

When the execution is working like the previous situation and if a new failure rises in one of the clouds, the three clusters will be together in one cloud, as shown in Fig. 5, and henceforth, all the checkpoints will be stored in one cloud storage.

Although this situation will leave the execution exposed to the same vulnerabilities, that lead us to develop this type architecture. Such vulnerabilities like failures in the cloud controller, the storage and the physical computer that the virtual nodes are mounted on. This vulnerability will now allow us to be able to guaranty availability.

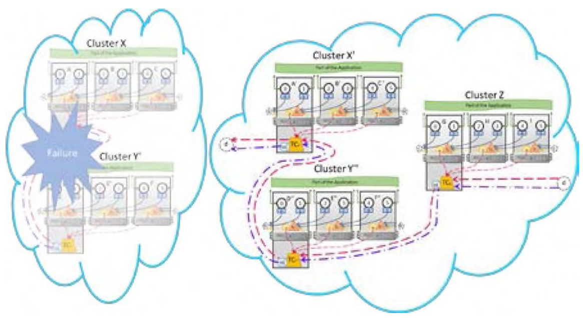

Fig. 5 - H-RADIC Recovery function - spare nodes/cluster in the same cloud and not more clouds left.

4. Experiments

To test the architecture, we mounted 3 clusters running on CentOS. Each one of them has a different NAS Parallel Benchmark [13] program compiled with an implementation of MPI named: MPICH [14]. H-RADIC will performing the checkpoints for this experiments in a coordinated approach, using the open source software package: Distributed Multi-Threaded CheckPointing (DMTCP) [15]. Each cluster has 3 nodes and every node have 8 processors, 24 processors per cluster.

The DMTCP allows to work with the checkpoints at an application layer of the OSI model, making it perfect for virtualized environments. Also, whenever a checkpoint is done, it automatically compresses it using gzip.

4.1. Pseudocode of H-RADIC

Before running the programs, they have to be compiled with the MPI implementation and also the inventory of nodes have to be depicted in the H-RADIC Table (Table 1); it's been established that each cluster will have a Cluster Protector, this will be the previous row in the table, and it´s expressed as (cluster-1), if the program is in cluster Y, the cluster protector will be cluster X. If the cluster it's in the first row (cluster X), then cluster protector will be the las row in the table (cluster Z), to complete a full loop.

Table 1 - H-RADIC TABLE

	Cluster	***Nodes***	***Controller Node***
1	clusterX	A, B, C	A
2	clusterY	D, E, F	E
3	clusterZ	G, H, I	G

All the logic described in the H-RADIC functions can be summarized in Algorithm 1.

Algorithm 1 - H-RADIC pseudocode

```
1: // FUNCTIONS
2: function protection (program, cluster,
   ckPTime){
3:   Send the program to the controllerNode in
     cluster
4:     launch dmtcpCoordinator with the program
       and ckPTime
5:     call detection(cluster, ckPTime)
6:     while
7:       perform checkPoint every ckPTime
         seconds
8:         send checkPoint to (cluster-1)
9:       if program ends
```

```
            broadcast succesfull end to all HRADIC Tables
            break while
    }

function detection (cluster, ckPTime){
    // Goes to the HRADIC Table and takes the next cluster in the table to protect
    establish watchdog protocol between cluster and (cluster+1)
    while
        get heartbeat from (cluster+1)
        send heartbeat to (cluster-1)
        if heartbeat/watchdogProtocol fails
            ask (cluster+2) to establish connection with (cluster+1)
            if the connection is established
                nothing happens
            else if (cluster+2) can't see (cluster+1)
                recovery(cluster+1, ckPTime)
}

function recovery (cluster+1, ckPTime){
    search for spareCluster in other clouds or in the local cloud
    inizialize Nodes in spareCluster = Nodes in (cluster+1)
    send checkPoint files to nodes in spareCluster
    call errorMasking(spareCluster, cluster+1)
    // Call the protection fuction again, but this time run the checkpoint instead of the original program
    call protection(checkPoint,spareCluster,ckPTime)
}

function errorMasking (spareCluster, cluster+1){
    remove (cluster+1) info. in the HRADIC Table
    add spareCluster info. in the HRADIC Table
    broadcast the update to all HRADIC Tables in all the clusters
}

// MAIN
// Launch all the protection functions in parallel
execute protection(bt.C.16, clusterX, 60)
execute protection(lu.C.16, clusterY, 80)
execute protection(cg.C.16, clusterZ, 70)
```

4.2. Results

Calculating the optimal checkpoint interval is a controversial subject; for these experiments, all the programs were executed several times, this was done to identify the average time that each node needed to perform a checkpoint and move it to the different nodes and also to the cluster protector.

The experiment is measuring the time that it takes the application to finish its execution, in the following cases: 1) the application without applying H-RADIC but performing checkpoints, 2) the application with H-RADIC and without errors or failures and, 3) the application with H-RADIC and an induced error. Then by taking the increase percentage of time in 2) and 3), we got Fig. 6.

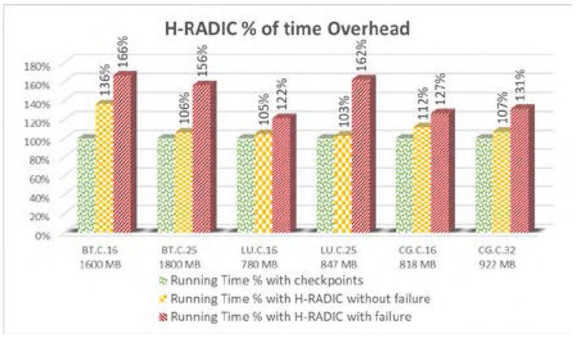

Fig. 6 - H-RADIC percentage of time overhead.

4.2.1. Overhead Breakdown

For the calculation of the overhead, we measured different time variables (as shown in Table 2): the first percentage shown in Fig. 6 is the application running with checkpoints as Eq (1), this time is considered the point of departure from where we are calculating the overhead.

$$T_{ckpt} = p + (ckpt \cdot n) \text{ (Eq. 1)}$$

In order to establish the RADIC and H-RADIC protection, all the checkpoints needed to be copied to the protector nodes and the cluster protector node respectively. This process is done at the background Eq (2) while the application keeps on executing. This process is represented in Fig. 2, by the checkpoint's and cluster checkpoint's lines. Since moving the files take some computational resources, it affects the program execution time, although its not directly in the critical path.

$$bkgrd = chkpSize \cdot protect \text{ (Eq. 2)}$$

The *Running Time percentage with H-RADIC without failure* in Fig. 6 is the time that the application took to execute considering the RADIC and H-RADIC protection in Eq (3).

$$T_{H-RADIC} = T_{ckpt} + bkgrd \text{ (Eq. 3)}$$

Finally, the third experiment, the *Running Time percentage with H-RADIC with failure* in Fig. 6 is the time that took the framework to restart the execution in a spare cluster Eq (4), considering the time to move the checkpoint files and the time to restart the execution.

$$T_{H-RADIC\,\&\,Failure} = T_{H-RADIC} + move + restart$$
(Eq. 4)

Table 2 - Time's variables description.

Variable	**Description**
T_{ckpt}	Execution time with checkpoints
p	Program execution time
ckpt	Time to create a checkpoint
n	Number of checkpoints per execution
$T_{H\text{-}RADIC}$	Execution time with H-RADIC without failures
ckptSize	Checkpoint size in MB
protect	Time to move the ckpt to establish H-RADIC protection.
bkgrd	Unpredictable time impact on p
$T_{H\text{-}RADIC\,\&Failure}$	Execution time with H-RADIC with failure
move	Time to move the checkpoint files to the spare cluster
restart	Time to restart the application

In general, we can appreciate that the overhead caused by implementing H-RADIC and having 0 failures, does not really increase the program execution time, however if a failure rises, we can see the overhead percentage from as low as 22% to as high as 66% in time. This bottleneck is mainly due because of the time taken to move the checkpoint to the nodes in the spare cluster, since the time to restart the application once that the checkpoints where in the spare cluster is negligible.

5. Conclusion

The proposal in this paper is based on the RADIC architecture, that is capable of assure a successful execution of the application even when failures occur. We analyzed every component of the framework and identified the improvement areas.

By translating every RADIC concept to H-RADIC and taking them to a virtual multi-clusters level, we develop a FT framework capable of overcome fatal fails like the loss of a full site. The H-RADIC architecture fully implements RADIC and the new benefits from H-RADIC and it allows us to guarantee the completion of the execution in the event of errors like, loss of nodes, loss of communication between sites or general fails in several sites. It's a solution that can be implemented in virtual and non-virtual environments.

Considering that a traditional FT system, that consist on having one or two copies of a full site in another one, waiting for the main site to fail, with H-RADIC, by distributing the work load in different sites, the need of resources is way less that the ones needed for the traditional FT system.

Finally, the experiments show us that the overhead in most of the programs is reasonable and proves the theory behind the concept.

6. Future Work

Test the architecture implementing semi coordinated and no-coordinated checkpoints, to run a single program in several virtual clusters at the same time.

Develop the architecture in a way that if the performance is decreasing, it can request more resources live to the cloud provider, to assure competition before the due date.

Implement a mechanism that accelerates the transfer rate, by improving I/O, use incremental checkpoints and/or compress [16] the checkpoints even more.

Acknowledgements

The first author acknowledges the support from the National Science and Technology Council of Mexico (Consejo Nacional de Ciencia y Tecnología, CONACYT) that sponsored through a scholarship, and special thanks to all the team at CAOS and the UAB.

References

[1] B. Darrow, "Windows Azure outage hits Europe," 26-Jul-2012. [Online]. Available: https://gigaom.com/2012/07/26/windows-azure-outage-hits-europe/. [Accessed: 30-Mar-2018].

[2] O. Malik, "Severe storms cause Amazon Web Services outage," 29-Jun-2012. [Online]. Available: https://gigaom.com/2012/06/29/some-of-amazon-web-services-are-down-again/. [Accessed: 30-Mar-2018].

[3] "Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region," *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/message/41926/. [Accessed: 31-Mar-2018].

[4] "Google Cloud Status Dashboard." [Online]. Available: https://status.cloud.google.com/incident/storage/17002. [Accessed: 31-Mar-2018].

[5] J. Hult, "Oracle Cloud - unplanned outage - November 7, 2017," *JonathanHult.com*, 17-Nov-2017. .

[6] J. Villamayor, D. Rexachs, and E. Luque, "RaaS: Resiliance as a Service – Fault Tolerance for High Performance Computing in Clouds," p. Accepted, Mar. 2015.

[7] A. Gómez, L. M. Carril, R. Valin, J. C. Mouriño, and C. Cotelo, "Fault-tolerant virtual cluster experiments on federated sites using BonFIRE," *Future Gener. Comput. Syst.*, vol. 34, pp. 17–25, May 2014.

[8] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High Performance Fault Tolerance Interface for Hybrid Systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2011, p. 32:1–32:32.

[9] S. Di, Y. Robert, F. Vivien, and F. Cappello, "Toward an Optimal Online Checkpoint Solution under a Two-Level HPC Checkpoint Model," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 1, pp. 244–259, Jan. 2017.

[10] I. P. Egwutuoha, S. Chen, D. Levy, B. Selic, and R. Calvo, "Cost-oriented proactive fault tolerance approach to high performance computing (HPC) in the cloud," *Int. J. Parallel Emergent Distrib. Syst.*, vol. 29, no. 4, pp. 363–378, Jul. 2014.

[11] L. Fialho, G. Santos, A. Duarte, D. Rexachs, and E. Luque, "Challenges and Issues of the Integration of RADIC into Open MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer, Berlin, Heidelberg, 2009, pp. 73–83.

[12] M. Castro-León, H. Meyer, D. Rexachs, and E. Luque, "Fault tolerance at system level based on RADIC architecture," *J. Parallel Distrib. Comput.*, vol. 86, pp. 98–111, Dec. 2015.

[13] "NAS Parallel Benchmarks," *NASA Advanced Supercomputing Division*. [Online]. Available: https://www.nas.nasa.gov/publications/npb.html . [Accessed: 23-May-2018].

[14] "MPICH | High-Performance Portable MPI," *MPICH*. [Online]. Available: https://www.mpich.org/. [Accessed: 02-Jun-2018].

[15] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–12.

[16] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello, "Improving Performance of Iterative Methods by Lossy Checkponting," *ArXiv180411268 Cs*, Apr. 2018.