REMEM: REmote MEMory as Checkpointing Storage

Hui Jin*, Xian-He Sun*, Yong Chen†, Tao Ke*

*Department of Computer Science
Illinois Institute of Technology
Chicago, Illinois, USA
{hjin6, sun, tke1}@iit.edu

†Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
cheny@ornl.gov

Abstract—**Checkpointing is a widely used mechanism for supporting fault tolerance, but notorious in its high-cost disk access. The idea of memory-based checkpointing has been extensively studied in research but made little success in practice due to its complexity and potential reliability concerns. In this study we present the design and implementation of REMEM, a REmote MEMory checkpointing system to extend the checkpointing storage from disk to remote memory. A unique feature of REMEM is that it can be integrated into existing disk-based checkpointing systems seamlessly. A user can flexibly switch between REMEM and disk as checkpointing storage to balance the efficiency and reliability. The implementation of REMEM on Open MPI is also introduced. The experimental results confirm that REMEM and the proposed adaptive checkpointing storage selection are promising in both performance, reliability and scalability.**

I. Introduction

As a widely used fault-tolerant mechanism, Checkpointing/Restart(C/R) has been well studied and has real potential for practical usage. Several checkpointing systems are available and contribute to mitigate the impact of failures on the application performance [1] [2] [3] [4] [5]. Most of these systems checkpoint the snapshots to the stable storage of the cluster. A stable storage generally means non-volatile disk storage, where the I/O bottleneck, or the so called I/O wall problem, is increasingly becoming an identified performance bottleneck of HPC even without the burden of checkpointing/restart [6] [7]. In addition, checkpointing snapshots are usually large, and the cost of storing/retrieving them to/from disks is high [8]. With large systems composed of hundreds of thousands of nodes, the conventional disk-based checkpointing will overwhelm the slow storage devices [9].

Memory-based checkpointing is a promising solution to break through the bottleneck from the stable storage [8] [10] [9] [11]. However, due to its complexity and potential threat to reliability, memory-based checkpointing is rarely supported by the mainstream of current checkpointing systems.

In this research we present REMEM, a reliable and scalable REmote MEMory checkpointing system, to extend the storage of traditional disk-based checkpointing to remote memory. We first introduce the design of REMEM that can be seamlessly integrated to existing disk-based checkpointing system, with the consideration of both space efficiency and flexibility. Next, we introduce the methodology of implementing REMEM on Open MPI. The design of REMEM supports a flexible switch between traditional disk-based checkpointing and REMEM. This feature makes REMEM an augment of traditional disk-based checkpointing, instead of a complete substitution. In addition, we present a quantitative approach to dynamically select REMEM as checkpointing storage when reliability permits. Finally, extensive experiments are conducted on the implementation of Open MPI + REMEM. The experimental results confirm the benefits of REMEM and the proposed checkpointing storage selection strategy.

The rest of this paper is organized as follows. Section II reviews related works and discusses their relationship with our work. We introduce the design of REMEM in section III. In section IV we present the implementation of REMEM on Open MPI. Section V presents the adaptive checkpointing selection strategy with the consideration of both efficiency and reliability. In section VI, we evaluate the performance of REMEM and compare it with traditional disk-based checkpointing. Section VII concludes this study.

II. Related Work

The idea of memory-based checkpointing (called diskless checkpointing originally) was first proposed by Plank et al. in [12]. There are two main diskless checkpointing schemes: parity-based and neighbor-based checkpointing.

The parity-based approach [8] checkpoints the application's image to each node's spare memory. The parity of all the checkpoints is calculated and saved in extra checkpointing processors to make the parallel application recoverable in the event of failures. The neighbor-based approach [13] [10] removes the coding/decoding calculation of the parity-based approach by introducing one more copy of checkpointing image, i.e. each node checkpoints the application image to both its own memory and the neighbor's.

Though fledged in theoretical study, it is still difficult to adopt diskless checkpointing in practice. Several system level checkpointing systems are already in public as general fault tolerant tools [1] [2] [3] [4] [5] for disk-based checkpointing. However, none of them has direct support of memory-based checkpointing. Scalable Checkpointing/Restart(SCR) library was introduced recently to support diskless checkpointing at application level [14]. As an all-in-one application level solution, it is difficult, if not impossible, for SCR to split the diskless functionality and benefit other existing system level checkpointing systems. Our work in this study bridges the gap between theoretical study and practical implementation of memory-based checkpointing. We present a solution that transparently implements memory as checkpointing storage.

Recent advances in failure analysis and prediction make it possible for failure predictions with satisfactory accuracy and warning lead time [15] [16]. In this research we propose to utilize the failure predictors to adaptively select checkpointing storage. The proposed adaptive checkpointing storage selection is motivated to solve the reliability concerns of REMEM. The adaptive checkpointing storage algorithm automatically selects REMEM as checkpointing storage if the reliability permits, otherwise disk will be selected to minimize the potential concurrent failure cost. The selection is based on the runtime predictor information, instead of static two-level checkpointing as proposed in [17].

Some researchers proposed to utilize the emerging storage materials, such as PCRAM [18] and SSD [19], as checkpointing storage. REMEM can be easily extended to facilitate the implementation of these approaches for a better checkpointing performance.

In a previous work [20], we have presented the preliminary design and experimental result of memory-based checkpointing system. We have also analytically studied the reliability of memory-based checkpointing, which confirmed its feasibility from the perspective of reliability. This work complements [20] with a detailed design, an implementation methodology on real checkpointing environment, and more representative experimental results on parallel environment with representative benchmarks.

III. REMEM Design

A. *Design Goals of REMEM*

The design of REMEM is based on the following principles.

1) Reliability: Reliability is the primary goal of a checkpointing system. REMEM introduces a reliability concern due to the volatile storage of the memory, which should be specially taken care of.

2) Scalability: Memory-based checkpointing is proposed to optimize the storage overhead due to the centralization of I/O server. So the performance of REMEM should be scalable as the number of processes increased.

3) Space Efficiency: Though the available memory is increased steadily, memory is still precious. REMEM targets to use minimal memory with the reliability guaranteed.

4) Transparency: Instead of a fresh new checkpointing system, REMEM should be an improvement and complement to the existing disk-based checkpointing systems. We expect REMEM to support all the disk-based checkpointing system in a seamless and transparent manner: it makes no difference for the user to do disk-based or memory-based checkpointing.

5) Simplicity and Flexibility: The memory-based checkpointing should keep the involvement of the system administrator or the user to minimal. We expect REMEM to work in combination with disk-based checkpointing for a balance between efficiency and reliability, which requires a flexible switch between the two.

B. *Overview of REMEM*

We illustrate the hierarchy view of REMEM in Figure 1. The REMEM system has three primary phases, *node matching*, *system configuration* and *job scheduling*. The node matching phase makes the decision of paring the nodes with each other. The system configuration phase sets up the remote memories as checkpointing repository based on the decisions from node matching phase. Job scheduling phase is responsible for scheduling parallel jobs onto the nodes. The functionality of each phase is discussed as follows.

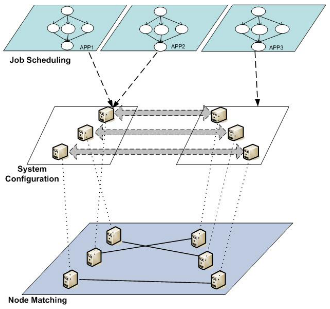


Figure 1: System Overview

- *Node Matching*. In this phase REMEM makes the node matching decision. We term node A as the *source node* and node B as the *backup node* if node B is selected to share its spare memory for node A to checkpoint. In such case, we call node A and B as *paired* nodes. The aim of the node matching is to decide the source-backup node pairs. A desired node matching strategy should minimize the possibility of concurrent failures for the applications. The input of the node matching phase is the physical information and/or the failure log of each node. The output of the node matching phase is the node matching decision, which is organized into a file. This phase addresses the reliability.
  Note that the backup node is still available to run applications and keep the utilization of the system. In [20], we have evaluated the size of memory that can

be shared by the backup node while keeping the host applications intact.

- *System Configuration*. Based on the node matching decision from node matching phase, REMEM configures the available memory as checkpointing storage. System configuration phase is responsible to execute the node matching decision. System configuration is a key component of the system since its implementation decides the transparency, simplicity and flexibility of the design goal. We propose to use virtual memory file system and Network File System (NFS) protocol to implement the system configuration, which is detailed in sub-section III-D.
- *Job Scheduling*. REMEM takes a job scheduling strategy that assigns paired nodes to different parallel jobs. The first motivation is to reduce the possibility of concurrent failures. The processes of a parallel application are always correlated due to the communication and coordination among them. The processes that belong to one parallel job are more likely to fail concurrently. We can limit the impact of a failed process to only one side of paired nodes for a parallel application by avoiding both sides of paired nodes to execute one parallel job. Furthermore, without this rule, a node may work as both source node and backup node for one parallel application. If failures happen to such nodes, the application will not be recoverable for coordinated checkpointing.

C. Node Matching

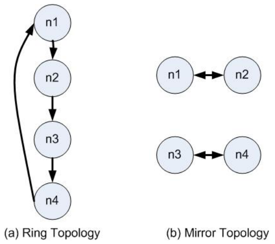


Figure 2: Different Matching Topologies

In general there are two types of paring strategy corresponding to two different node topologies. Ring topology will be formed if every node has different source and backup nodes, as plotted in Figure 2a. Figure 2b reflects the mirror topology corresponding with mutually paired matching. In [11], the authors examines the reliability of these two topologies and confirms that mutually paired node matching is more reliable.

We can get higher reliability by pairing nodes that belong to different components, e.g., power supply, network switch, etc. If both the paired nodes are from one component, the failure of the common component will corrupt both processes and we will not be able to recover in such case. For example, if both the source and backup nodes are managed by one power supply, the failure of the power supply will fail both nodes and the system will not be able to restart.

D. System Configuration

We propose to use virtual memory file system and NFS protocol to implement remote memory checkpointing:

1) Backup node creates a local directory and mounts it from the virtual memory;
2) The local directory of the virtual memory is exported as an NFS server to the local network;
3) The source node mounts its local directory from the remote memory that is exported by the backup node.

We have chosen *tmpfs* file system for the actual implementation. Tmpfs [21] is a file system that uses resources and structures of the virtual memory subsystem. Rather than using dedicated physical memory such as “RAM disk”, tmpfs uses the operating system page cache for file data. Tmpfs provides increased performance for file reads and writes and has no adverse effects on the overall system performance. The use of virtual memory will also handle potential system crash if the memory is occupied by the running application and the checkpointing image exceeds the available physical memory.

By treating remote memory as normal file directory, REMEM can be easily adopted to the existing disk-based checkpointing system with minimal modification. Such design achieves the transparency of the design goal. The checkpointing operations of our approach taken by users have no difference with traditional disk-based approaches. This implementation is simple. To work as a backup node, a node simply adds the memory to the NFS export configuration file and restart the NFS server service. To work as a source node, a node gets the backup node from the node matching decision file, and then issues a mount command to mount the remote memory from the backup node. The implementation does not need modification to the checkpointing system, OS or the application. The checkpointing storage can be easily switched between remote memory or normal disk by simply umounting the memory NFS entry or writing to another directory on permanent storage.

E. Failure Handling

The failure handling of a compute node (source node) is not abnormal: loads the most recent image from remote memory as an ordinary file and the application will be resumed. If there is no failure on the backup nodes when recovering, we will simply load the images from REMEM to restart the application. However, if both the source and backup nodes fail when restart is in progress, the previous disk-based checkpointing image is restored instead of the current memory-based checkpointing image.

IV. Implementation of REMEM on Open MPI

A. Open MPI Overview

Open MPI is an open source MPI-2 implementation that provides a high performance, robust, parallel execution environment for a wide variety of computing environments [22]. Open MPI supports a transparent, coordinated checkpoint/restart implementation supported primarily by the BLCR library [2] [5] [23].

Open MPI introduces the concept of *snapshot reference* to abstract the users away from tracking multiple different files for a single distributed checkpoint interval [23]. A snapshot reference is a single named reference that corresponds to one checkpoint of either one single process or parallel job. Local snapshot reference is referred as the directory containing the snapshot of a single process and the corresponding metadata file. Global snapshot reference contains all the local snapshots and a global metadata file that describes the aggregated local snapshot references, rank information and the checkpointing interval.

Figure 3 demonstrates the organization of checkpointing metadata files and the snapshots. *Base global snapshot directory* (base directory) that sits on the root of Figure 3 is the default location that stores all the checkpointing files for all the processes. The naming schemes of directories/files are indicated in the brackets.

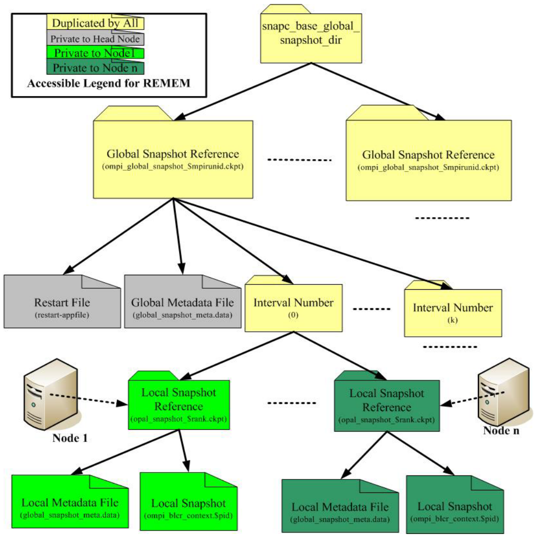

Duplicated by All
Private to Head Node
Private to Node1
Private to Node n
Accessible Legend for REMEM
snapc_base_global_snapshot_dir
Global Snapshot Reference (ompi_global_snapshot_Smpirunid.ckpt)
Global Snapshot Reference (ompi_global_snapshot_Smpirunid.ckpt)
Restart File (restart-appfile)
Global Metadata File (global_snapshot_meta.data)
Interval Number (0)
Interval Number (k)
Node 1
Local Snapshot Reference (opal_snapshot_$rank.ckpt)
Local Snapshot Reference (opal_snapshot_$rank.ckpt)
Node n
Local Metadata File (global_snapshot_meta.data)
Local Snapshot (ompi_blcr_context.$pid)
Local Metadata File (global_snapshot_meta.data)
Local Snapshot (ompi_blcr_context.$pid)

Figure 3: Snapshot Organization in Open MPI

B. Checkpointing/Restart on Open MPI

1) Checkpointing: Coordinated checkpointing of Open MPI on a NFS-based system is implemented with the following steps.

a The head node creates the directory for the global snapshot reference under the base directory and broadcasts the checkpointing requests to all the nodes that have a process for the application.

b A compute node receives the checkpointing request, interacts with others to achieve the global consistent state.

c Each node creates a directory for its local snapshot reference and dumps its process image to the designed path.

d When notified by a compute node that one local checkpointing is completed, the head node updates its global metadata file.

e Checkpointing is terminated when all the compute nodes finish their local checkpointing and notify the head node.

2) Restart:

a The head node generates restart file based on the metadata file of the global snap shot reference. Each entry of the restart file corresponds to one process and specifies the mpirun options, the local snapshot.

b The head node issues an mpirun request with the –app option with the value of the restart file, the host file is specified by the user.

c The restart completes when all the processes have been restarted on its allocated node.

C. Checkpointing/Restart on Open MPI + REMEM

REMEM first creates a base directory for each node, which is not shared. This means that each node has a base directory of the same name, but their physical storages are different. For the head node, this directory is simply used as a normal file that is stored on the local disk. For the compute nodes, the base directory is mounted as the NFS client of the remote memory from their backup node.

To execute a checkpoint for Open MPI + REMEM, the head node broadcasts a request to the backup nodes to clean the existing local snapshots on their memory and save spaces for the ongoing checkpointing. This procedure is fast since each backup node can clean their memory concurrently. Also, the cleanup is not part of the checkpointing overhead since it is executed on the backup nodes and does not interrupt the application running on the source nodes.

We illustrate the accessibility of different components with different colored legends in Figure 3. The global metadata file and the future restart file are privately owned by the head node. Each compute node has private access to its own local snapshot reference. All the nodes have duplicated copies of the base directory, global snapshot directory and the interval number folders, but physically these directories are stored at different locations.

The only prerequisite to restart an application for Open MPI + REMEM is to make all the snapshot reference data, including both metadata files and the local snapshots of each process, accessible to the head node. We take a straightforward way to implement this: REMEM creates a directory for the local snapshot reference of each process on the head node, and mounts them from the physical storage that stores the actual local snapshots.

The extra cost for the restart of Open MPI + REMEM is the overhead of mounting multiple local snapshot references on the head node, which is usually a transient process and can be parallelized to mitigate the delay from individual mount operation. To avoid excessive number of NFS clients on the head node, REMEM umounts them after the restart is completed. This cost should not be considered as part of the restart since it can be executed after the mpi job has been launched on the compute nodes.

V. Adaptive Checkpointing Storage Selection

Checkpointing is taken periodically during the execution of a parallel application with fixed interval τ. There are two options for checkpointing storage when issuing the checkpointing: disk and REMEM. C_{disk}, R_{disk} and C_{remem}, R_{remem} denote the checkpointing and restart overhead for disk and REMEM, respectively. We define P_{source} as the possibility of one or more source node failure for the next checkpointing segment of τ. Similarly, P_{backup} reflects the possibility of failure for the backup nodes. We have to rollback the application to the last available checkpointing when the failure is recovered, which causes a *rework* cost of δ.

O_{disk} and O_{remem} are used to quantify the *overall overhead* for disk-based checkpointing and REMEM, respectively. Our selection of REMEM or disk is biased towards the one with less overall overhead. Three factors contribute to the overall overhead: Checkpointing overhead, restart cost and rework. REMEM outperforms disk-based checkpointing with lower checkpointing and restart overhead, but may penalize the performance with more rework cost due to the volatile memory. Next we present an analytical study for the overall overhead.

If disk is selected as the checkpointing storage, the overall overhead is:

$$\begin{aligned} O_{disk} &= C_{disk} + P_{source} \times (\delta + R_{disk}) \\ &= C_{disk} + P_{source} \times (\tau/2 + R_{disk}) \end{aligned} \quad (1)$$

Checkpointing overhead will contribute constantly to the overall overhead. The second part of formula 1 reveals that the rework and restart costs are only counted when the failure does happen to the source nodes. Note that from [24], the expected rework cost can be approximated by $\tau/2$, with the fact that failure is expected to occur half way through a checkpointing segment.

The scenario for REMEM is relatively more complex since the reliability of backup nodes is another factor to decide the overall overhead. We illustrate the expected overhead as:

$$\begin{aligned} O_{remem} =& C_{remem} + P_{source}(1 - P_{backup}) \times (\tau/2 + R_{remem}) \\ &+ P_{source}P_{backup} \times (\tau/2 + n\tau + R_{disk}) \end{aligned} \quad (2)$$

Similarly, checkpointing overhead C_{remem} is an inevitable contributor. If source nodes fail during the checkpointing segment of τ but backup nodes are healthy, we simply restart from the snapshots stored in the remote memory of backup nodes, which constitutes the second part of formula 2. Under the condition of both source and backup nodes failure, we have to rollback to the last disk-based checkpointing that holds the permanent snapshots for recovery. This will increase the rework cost by $n\tau$, where n denotes the number of segments between last disk-based checkpointing and the current one.

We demonstrate the checkpointing storage selection algorithm in Algorithm 1. The rationale behind the algorithm is to select the checkpointing storage with less expected overhead.

Algorithm 1 Checkpointing Storage Selection Algorithm

```
$n = 0$.
for each checkpointing do
    if there is a failure alarm for the source nodes then
        $P_{source} = precision$
    else
        $P_{source} = 1 - recall$
    end if
    if there is a failure alarm for the backup nodes then
        $P_{backup} = precision$
    else
        $P_{backup} = 1 - recall$
    end if
    Calculate $O_{disk}$ and $O_{remem}$ from formulas 1 and 2.
    if $O_{remem} < O_{disk}$ then
        Select REMEM as checkpointing storage.
        $n = n + 1$.
    else
        Select Disk as checkpointing storage.
        $n = 0$.
    end if
end for
```

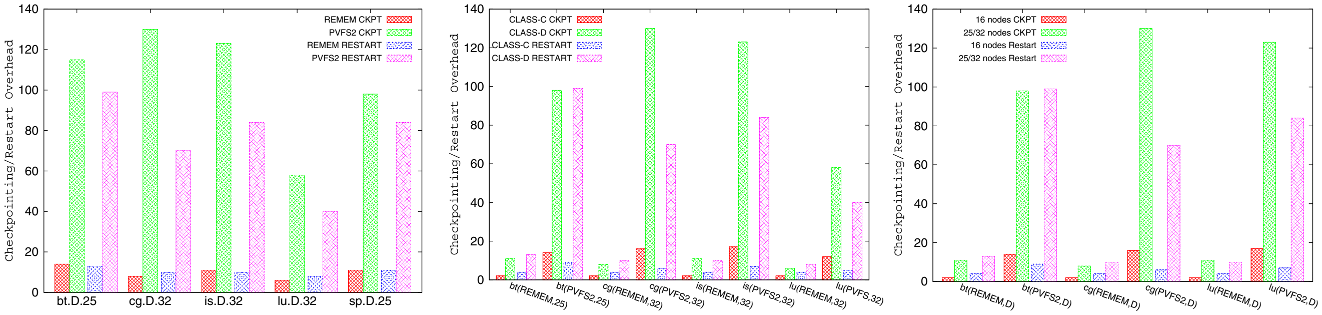
We take the advantages of recent advances in failure analysis and prediction such as [15] [16]and assume the existence of failure predictors with acceptable accuracy and lead time.

When making the checkpointing storage decision, we first check with the failure predictor and decide the corresponding P_{source} and P_{backup}. There are two metrics to evaluate the predictor accuracy: $precision$ and $recall$. While precision reflects the proportion of correct predictions to all the predictions made, recall is defined as the proportion of correct predictions to the number of failures. When there is a failure alarm, the failure probability is $precision$; otherwise, $1 - recall$ is used to reflect the failure probability.

VI. Experimental Study

A. Experimental Setup

Our experiments were conducted on a 65-node Sun Fire Linux-based cluster. The cluster is composed of one Sun Fire X4240 head node, with dual 2.7 GHz Opteron quad-core processors and 8GB memory, and 64 Sun Fire X2200 compute nodes with dual 2.3GHz Opteron quad-core processors and 8GB memory. Each compute node has a 250GB



(a) Class D

(b) Problem Size Scaling

(c) Task Scaling

Figure 4: Checkpointing/Restart Overhead on NPB Benchmarks

7.2K-RPM SATA hard drive. The nodes are connected fully with Gigabits Ethernet.

The software environment includes Ubuntu enterprise server with Linux kernel 2.6.10, Open MPI v1.3.3 and GCC 4.3.3. REMEM was implemented on the Open MPI with the support of tmpfs and NFS 3.0.

The 64 compute nodes are organized in two groups naturally by their rack id. The nodes from the two groups are mutually mapped. So we have two groups, each of which is composed of 32 compute nodes. With REMEM, the application running on one group will be checkpointed to the backup nodes of another group.

We have 4 dedicated X2200 computer nodes configured as PVFS2 servers [25]. Each node works as both metadata server and the I/O server. All compute nodes were used as client nodes. The strip size of PVFS2 server is configured as 64KB.

Results were obtained for the NAS Parallel benchmarks (NPB) version 3.3 [26]. The experiments have been conducted on all the compilable benchmarks with class D. The maximum number of processes of each benchmark is up to 32. We also evaluate the performance for some benchmark of class C to observe the overhead with the problem size scaled.

B. REMEM Performance

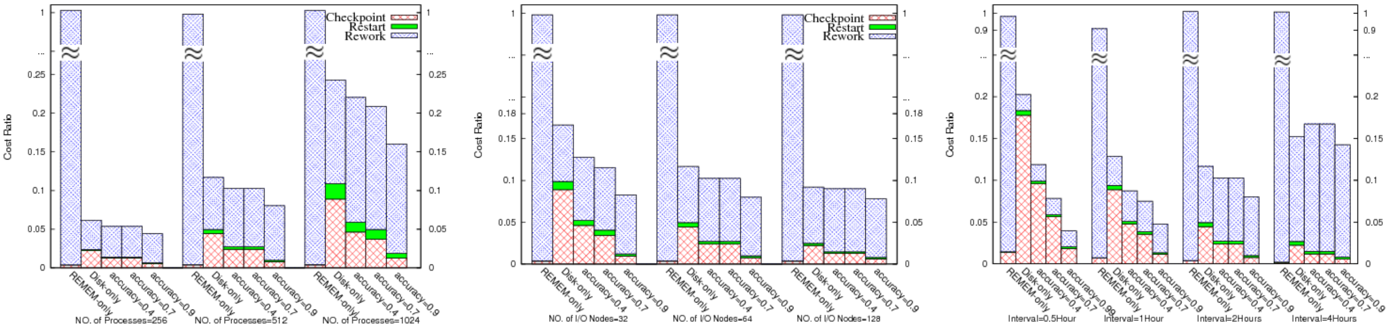
1) Overhead for NPB Benchmark Class D: The experimental results with different benchmarks of class D is presented in Figure 4a, in which REMEM presents significantly advantage over PVFS2 based checkpointing. In the ideal case of benchmark cg, class D with process number of 32, the REMEM checkpointing overhead is just 8 seconds, compared with that of 130 seconds for PVFS2. Similar observations can be found for the restart overhead. An interesting observation is that, different from PVFS2 based checkpointing, the restart cost of REMEM is close to the checkpointing cost. For some benchmarks such as bt.D.25 and cg.D.32, the restart overheads are even 2 seconds more than those of checkpointing. The essence of both checkpointing and restarting is to transfer image from the memory of one node to another, so the overhead of the two is supposed to be close, if not equal.

2) Problem Scaling Performance: In Figure 4b we scale the problem size of some benchmarks from class C to D and compare the differences. With the problem size increased, REMEM exhibits more scalable performance. Taken benchmark bt of 32 processes for example, the image size of class D is 14 times larger than that of class C. We find that the checkpointing bandwidth for PVFS2 is increased by twice from class C to D. The bandwidth of restart on PVFS2 for class D is 1.5 times bigger than that of class C. REMEM increases the bandwidth of both checkpointing and restart by a factor of 5 when the problem size is changed from C to D. The growth of bandwidth with increased problem size is due to that the bandwidth is not completely saturated for class C, which leaves improvement spaces for a larger image size of class D.

3) Task Scaling Performance: Figure 4c depicts the changes of overhead with different number of processes for some benchmarks of class D. REMEM presents excellent scalability performance. As the number of processes increased, we observe that both the checkpointing and restart overhead are reduced. The size of local snapshot on each node will be reduced with more number of processes, which will speed up the checkpointing/restart speed of REMEM since the data transfer is completely distributed with REMEM. PVFS2 based checkpointing shows poor scalability since that the number of fixed number of I/O servers will limit the performance for more number of processes due to resource contention.

C. Evaluation of Adaptive Checkpointing Storage Selection

We simulate a cluster with 2048 compute nodes. For each node, we generate a series of failure arrivals with *Weibull* distribution. The MTBF of each node is set as 7668 Hours and the shape parameter is 0.7. The assumption of failure distributions and the values of the parameters are based on the observations from [27]. We obtain the



(a) Performance with Different Number of Processes (b) Performance with Different No. of I/O Nodes (c) Performance with Different Checkpointing Interval

Figure 5: Performance Evaluation of Adaptive Checkpointing Storage Selection

REMEM and PVFS2 bandwidth based on the observation from sub-sectionVI-B and use them to model checkpointing and restart costs under different scenarios. The default values for the number of processes, I/O nodes and checkpointing interval are set as 512, 64 and 2 hours, respectively.

We take continuous 10000 checkpointing segments. The adaptive checkpointing storage selection of each segment is based on Algorithm 1.

The overall elapsed time of the 10000 segments includes the useful work progress, rework cost, checkpointing and restart overhead. The sum of work progress and rework cost is fixed regardless of the checkpointing storage selection, which is equivalent to $length = 10000 \times \tau$. We divide rework, checkpointing and restart cost by $length$ and use the normalized cost ratios as the evaluation metric. In general, a lower cost ratio reflects a better checkpointing storage strategy.

Figure 5 illustrates the performance of adaptive checkpointing under different scenarios. One group of bars represents the performance with one configuration. The last three bars of each group reflect performance with different predictor accuracies. Here we use $F-measure = 2\frac{precision \times recall}{precision+recall}$ to curve the accuracy of failure predictor. For the sake of comparison, we also measure the performance if REMEM and disk are constantly selected as checkpointing storage and plot the cost ratios in the first two bars.

In 5a we range the number of processes from 256, 512 to 1024. Our first observation is that though featured with significantly low checkpointing and restart costs, it is still not acceptable in practice if we only use REMEM as checkpointing storage. If REMEM is always selected as checkpointing storage, we cannot afford the rework cost (nearly 100%). The advantage of adaptive checkpointing storage selection over disk-only approach goes up as the number of processes is increased. This observation reveals the potential of REMEM and it adaptive checkpointing storage selection for large-scale computing environment.

To observe the performance with different disk-based checkpointing overhead, we vary the number of I/O nodes of PVFS2 from 32, 64 to 128 in Figure 5b. REMEM-only is still not preferable due to its unacceptable rework cost. While observing a steady improvement of checkpointing and restart cost ratio for the second bar of disk-only checkpointing storage, its rework cost is constantly kept as high as 9.3%. The last three bars of adaptive checkpointing storage selection outperforms the second bar of disk-only checkpointing with different degrees. A detailed study of the source data reveals that the advantage of adaptive approach over disk-only is mainly originated from the improvement to checkpointing and restart cost. Disk-only checkpointing always stores checkpointing images to permanent storage thus minimizes the rework cost. The proposed adaptive checkpointing saves the cost by lowering checkpointing and restart cost while maintaining the rework cost close to that of disk-only checkpointing.

Figure 5c presents cost ratios with checkpointing intervals varied from 0.5, 1, 2 to 4 hours. The absolute value of REMEM-only checkpointing overhead is fixed for different intervals. However, the increased rework cost due to different $length$ values makes checkpointing cost ratio decreased for longer checkpointing intervals. Similar observations can be found for the checkpointing overhead of disk-only checkpointing. There is an optimal checkpointing interval for disk-only checkpointing: when the checkpointing intervals grows from 0.5 to 4, the total cost ratio is reduced at first, and grows up later on. Longer checkpointing interval does decrease checkpointing overhead ratio. However, the rework cost will be correspondingly increased and the improvement from checkpoint cost will be reduced.

VII. Conclusions and Future Work

Fault tolerance has become a critical issue of large-scale HPC and the emerging Cloud computing. In the meantime, the slow disk performance and the large number of processors of current HPC technology prevent the conventional disk-based checkpointing strategy working effectively on modern HPC systems. The emerging Cloud computing system built upon large-scale clusters suffers the same issue as well. Under such observation, in this paper, we revisit the idea of memory-based checkpointing and present REMEM to extend the checkpointing storage from disk to remote memory. REMEM is designed to be transparently supported by existing disk-based checkpointing systems. The design of REMEM needs little modification to the existing checkpointing system.

REMEM is not a complete substitution for existing disk-based checkpointing due to the volatile memory and potential excessive memory usage. It should be used in combination with traditional disk-based checkpointing towards a reliable and effective solution. We have made two efforts to facilitate the hybrid checkpointing. First, the alternation of memory-based and disk-based checkpointing is designed to be simple and flexible. Furthermore, an adaptive checkpointing storage selection strategy is proposed to balance the reliability and efficiency.

The experimental results have confirmed that REMEM outperforms PVFS2-based checkpointing in both performance and scalability. We also highlight the potential of adaptive checkpointing storage selection towards a reliable and efficient fault tolerant technique for large-scale systems.

In the future, we are interested in extending the idea of REMEM to other emerging storage alternatives such as PCRAM [18] and SSD [19]. The long term goal of our research is to build a reliable, efficient and scalable checkpointing system that utilizes the heterogeneous storage medias of the system in a unified and transparent way.

ACKNOWLEDGMENT

This research was supported in part by National Science Foundation under NSF grant CCF-0937877, CNS-0834514, CNS-0751200, CCF-0702737, and by Department of Energy SciDAC-2 program under the contract No. DE-FC02-06ER41442.

REFERENCES

[1] J. Plank, M. Beck, G. Kingsley and K. Li, ”Libckpt: Transparent under Unix”, *USENIX Winter 1995 Technical Conference*, 1995.

[2] Berkerley Lab Checkpoint/Restart (BLCR). *https://ftg.lbl.gov/CheckpointRestart/.*

[3] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, etc, MPICH-V Project: A Multiprotocol Automatic Fault Tolerant MPI, *Proc. of International Journal of High Performance Comuting Applications*, 2006.

[4] S. Sankaran, J. Squeyres, B. Barrett, A. Lumsdaine, etc, The LAM/MPI Checkpoint/Restart Framework: System-Initialized Checkpointing, *Proc. of International Journal of High Performance Computing Applications*, 2005.

[5] J. Hursey, T. Mattox and A. Lumsdaine, Interconnect Agnostic Checkpoint/Restart in Open MPI. *Proc. of 18th ACM International Symposium on High Performance Distributed Computing (HPDC'09)*, 2009.

[6] Y. Chen, S. Byna, X.-H Sun, R. Thakur, and W. Gropp, Hiding I/O Latency with Pre-execution Prefetching for Parallel Applications, *Proc. of ACM/IEEE SuperComputing Conference 2008 (SC'08)*, 2008.

[7] Y. Chen, X.-H Sun, R. Thakur, H. Song and H. Jin, Improving Parallel I/O Performance with Data Layout Awareness, *Proc. of the IEEE International Conference on Cluster Computing 2010 (Cluster'10)*, 2010.

[8] J. Plank, K. Li and M. Puening,Diskless Checkpointing, *IEEE Trans. Parallel and Distributed Systems*, Vol. 9(10), 1998.

[9] C.-D. Lu, Scalable Diskless Checkpointing for Large Parallel Systems, *Ph.D dissertation, Department of Computer Science. University of Illinois at Urbana-Champaign*, 2005.

[10] L. M. Silva and J.G. Silva, An Experimental Study About Diskless Checkpointing, *Proc of 24th Euromicro Conference*, pp. 395-402, 1998.

[11] Z. Chen, G. Fagg, E. Gabriel, etc, Fault Tolerant High Performacne Computing by a Coding Approach, *Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2005 (PPoPP'05)*, 2005.

[12] J. Plank K. Li, ”Faster Checkpointing with $N + 1$ Parity”, *Proc. of 24th International Symposium . Fault-Tolerant Computing*, pp. 288-297, Austin,June 1994.

[13] T. Chiueh, P. Deng, Evaluation of Checkpointing Mechanism for Massively Parallel Machnes, *Proc. of 26th Fault-Tolerent Computing Symposium, (FTCS-26)*, pp. 370-379, 1996.

[14] A. Moody, G. Bronevetsky, K. Monhror and B. R. Supinski, Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System, *Proc. of ACM/IEEE SuperComputing Conference 2010 (SC'10)*, 2010.

[15] J. Gu, Z. Zheng, Z. Lan, etc, Dynamic Meta-Learning for Failure Prediction in Large-scale Systems: A Case Study, *Proc. 37th International Conference on Parallel Processing (ICPP'08)*, Sept, 2008.

[16] R. Gu, S. Papadimitriou, P. S. Yu, and S.-P. Chang, Toward Predictive Failure Management for Distributed Stream Processing Systems, *Proc of IEEE ICDCS 2008*.

[17] N.H. Vaidya, A Case for Two-Level Recovery Schemes, *Proc of ACM SIGMETRICS Conf. Measurement and Moedeling of Computer Systems*, 1995.

[18] X. Dong, N. Muralimanohar, M. Jouppi, etc, Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Furture Exacale Systems, *Proc of SuperComputing'09*.

[19] L.B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka, Distributed Diskless Checkpoint for Large Scale Systems, *Proc of 10th International Symposium on Cluster, Cloud and Grid Computing (CCGrid'10)*

[20] H. Jin, X.-H. Sun, B. Xie and Y. Chen, An Implementation and Evaluation of Memory-based Checkpointing. *Poster of ACM/IEEE SuperComputing Conference 2009 (SC'09)*, Nov, 2009. *http://www.cs.iit.edu/˜scs/research/posters.html*

[21] Peter Snyder, Tmpfs: A Virtual Memory File System, *Proc. of The Autuman 1990 European Unix User Group Conference*, Nice, France, 1990.

[22] E. Garbriel, G.E.Fagg, G. Bosilica,etc, Open MPI: Goals, Concept, and Design of a Next Gneration MPI Implementation, *Proc. of 11th European PVM/MPI Users' Group Meeting*, 2004.

[23] J. Hursey, J. M. Squyres, T. I. Mattox and A. Lumsdaine, The Disign and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. *Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS'07)*,2007.

[24] J.T. Daly, A Higher Order Estimate of The Optimum Checkpoint Interval for Restart Dumps, *Furture Generation Computer Systems*, Vol. 22 pp.301-312, 2006.

[25] Parallel virtual File System (PVFS), *http://www.pvfs.org*

[26] F. Wong, R. Martin, R. Arpaci-Dusseau, and D. Culler, Architectural Requirements and Scalability of The NAS Parallell Bencharmks, *Proc. of ACM/IEEE SuperComputing Conference 1999 (SC'99)*, 1999.

[27] B. Schroeder, G. A. Gibson, A Large-Scale Study of Failures in High-Performance Computing Systems, *Proc. of Internetional Conference on Dependable Systems and Networks (DSN'06)*, 2006.