Rejuvenating Shadows: Fault Tolerance with Forward Recovery

Xiaolong Cui, Taieb Znati, Rami Melhem
Computer Science Department
University of Pittsburgh
Pittsburgh, USA
Email: {mclarencui, znati, melhem}@cs.pitt.edu

***Abstract*—In today's large-scale High Performance Computing (HPC) systems, an increasing portion of the computing capacity is wasted due to failures and recoveries. It is expected that exascale machines will decrease the mean time between failures to a few hours. This makes fault tolerance a major challenge for the HPC community. Moreover, the stringent power cap set by the US Department of Energy for exascale computing further complicates this challenge. This work explores novel methodologies to fault tolerance that achieves forward recovery, power-awareness, and scalability. The proposed model, referred to as Rejuvenating Shadows, has been implemented in MPI, and empirically evaluated with multiple benchmark applications that represent a wide range of HPC workloads. The results demonstrate that Rejuvenating Shadows has negligible runtime overhead during failure-free execution, and can achieve significant advantage over checkpointing/restart when failures are prone.**

***Keywords*-Rejuvenation; Leaping; Extreme-scale computing; Forward recovery; Reliability;**

I. Introduction

The path to extreme scale computing involves several major road blocks and numerous challenges inherent to the complexity and scale of these systems. A key challenge stems from the stringent requirement, set by the US Department of Energy, to operate in a power envelope of 20 Megawatts. Another challenge stems from the huge number of components, order of magnitudes higher than in existing HPC systems, which will lead to frequent system failures, significantly limiting computational progress [1]. This puts into question the viability of traditional fault-tolerance methods and calls for a reconsideration of the fault-tolerance and power-awareness problem, at scale.

A common approach to resilience relies on checkpointing and rollback recovery. Specifically, the execution state is periodically saved to a stable storage to allow recovery from a failure by restarting from a checkpoint either on a spare or on the failed processor after rebooting it. As the rate of failure increases, however, the time to periodically checkpoint and restart approaches the system's Mean Time Between Failures (MTBF), leading to a significant drop in efficiency and increase in power [2], [3].

A second approach to resilience is replication, which exploits hardware redundancy by executing simultaneously multiple instances of the same task on separate processors [4]. The physical isolation of processors ensures that faults occur independently, thereby enhancing tolerance to failure. This approach, however, suffers from low efficiency, as it dedicates 50% of the computing infrastructure to the execution of replicas. Furthermore, achieving exascale performance, while operating within the 20 MW power cap, becomes challenging and may lead to high energy costs.

To address these shortcomings, the *Shadow Replication* computational model, which explores radically different fault tolerance methodologies, has been proposed to address resilience in extreme-scale, failure-prone computing environments [5]. Its basic tenet is to associate with each main process a suite of coordinated shadow processes, physically isolated from their associated main process to hide failures, ensure system level consistency and meet the performance requirements of the underlying application. For elastic applications whose performance requirements include multiple attributes, a single shadow that runs as a replica of its associated main, but at a lower execution rate, would be sufficient to achieve acceptable response time. This Lazy Shadowing scheme, which is described in [6], strikes a balance between energy consumption and time-to-completion in error-prone exascale computing infrastructure. Experimental results demonstrate its ability to achieve higher performance and significant energy savings in comparison to existing approaches in most cases.

Despite its viability to tolerate failure in a wide range of exascale systems, Lazy Shadowing assumes that either the main or the shadow fails, but not both. Consequently, the resiliency of the system decreases as failure increases. Furthermore, when failure occurs, shadows are designed to substitute for their associated mains. The tight coupling and ensuing fate sharing between a main and its shadow increase the implementation complexity of Lazy Shadowing and reduce the efficiency of the system to deal with failures.

In this paper, we introduce the new concept of *Rejuvenating Shadows* to reduce Lazy Shadowing's vulnerability to multiple failures and enhance its performance. In this new model, shadows are no longer replicas, which are promoted to substitutes for their associated main processes upon a failure. Instead, each shadow is a *rescuer* whose role is to restore the associated main to its exact state before failure.

Rejuvenation ensures that the vulnerability of the system

to failure does not increase after a failure occurs. Furthermore, it can handle different types of failure, including transient and crash failures. The main contributions of this paper are as follows:

- Proposal of Rejuvenating Shadows as an enhanced scheme of Lazy Shadowing that incorporates rejuvenation techniques for consistent reliability.
- A full-feature implementation of Rejuvenating Shadows for Message Passing Interface.
- An implementation of application-level in-memory Checkpointing/Restart for comparison.
- A thorough evaluation of the overhead and performance of the implementation with multiple benchmark applications and under various failures.

The rest of the paper is organized as follows. We begin with a survey on related work in Section II. Section III introduces fault model and system design, followed by discussion on implementation details in Section IV. Section V presents empirical evaluation results. Section VI concludes this work and points out future directions.

II. RELATED WORK

The study of fault tolerance has been fruitful for large-scale High Performance Computing [7]. Checkpointing/restart periodically saves the execution state to stable storage, with the anticipation that computation can be restarted from a saved checkpoint [8]. Message logging protocols allow a system to recover beyond the most recent consistent checkpoint by combining checkpointing with logging of non-deterministic events [9]. Proactive fault tolerance relies on a prediction model to forecast faults, so that preventive measures, such as task migration or checkpointing, can be taken [10]. Algorithm-based fault tolerance (ABFT) uses redundant information inherent of its coding of the problem to achieve resilience.

For the past 30 years, disk-based coordinated checkpointing has been the primary fault tolerance mechanism in production HPC systems [11]. Coordinated Checkpointing gains its popularity from its simplicity and ease of implementation. Its major drawback, however, is the lack of scalability [12]. Uncoordinated checkpointing allows processes to record their states independently, thereby reducing the overhead during fault free operation [13]. However, the scheme requires that each process maintain multiple checkpoints, necessary to construct a consistent state during recovery, and complicates the garbage collection scheme.

One of the largest overheads in any checkpointing technique is the time to save checkpoint to stable storage. To mitigate this issue, multiple optimization techniques have been proposed, including incremental checkpointing, in-memory checkpointing, and multi-level checkpointing [14], [15], [16], [17]. Although well-explored, these techniques have not been widely adopted in HPC environments due to implementation complexity.

Recently, process replication has been proposed as a viable alternative to checkpointing in HPC, as replication can significantly increase system availability and achieve higher efficiency than checkpointing in failure-prone systems [18], [19]. In addition, full and partial replication have also been used to augment existing checkpointing techniques, and to guard against silent data corruption [20], [21], [22].

Many efforts aim at providing fault tolerance to MPI, which is the de facto programming paradigm for HPC. Checkpointing and message logging are supported by either building them from scratch or integrating with existing solutions [23], [24], [17]. Several replication schemes are implemented in MPI, with the runtime overhead ranging from negligible to 70%, depending on the application communication patterns [25], [11], [22]. ULFM provides a set of MPI extensions that enable the deployment of application specific fault tolerance strategies [26].

III. REJUVENATING SHADOWS MODEL

We adopt the fail-stop fault model, which defines that a process halts in response to a failure and its internal state and memory contents are irretrievably lost [27]. Furthermore, we assume that the machine on which failure occurred can be rebooted for starting a new process after the failure. Note that this is the same assumption for checkpointing/restart. Below we discuss the components of the Rejuvenating Shadows model and its design trade-offs.

A. Shadowing

The basic tenet of Shadowing is to associate with each process (referred to as main) a replica process (referred to as shadow), which potentially runs at a lower execution rate to save power and computing resources. Originally, if a main fails, its shadow is promoted to assume the role of the main. In this work, however, we deviate from the original concept of shadow as a replica [6] and use the state of the shadow to restore the main to its state prior to the failure.

State consistency between mains and shadows is required both during normal execution and following a failure. As depicted in Figure 1, we designed a protocol to enforce sequential consistency, i.e., each shadow sees the same message order and operation results as its main. Instead of sending messages from main to main and shadow to shadow [11], we choose to let the sending main forward each message to the receiving shadow. This allows us to speed up a single shadow when a main fails. We assume that two copies of the same message are sent in an atomic manner.[1] Note that, the SYNC message in Figure 1 is only used to address potential non-determinism as will be discussed in details in the next section.

To reduce the execution rate of the shadows, one can use Dynamic Voltage and Frequency Scaling (DVFS) [28],

[1] this property can be ensured, for example, by using the NIC multicast functionality of the network.

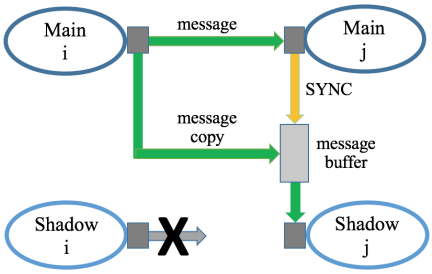


Figure 1. Consistency protocol for Rejuvenating Shadows.

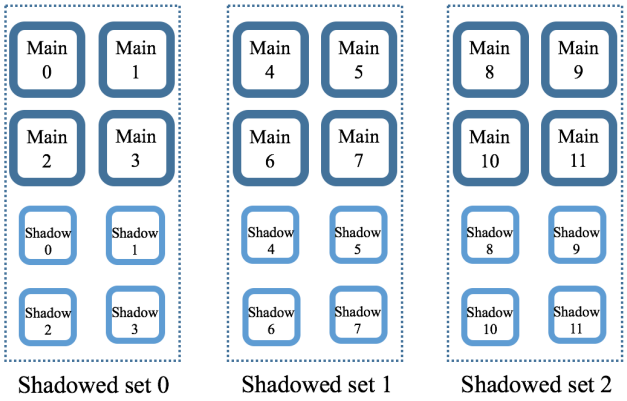


Figure 2. Logical organization of 12 mains and their shadows with every 4 shadows collocated, resulting in 3 shadowed sets.

process collocation [6], or a combination of both. Because of the undesirable consequences entailed by DVFS, such as reduced reliability and limited control granularity, we choose to use only collocation in this work. Collocation increases memory requirement for the nodes that execute shadows. Note that, however, this problem is not intrinsic to Rejuvenating Shadows, as in-memory and multi-level checkpointing also require extra memory. The number of shadows collocated on a processing unit is referred to as the *collocation ratio*. Also, the term *shadowed set* is used to refer to a set of mains and their associated shadows of which the shadows are collocated. Figure 2 shows an example of three shadowed sets with collocation ratio of 4.

B. Leaping

Since shadows execute slower than mains, recovery of a main failure requires a shadow to catch up, thus introduces delay to the execution. To mitigate this issue, we propose a technique, referred to as *Leaping*, that takes advantage of the recovery time to synchronize the state of the shadows with that of their non-faulty mains. As a result, shadows achieve forward progress with minimal overhead, and furthermore, the recovery time of future failures, if any, is minimized. Leaping always takes between a pair of main and shadow. To avoid ambiguity, the process which provides state in a leaping is referred to as the target process, and the other process which receives and updates state is referred to as the roll forward process.

C. Rejuvenation

Lazy Shadowing assumes that when a main fails, its shadow increases its execution rate to speed up recovery. With collocation, this is accomplished by terminating the other collocated shadows. As a result, however, only one instance of each task in the shadowed set remains, thereby making the shadow set vulnerable to future faults.

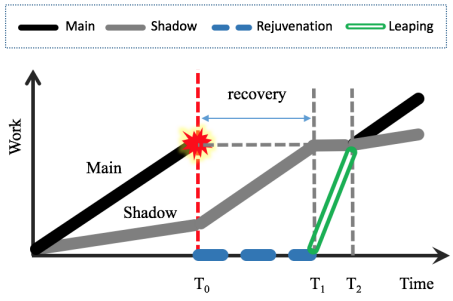
The shadow set vulnerability can be avoided by using rejuvenation, whereby a new process is spawned for either a failed man or shadow. This eliminates the need for a shadow to permanently substitute for a main. As result, collocated shadows need not to be terminated, but only temporarily suspended during the recovery process.

The problem, however, is that the newly spawned process will start from its initial state and potentially delay the entire execution. To deal with this issue, we extend the concept of leaping to synchronize the new process' state with the state of its associated living process. We ignore the case where both the main and shadow of a task fail simultaneously, due to the extremely low probability. If necessary, however, the shadow replication model is always able to achieve higher reliability with the cost of more shadows per task.

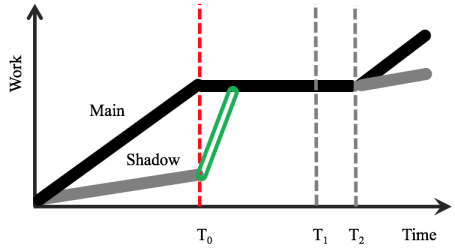
Figure 3 illustrates the failure recovery process with rejuvenation, assuming that a main M_i fails at time T_0. In order for its shadow S_i to speed up, the shadows collocated with S_i are temporarily suspended. Meanwhile, the failed processor is rebooted and then a new process is launched for M_i. When, at T_1, S_i catches up with the state of M_i before its failure, leaping is performed to advance the new process to the current state of S_i. The leaping is not initiated until S_i catches up so as to avoid the need of message logging.

Because of the failure of M_i, the other mains are blocked at the next synchronization point, which is assumed to be immediately after T_0. During the idle time, a leaping is opportunistically performed to transfer state from each living main to its shadow. Therefore, this leaping has minimal overhead as it overlaps with the recovery, as shown in Figure 3(b). Figure 3(c) shows that leaping for the shadows collocated with S_i are delayed until the recovery completes at T_1. After the leaping finishes at T_2, all mains and shadow resume normal execution with the same level of resilience as before the failure.

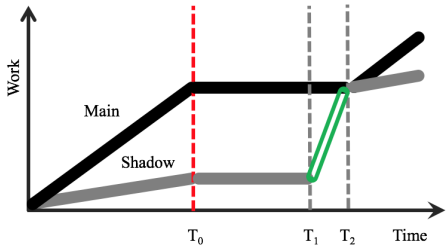
Figure 3 and the above analysis assume that the time for rebooting is no longer than the recovery time. If the new M_i is not yet ready when S_i catches up at T_1, however, we have two design choices: 1) S_i can continue execution and take the role of a main; or 2) S_i can wait for the new M_i to launch. The first option requires all the other processes to update their internal process mapping in order to correctly receive messages from this shadow (see Figure 1). This not only complicates the implementation, but also requires expensive global coordination that is detrimental to scalability. We therefore chose the second design option.



(a) Faulty task



(b) Non-faulty tasks in different shadowed sets



(c) Non-faulty tasks in the same shadowed set

Figure 3. Recovery and rejuvenation after a main process fails.

IV. IMPLEMENTATION

This section presents the details of our implementation of rsMPI, which is an MPI library for Rejuvenating Shadows. Similar to rMPI and RedMPI [11], [22], rsMPI is implemented as a separate layer between MPI and user application. It uses the standard MPI profiling interface to intercept every MPI call (using function wrappers) and enforces Rejuvenating Shadows logic. When used, rsMPI transparently spawns the shadow processes during the initialization phase, manages the coordination between main and shadow processes, and guarantees order and consistency for messages and non-deterministic MPI events.

A. MPI rank

A rsMPI world has 3 types of identities: main process, shadow process, and coordinator process that coordinates between main and shadow. A static mapping between rsMPI rank and application-visible MPI rank is maintained so that each process can retrieve its identity. For example, if the user specifies N processes to run with collocation ratio of 4, rsMPI will translate it into $N+N+N/4$ processes, with the first N ranks being the mains, the next N ranks being the shadows, and the last $N/4$ ranks being the coordinators. Using the MPI profiling interface, we added wrapper for MPI_Comm_rank() and MPI_Comm_size(), so that each process (main or shadow) gets its correct execution path.

B. Execution rate control

While the mains always execute at maximum rate for HPC's throughput consideration, the shadows are configured to execute slower by collocation as specified by user in a configuration file. Accordingly, rsMPI will generate an MPI rankfile and provide it to the MPI runtime to control the process mapping. Note that rsMPI always maps the main and shadow of the same task onto different nodes. This is preferred as a fault on a node will not affect both of them. To minimize resource usage, each coordinator is collocated with all the shadows in the shadowed set. Since a coordinator simply waits for incoming control messages (discussed below) and does minimal work, it has negligible impact on the execution rate of the collocated shadows.

C. Coordination between mains and shadows

Each shadowed set has a coordinator process dedicated to coordination between the mains and shadows in the shadowed set. Coordinators do not execute any application code, but just wait for rsMPI defined control messages, and then carry out some coordination work accordingly. There are three types of control messages: termination, failure, and leaping. Correspondingly, each coordinator is responsible for three actions:

- when a main finishes, it notifies its coordinator, which then forces the associated shadow to terminate.
- when a main fails, its associated shadow needs to catch up, so the coordinator temporarily suspends the other collocated shadows, and resumes their execution once the recovery is done.
- when a main initiates a leaping after detecting another main's failure, the coordinator triggers leaping at the associated shadow.

To separate control messages from data messages, rsMPI uses a dedicated MPI communicator for the control messages. This Control Communicator is created by the wrapper to the MPI_Init call. In addition, to ensure fast response and minimize the number of messages, coordinators also use OS signals to communicate with their collocated shadows.

D. Message passing and consistency

We wrapped around every MPI communication function and implemented the consistency protocol described in Section III-A. For sending functions, such as MPI_Send() and MPI_Isend(), rsMPI requires the main to duplicate the sending while the messages are suppressed at the shadow (see Figure 1). For receiving functions, such as MPI_Recv() and MPI_Irecv(), both the main and the shadow does one

receiving from the main process at the sending side. Internally, collective communication in rsMPI uses point-to-point communication in a binomial tree topology, which demonstrates excellent scalability.

We assume that only MPI operations can introduce non-determinism, and the SYNC message shown in Figure 1 is used to enforce consistency. For example, MPI_ANY_SOURCE may result in different message orders between a main and its shadow. To deal with this, we serialize the receiving of MPI_ANY_SOURCE messages by having the main finish the receiving and then use a SYNC message to forward the message source to its shadow, which then issues a receiving from the specific source.

E. Leaping

Different from Checkpointing where the process state is saved, leaping directly transfers process state between a main and its shadow. To reduce the size of data involved in saving state, rsMPI borrows the idea from application-level checkpointing [29], and provides the following API for users to register any data as process state:

```
void leap_register_state(void *addr, int count, \
     MPI_Datatype dt);
```

For each piece of data to register, three arguments are needed: a pointer to the memory address, the number of data items, and the datatype. Application developer could use domain knowledge to identify only necessary state data, or use compiler techniques to automate this [30].

rsMPI uses MPI messages to transfer process state. Although multiple pieces of data can be registered as a process' state, only a single message needs to be transferred, as MPI supports derived datatypes. To prevent the messages carrying process state from mixing with application messages, rsMPI uses a separate Control Communicator for transferring process state. With the synchronization of leaping by coordinator and the fast transfer of process state via MPI messages, the overhead of leaping is minimized.

To make sure a pair of main and shadow stay consistent after a leaping, not only user-defined states should be transferred, but also lower level states, such as program counter and message buffer, need to be correctly updated. Specifically, the roll forward process needs to satisfy two requirements: 1) it need to discard all obsolete messages after the leaping; 2) it need to resume execution at the same point as the target process. We discuss our solutions below, under the assumption that the application's main body consists of a loop, which is true in most cases.

To correctly discard all obsolete messages, rsMPI borrows the idea of "determinants" from message logging [9], and requires every main and shadow to log the metadata (i.e., MPI source, tag, and communicator) for all received messages. Then during leaping, the metadata at the target process is transferred to the roll forward process, so that the later can combine MPI probe and receive to remove the messages that have been consumed by the former but not by itself.

To resume execution from the same point, we restrict leaping to always occur at certain possible points, and use internal counter to make sure that both the roll forward and target processes start leaping from the same point. For example, when a main initiates a leaping, the coordinator will trigger a signal handler at the associated shadow. The signal handler does not carry out leaping, but sets a flag for leaping and receives from its main a counter value that indicates the leaping point. Only when both the flag is set and counter value matches will the shadow start leaping. In this way, it is guaranteed that after leaping the main and shadow will resume execution from the same point. To balance the trade-off between implementation overhead and flexibility, we choose MPI receive operations as the only possible leaping points.

V. EVALUATION

We deployed rsMPI on a medium sized cluster and utilized up to 21 nodes (420 cores) for testing and benchmarking. Each node consists of a 2-way SMPs with Intel Haswell E5-2660 v3 processors of 10 cores per socket (20 cores per node), and is configured with 128 GB RAM. Nodes are connected via 56 GB/s FDR InfiniBand.

We used benchmarks from the Sandia National Lab Mantevo Project and NAS Parallel Benchmarks (NPB), and evaluated rsMPI with various problem sizes and number of processes. CoMD is a proxy for molecular dynamics application. MiniAero is an explicit unstructured finite volume code that solves the Navier-Stokes equations. Both MiniFE and HPCCG are unstructured implicit finite element codes, but HPCCG uses MPI_ANY_SOURCE receive operations and can demonstrate rsMPI's capability of handling non-deterministic events. IS, EP, and CG from NPB represent integer sort, embarrassingly parallel, and conjugate gradient applications, respectively. These applications cover key simulation workloads and represent both different communication patterns and computation-to-communication ratios.

We also implemented double in-memory checkpointing to compare with rsMPI in the presence of failures. Same as leaping in rsMPI, our application-level checkpointing provides an API for process state registration. This API requires the same parameters, but internally, it allocates extra memory in order to store the state of a "buddy" process. Another provided API is checkpoint(), which inserts a checkpoint in the application code. For fairness, MPI messages are used to transfer state between buddies. For both rsMPI and checkpointing/restart, we assume a 60 seconds rebooting time after a failure.

A. Measurement of runtime overhead

While the hardware overhead for rsMPI is straightforward (e.g., collocation ratio of 4 results in the need for 25%

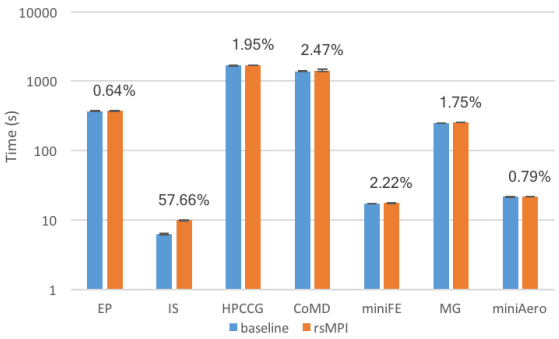


Figure 4. Comparison of execution time between baseline and rsMPI using 256 application-visible processes and collocation ratio of 2 for rsMPI.

more hardware cost), the runtime overhead of the enforced consistency protocol depend on applications. To measure this overhead we ran each benchmark application linked to rsMPI and compared the execution time with the baseline, where each application runs with unmodified OpenMPI.

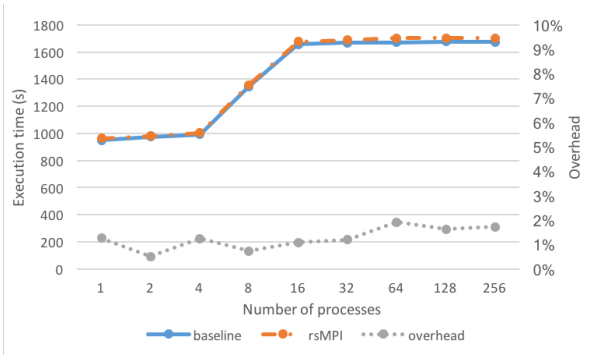
Figure 4 shows the comparison of the execution time for the 7 applications in the absence of faults. All the experiments are conducted with 256 application-visible processes. That is, the baseline always uses 256 MPI ranks, while rsMPI uses 256 mains together with 256 shadows. Each result shows the average execution time of 5 runs, the standard deviation, and rsMPI's runtime overhead.

From the figure we can see that rsMPI has comparable execution time to the baseline for all applications except IS. The reason for the exception of IS is that IS uses all-to-all communication and is heavily communication-intensive. We argue that communication-intensive applications like IS are not scalable, and as a result, they are not suitable for large-scale HPC. For all other applications, the overhead varies from 0.64% (EP) to 2.47% (CoMD). Even for HPCCG, which uses MPI_ANY_SOURCE and adds extra work to our consistency protocol, the overhead is only 1.95%, thanks to the asynchronous semantics of MPI_Send. Therefore, we conclude that rsMPI's runtime overheads are modest for scalable HPC applications that exhibit a fair communication-to-computation ratio.

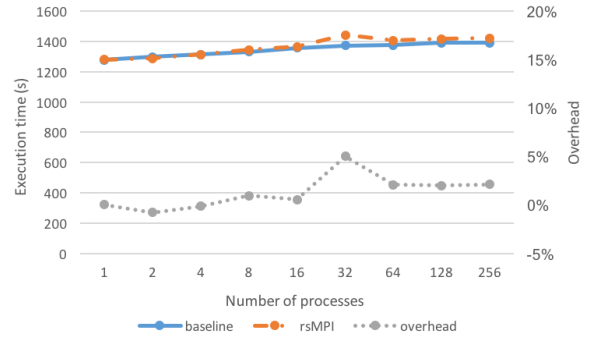
B. Scalability

We also assessed the applications' weak scalability, which is defined as how the execution time varies with the number of processes for a fixed problem size per process. Among the seven applications, HPCCG, CoMD, and miniAero allow us to configure the input for weak scaling test. The results for miniAero are similar to those of CoMD, so we only show the results for HPCCG and CoMD in Figure 5.

Comparing between Figure 5(a) and Figure 5(b), it is obvious that HPCCG and CoMD have different weak scaling characteristics. While the execution time for CoMD increases by 8.9% from 1 process to 256 processes, the execution time is almost doubled for HPCCG. However, further analysis shows that from 16 to 256 processes, the



(a) HPCCG weak scalability



(b) CoMD weak scalability

Figure 5. Scalability test for number of processes from 1 to 256. Collocation ratio is 2 for rsMPI.

execution time increases by only 2.5% for CoMD, and 1.0% for HPCCG. We suspect that the results are not only determined by the scalability of the application, but also impacted by other factors, such as cache and memory contention on the same node, and network interference from other jobs running on the cluster. To predict the overhead at exascale, we applied curve fitting to derive the correlation between runtime overhead and the number of processes. At 2^{20} processes, it is projected that the overhead is 3.1% for CoMD and 7.6% for HPCCG.

C. Performance under failures

As one main goal of this work is to achieve fault tolerance, an integrated fault injector is required to evaluate the effectiveness and efficiency of rsMPI to tolerate failures during execution. To produce failures in a manner similar to naturally occurring process failures, our failure injector is designed to be distributed and co-exist with all rsMPI processes. Failure is injected by sending a specific signal to a randomly picked target process.

We assume that the underlying hardware platform has a Reliability, Availability and Serviceability (RAS) system that provides failure detection. In our test system, we emulate the RAS functionality by associating a signal handler with every process. The signal handler catches failure signals sent from the failure injector, and uses a rsMPI defined failure message via a dedicated communicator to notify all other processes.

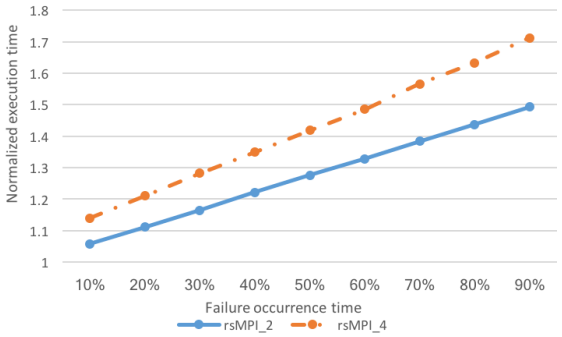


Figure 6. Execution time of HPCCG with a single failure injected at various time.

The first step was to test the effectiveness of leaping. Figure 6 shows the execution time of HPCCG with a single failure injected at a specific time, measured as a proportion of the total execution of the application. The execution time is normalized to that of the fault-free baseline. The blue solid line and red dashed line represent rsMPI with collocation ratio of 2 and 4, respectively. For simplicity, they are referred to as rsMPI_2 and rsMPI_4 in the following text.

As shown in Figure 6, rsMPI's execution time increases with the failure occurrence time, regardless of the collocation ratio. The reason is that recovery time in rsMPI is proportional to the amount of divergence between mains and shadows, which grows as the execution proceeds. Another factor that determines the divergence is the shadow's execution rate. The slower the shadows execute, the faster the divergence grows. As a result, rsMPI_2 can recover faster than rsMPI_4, and therefore achieves better execution time.

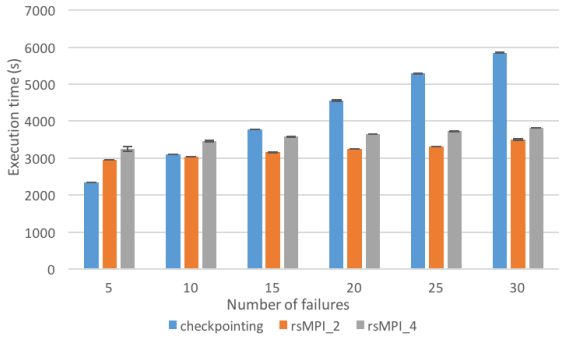
The results in Figure 6 suggests that rsMPI is better suited to environments where failures are frequent. This stems from the fact that, due to leaping, the divergence between mains and shadows is eliminated after every failure recovery. To demonstrate the above analysis, we compare rsMPI with checkpointing under various failure rates.

For fairness, we take into account the extra hardware cost for rsMPI by defining the following metric:

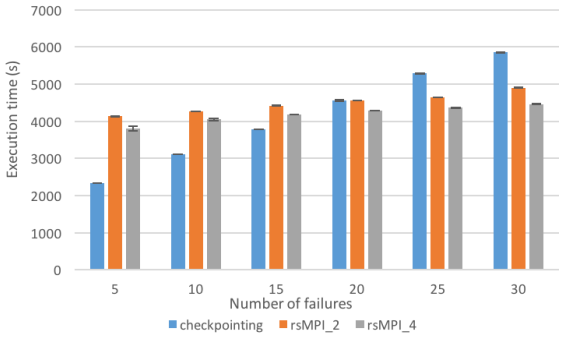
$$\text{Weighted execution time} = T_e \times S_p,$$

where T_e is the wall-clock execution time and S_p is the projected speedup. For example, we measured that the speedup of HPCCG from 128 processes to 256 processes is 1.88, and rsMPI_2 needs 1.5 times more nodes than checkpointing, so the projected speedup is $1.5 \times \frac{1.88}{2} = 1.41$. Similarly, we calculate the projected speedup for rsMPI_4 as $1.25 \times \frac{1.88}{2} = 1.17$.

In this analysis, we set the checkpointing interval to $0.1T$, where T is the total execution time. To emulate failures, for both checkpointing and rsMPI, we randomly inject over T a number of faults, K, ranging from 5 to 30. This fault rate corresponds to a processor's MTBF of NT/K, where N is the number of processors. That is, the processor's MTBF is proportional to the total execution time and the number



(a) Wall-clock execution time



(b) Weighted execution time

Figure 7. Comparison between checkpointing and rsMPI with various number of failures injected to HPCCG. 256 application-visible processes, 10% checkpointing interval.

of processors. For example, when using a system of 64,000 processors and executing over 4 hours, injecting 10 faults corresponds to a processor's MTBF of 3 years.

Figure 7 compares checkpointing and rsMPI (rsMPI_2 and rsMPI_4), based on both wall-clock and weighted execution time. Without taking into consideration the hardware overhead, Figure 7(a) shows that, when the number of failures is small (e.g., 5 failures), checkpointing slightly outperforms rsMPI, with respect to wall-clock execution time. As the number of failures increases, however, rsMPI achieves significantly higher performance than checkpointing. For example, when the number of failures is 20, rsMPI_2 saves 28.7% in time compared to checkpointing. The saving rises up to 39.3%, when the number of failures is increased to 30. The results also show that, compared to checkpointing, rsMPI_4 reduces the execution time by 19.7% and 34.8%, when the number of failures are 20 and 30, respectively.

Considering both execution time and hardware overhead, Figure 7(b) compares the weighted execution time between checkpointing and rsMPI. As expected, checkpointing is better when the number of failures is small (e.g., 5 failures). When the number of failures increases, however, checkpointing loses its advantage quickly. At 30 failures, for example, rsMPI_2 and rsMPI_4 are 19.3% and 31.3% more efficient than checkpointing, respectively. Note that, when comparing rsMPI_2 and rsMPI_4, the former shows higher performance

with respect to wall-clock execution time, while the latter exhibits higher performance with respect to weighted execution time.

VI. CONCLUSION AND FUTURE WORK

In this paper, we propose a novel fault tolerance model of Rejuvenating Shadows. Careful design guarantees that Rejuvenating Shadows always achieves forward progress in the presence of failures, maintains consistent level of resilience, and minimizes implementation complexity and runtime overhead. Through empirical experiments, we demonstrated that Rejuvenating Shadows outperforms checkpointing/restart in both execution time and resource utilization, especially in failure-prone environments.

Leaping induced by failure has proven to be a critical mechanism in reducing the divergence between a main and its shadow, thus reducing the recovery time for subsequent failures. Consequently, the time to recover from a failure increases with failure intervals. Based on this observation, a proactive approach is to "force" leaping when the divergence between a main and its shadow exceeds a specified threshold. In our future work, we will study the concept to determine what behavior triggers forced leaping to optimize the average recovery time.

ACKNOWLEDGMENT

This research is based in part upon work supported by the Department of Energy under contract DE-SC0014376.

REFERENCES

[1] K. Bergman, S. Borkar, and et. al., "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office, Tech. Rep*, vol. 15, 2008.

[2] R. A. Oldfield, S. Arunagiri, and et. al., "Modeling the impact of checkpoints on next-generation systems," in *MSST'07*, Sept, pp. 30–46.

[3] E. N. Elnozahy and J. S. Plank, "Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 97–108, April 2004.

[4] J. F. Bartlett, "A nonstop kernel," in *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, ser. SOSP '81. New York, NY, USA: ACM, 1981, pp. 22–29.

[5] B. Mills, "Power-aware resilience for exascale computing," Ph.D. dissertation, University of Pittsburgh, 2014.

[6] X. Cui, T. Znati, and R. Melhem, "Adaptive and power-aware resilience for extreme-scale computing," in *Scalcom'16*, Toulouse, France, July 18-21 2016.

[7] T. Hérault and Y. Robert, *Fault-Tolerance Techniques for High-Performance Computing*. Springer, 2015.

[8] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

[9] E. Elnozahy and et. al., "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[10] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, "Fault prediction under the microscope: A closer look into hpc systems," in *SC'12*, p. 77.

[11] K. Ferreira, J. Stearley, and et. al., "Evaluating the viability of process replication reliability for exascale systems," ser. SC '11, pp. 44:1–44:12.

[12] P. Hargrove and J. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, no. 1, 2006, p. 494.

[13] A. Guermouche and et. al., "Uncoordinated checkpointing without domino effect for send-deterministic mpi applications," in *IPDPS*, May 2011, pp. 989–1000.

[14] S. Gao and et. al., "Real-time in-memory checkpointing for future hybrid memory systems," in *Proceedings of the 29th International Conference on Supercomputing*, 2015.

[15] S. Agarwal and et. al., "Adaptive incremental checkpointing for massively parallel systems," in *ICS 04*, St. Malo, France.

[16] G. Zheng, L. Shi, and L. V. Kalé, "Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi," in *Cluster Computing*. IEEE, 2004, pp. 93–103.

[17] A. Moody, G. Bronevetsky, K. Mohror, and B. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC*, 2010, pp. 1–11.

[18] R. Riesen, K. Ferreira, J. R. Stearley, R. Oldfield, J. H. L. III, K. T. Pedretti, and R. Brightwell, "Redundant computing for exascale systems," December 2010.

[19] F. Cappello, "Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities," *IJHPCA*, vol. 23, no. 3, pp. 212–226, 2009.

[20] J. Elliott and et. al., "Combining partial redundancy and checkpointing for HPC," in *ICDCS '12*, Washington, DC, US.

[21] X. Ni, E. Meneses, N. Jain, and L. V. Kalé, "Acr: Automatic checkpoint/restart for soft and hard error protection," ser. SC. New York, NY, USA: ACM, 2013, pp. 7:1–7:12.

[22] D. Fiala and et. al., "Detection and correction of silent data corruption for large-scale high-performance computing," ser. SC, Los Alamitos, CA, USA, 2012.

[23] G. Bosilca and et. al., "Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes," in *Supercomputing, ACM/IEEE 2002 Conference*. IEEE, 2002, pp. 29–29.

[24] M. Gamell, D. S. Katz, and et. al., "Exploring automatic, online failure recovery for scientific applications at extreme scales," in *SC'14*, 2014, pp. 895–906.

[25] C. Engelmann and S. Böhm, "Redundant execution of hpc applications with mr-mpi," in *PDCN*, 2011, pp. 15–17.

[26] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra, "An evaluation of user-level failure mitigation support in mpi," ser. EuroMPI'12, 2012, pp. 193–203.

[27] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Trans. Comput. Syst.*, vol. 1, no. 3, pp. 222–238, 1983.

[28] X. Cui and et. al., "Shadow replication: An energy-aware, fault-tolerant computational model for green cloud computing," *Energies*, vol. 7, no. 8, pp. 5151–5176, 2014.

[29] A. Beguelin and et. al., "Application level fault tolerance in heterogeneous networks of workstations," *Journal of Parallel and Distributed Computing*, vol. 43, pp. 147–155, 1997.

[30] G. Bronevetsky, D. Marques, and et. al., "Compiler-enhanced incremental checkpointing for openmp applications," in *IPDPS*, May 2009, pp. 1–12.