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, making fault tolerance a major challenge. This work explores novel methodologies to fault tolerance that achieve forward recovery, power-awareness, and scalability. The proposed model, referred to as Rejuvenating Shadows, is able to deal with multiple types of failure and maintain consistent level of resilience. An implementation is provided for MPI, and empirically evaluated with various benchmark applications that represent a wide range of HPC workloads. The results demonstrate Rejuvenating Shadows' ability to tolerate high failure rates, and to outperform in-memory checkpointing/restart in both execution time and resource utilization.**

***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 enormous number of components, order of magnitudes higher than in existing HPC systems, which will lead to frequent system failures, significantly limiting computational progress. Another challenge stems from the stringent requirement, set by the US Department of Energy, to operate in a power envelope of 20 Megawatts [1]. This puts into question the viability of traditional fault-tolerance methods and calls for a reconsideration of the resilience and power-awareness problems, at scale.

A common approach to resilience relies on time redundancy through checkpointing and rollback recovery. During normal execution, the computation state is periodically saved to a stable storage to allow recovery from a failure by restarting from a checkpoint. 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 as well as 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 achieve resilience in extreme-scale, failure-prone computing environments [5]. Its basic tenet is to associate each main process with a suite of coordinated shadow processes, to deal with failures and meet the performance requirements of the underlying application. For elastic applications, a single shadow that runs as a replica of its associated main, but at a lower execution rate, would be sufficient to achieve fault tolerance while maintaining acceptable response time. Based on this approach, the Lazy Shadowing scheme has been experimentally demonstrated to achieve higher performance and significant energy savings in comparison to existing approaches in most cases [6].

Despite its ability to tolerate failure in a wide range of exascale computing environments, Lazy Shadowing assumes that either the main or the shadow fails, but not both. Consequently, the resilience 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 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 maintain consistent level of resilience. In this new model, shadows are no longer replicas, that are promoted to substitute for their associated mains upon a failure. Instead, each shadow is a "rescuer" whose role is to restore the associated main to its exact state before failure. Rejuvenating Shadows can handle both transient and crash failures. The main contributions of this paper are as follows:

- A new fault tolerance model, Rejuvenating Shadows, to deal with different types of failure and preserve resilience across failures.
- A full-feature implementation of Rejuvenating Shadows

into Message Passing Interface (MPI) to enhance its tolerance to failure at scale.

- The implementation of an application-level in-memory Checkpointing/Restart scheme, which is used for comparative study.
- A thorough evaluation and comparative analysis of the overhead and performance of the Rejuvenating Shadows, using multiple benchmark applications, under various failures.

The rest of the paper is organized as follows. A review of related work is provided 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 in HPC has been the focus of research, with significant progress on how we mitigate the impact of failures [7]. Checkpointing/restart periodically saves the execution state to a stable storage, with the anticipation that, in case of failure, computation can be restarted from a saved checkpoint [8]. Message logging protocols, which combines checkpointing with logging of non-deterministic events, allow the system to recover beyond the most recent consistent checkpoint [9]. Proactive fault tolerance relies on a prediction model to forecast faults, so that preventive measures, such as task migration or checkpointing, can be undertaken [10]. Algorithm-based fault tolerance (ABFT) uses redundant information inherent to its computational and algorithmic structure of the problem to achieve resilience. For the past 30 years, however, 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 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 [14].

One of the largest overheads in disk-based checkpointing techniques is the time to save a checkpoint to a stable storage. Multiple optimization techniques have been proposed to reduce this overhead, including incremental checkpointing, in-memory checkpointing, and multi-level checkpointing [15], [16], [17], [18]. Although well-explored, these techniques have not been widely adopted in HPC environments due to their implementation complexity.

Recently, process replication has been proposed as a viable alternative to checkpointing in HPC, as replication can significantly increase system availability and potentially achieve higher efficiency in failure prone systems [19], [20]. Several replication schemes are implemented in MPI, with the runtime overhead ranging from negligible to 70%, depending on the application communication patterns [21], [11]. In addition, full and partial replication are used to augment existing checkpointing techniques, and to guard against silent data corruption [22], [23], [24].

Rejuvenating Shadows takes a different approach from these protocols and explores the trade-offs between time and hardware redundancy to achieve fault tolerance. The flexibility in balancing these trade-offs, together with the ability to rejuvenate after failures, allows Rejuvenating Shadows to maximize system efficiency and maintain a consistent level of resilience to failure.

III. REJUVENATING SHADOWS MODEL

Rejuvenating Shadows assumes a fail-stop fault model, whereby a failed process halts and its internal state and memory content are irretrievably lost [25]. Furthermore, we assume that the processor on which the failure occurred can be rebooted and used to start a new process[1]. This assumption also holds for checkpointing/restart to ensure fair comparative analysis.

Rejuvenating Shadows is a shadow replication based fault tolerance model, which associates a shadow to each main process. To achieve fault tolerance, shadows execute simultaneously with the mains, but on different nodes. Furthermore, to save power, shadows execute at a lower rate than their associated mains. When a main fails, the shadow increases its execution rate to speed up recovery. However, contrary to Lazy Shadowing, where a shadow substitutes for its failed main, Rejuvenating Shadows uses a shadow as a *rescuer* to restore the state of the main to where it was before failure. Upon state restoration, both the main and its associated shadow are active, thereby bringing back the system to its original level of fault tolerance.

To reduce a shadow's execution rate, Dynamic Voltage and Frequency Scaling (DVFS) can be applied. Its effectiveness, however, may be markedly limited by the granularity of voltage control, the number of frequencies available, and the negative effects on reliability [26]. An alternative approach to DVFS is to collocate multiple shadows on a single processor [6]. The desired execution rate can be achieved by controlling the collocation ratio, defined as the number of shadows that time-share a processor. An example of 3 *shadowed sets*, with a collocation ratio of 4, is depicted in Figure 1. A shadowed set refers to a set of mains and their collocated shadows.

Contrary to pure replication, shadow collocation reduces the number of processors required to achieve fault-tolerance, thereby reducing power and energy consumption. Furthermore, the collocation ratio can be adapted to reflect the

[1] Equivalently, a spare processor can be used for this purpose.

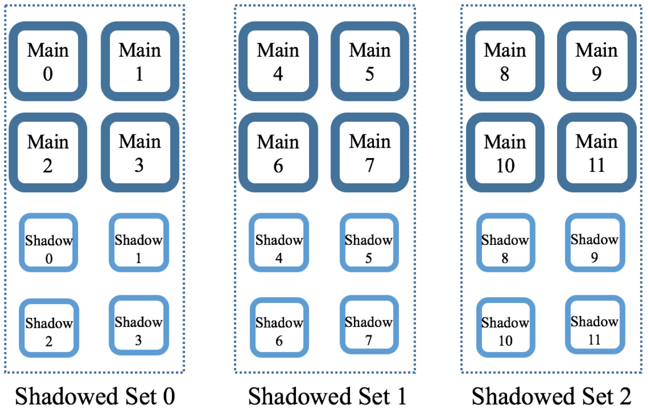


Figure 1. Logical organization of 12 mains and their shadows with every 4 shadows collocated, resulting in 3 shadowed sets.

propensity of the system to failure. This flexibility, however, comes at the increased cost of memory requirement at the shared processor. It is to be noted that this limitation is not intrinsic to Rejuvenating Shadows, as in-memory and multi-level checkpointing also require additional memory to store checkpoints. In this work, we only focus on collocation to reduce execution rate.

In the basic form of shadow replication, failures can have a significant impact on the performance. Since a shadow executes at a lower rate than its associated main, a *divergence* is likely to occur between the pair of processes, as computation progresses. The impact of computational divergence on time-to-completion can be prohibitive, as the time required for a lagging shadow to “catch up” with its associated main can be significant. Failures can also impact the resilience of the system. Upon failure of a main, the system can only rely on an “orphan” shadow to complete the task. A trivial approach to address this shortcoming is to associate a “suite” of shadows with each main. Such an approach, however, is resource wasteful and costly in terms of energy consumption. To address the impact of computational divergence on system performance, while maintaining a persistent resilience level against multiple failures, two techniques, *leaping* and *rejuvenation*, are proposed.

A. *Leaping*

The main objective of *leaping* is to ensure forward progress in the presence of failure. As stated above, the failure of a main may cause the remaining processes to halt at a synchronization point, until the shadow associated with the failed main catches up. This process can increase significantly the time-to-completion of the task. Leaping takes advantage of the idle time to synchronize the state of the shadows with that of their non-failed mains. Leaping eliminates current divergence between non-failed mains and their associated shadows, with minimal computational and communication overhead. As such, it reduces significantly the recovery time of future failures. Leaping always takes place between a main and a shadow pair, and does not require global coordination. The process which provides the leaping state is referred to as the *leap-provider*, while the rolling-forward process which receives the leaping state is referred to as the *leap-recipient*.

B. *Rejuvenation*

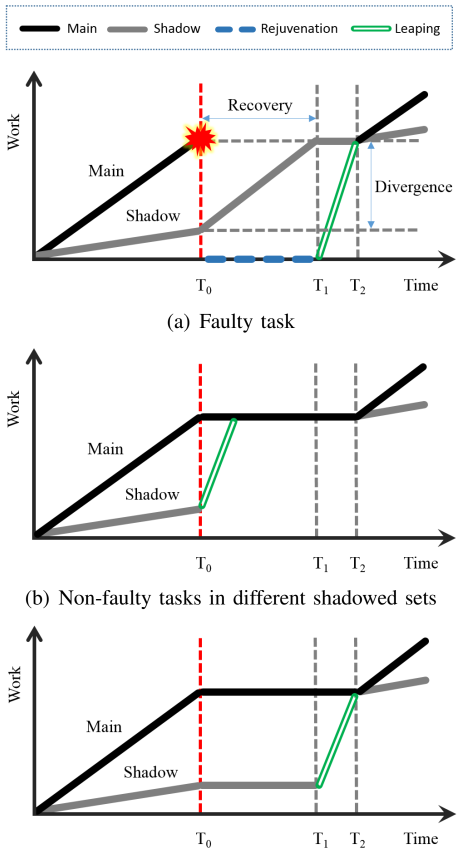
The main objective of rejuvenation is to enable the system to maintain its intended level of resilience, in the event of multiple failures[2]. Based on the shadow replication model, when a main fails, its associated shadow increases its execution rate to catch up with the failed main. The execution rate increase is achieved by terminating the remaining collocated shadows in the shadowed set, and allocating the processor to be used exclusively by the recovering shadow. Although efficient in reducing failure recovery time, this method reduces the execution of each task in the shadowed set to a single instance, thereby increasing the vulnerability of the system to subsequent failures. The proposed approach to address this shortcoming is to *rejuvenate* the failed process, by launching a new process. Furthermore, rather than staring the newly launched process from its initial state, leaping is invoked to synchronize the new process’ state with that of its associated living process. A direct implication of rejuvenation is that collocated shadows no longer need to be terminated, but only suspended until the recovery process is complete. In addition to restoring the system to its intended resilience level, rejuvenation also reduces the overall execution time.

Figure 2 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.

Because of the failure of M_i, the other mains are blocked at the next synchronization point, which is assumed to take place 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 2(b). Figure 2(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 2 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, two

[2] The case where both the main and shadow of a task fail simultaneously is not discussed specifically, as the occurrence of such an event is highly unlikely. Such failures can be handled by the shadow replication model by associating a suite of shadows to each main process.



(a) Faulty task

(b) Non-faulty tasks in different shadowed sets

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

Figure 2. Recovery and rejuvenation after a main process fails.

design choices are possible. In the first, S_i can assume the role of a main and continue execution. In the second, S_i can wait until the launching of the new M_i is complete. The first option requires that all non-failed processes update their internal mapping to identify the shadow as a new main and continue to correctly receive messages (see Figure 3). This not only complicates the implementation, but also requires expensive global coordination that is detrimental to scalability. We, therefore, chose the second design option.

IV. IMPLEMENTATION

This section presents the details of our rsMPI implementation of Rejuvenating Shadows, as an MPI library. Similar to rMPI and RedMPI [11], [24], 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.

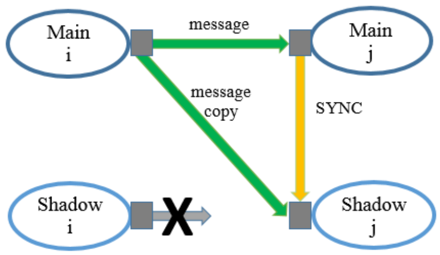


Figure 3. Consistency protocol for Rejuvenating Shadows.

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 each main executes on a separate processor at maximum rate for HPC's throughput consideration, shadows are configured to collocate and execute at a slower rate based on a user configuration file. Accordingly, rsMPI will generate an MPI rankfile and provide it to the MPI runtime to control process mapping. Note that rsMPI always maps the main and shadow of the same task onto different nodes. This is required to prevent a fault on one node from affecting both a main and its associated shadow. To minimize resource usage, each coordinator is collocated with the shadows in the shadowed set. A coordinator performs minimal work, as its main task is to simply handle incoming control messages (discussed below). As such, the impact of coordinator on the execution rate of collocated shadows is minimal.

C. Message passing and consistency

State consistency between mains and shadows is required both during normal execution and following a failure. As depicted in Figure 3, a protocol is used 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 main sender forward each message to the shadow receiver. 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[3].

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

For MPI point-to-point communication routines, we add wrappers to implement the above consistency protocol. Collective communication in rsMPI uses point-to-point communication in a binomial tree topology, which demonstrates excellent scalability. Assuming that only MPI operations can introduce non-determinism, the SYNC message shown in Figure 3 is used to enforce consistency in such cases. 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 receives from the specific source.

D. Coordination between mains and shadows

Each shadowed set has a coordinator process dedicated to coordination between the mains and shadows in the set. Coordinators do not execute 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. They corresponds to three actions:

- When a main finishes, the coordinator forces the associated shadow to terminate.
- When a main fails, the coordinator temporarily speeds up the associated shadow by suspending the other collocated shadows, until the recovery is done.
- When a main initiates a leaping, 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.

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 [27], 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 [28].

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 isolate state messages from application messages, rsMPI uses a separate MPI communicator to transfer process state. By using a coordinator to synchronize the leaping process and relying on MPI messages to rapidly transfer process state, 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 buffers, need to be correctly updated. Specifically, the leap-recipient needs to satisfy two requirements: 1) Discard all obsolete messages after the leaping; 2) Resume execution at the same point as the leap-provider. We discuss our solutions below, under the assumption that the application's main body consists of a loop, which is true in most HPC applications.

To correctly discard all obsolete messages, rsMPI borrows the idea of "determinants" from message logging [14], 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 leap-provider is transferred to the leap-recipient, 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 an internal counter to make sure that both the leap-recipient and leap-provider start leaping from the same point. For example, when a main initiates a leaping, the coordinator will trigger a specific 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 leap-recipient and leap-provider 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.

Benchmarks from the Sandia National Lab Mantevo Project and NAS Parallel Benchmarks (NPB) are used. 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 [17] 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% 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). Therefore, we conclude that rsMPI's runtime overheads are modest for 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.

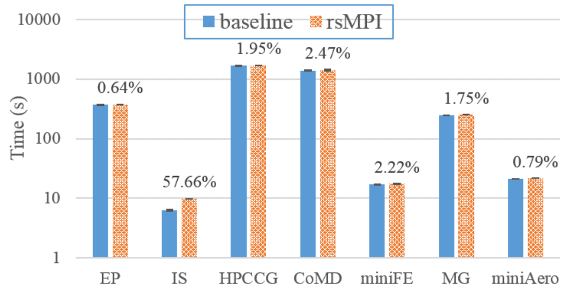
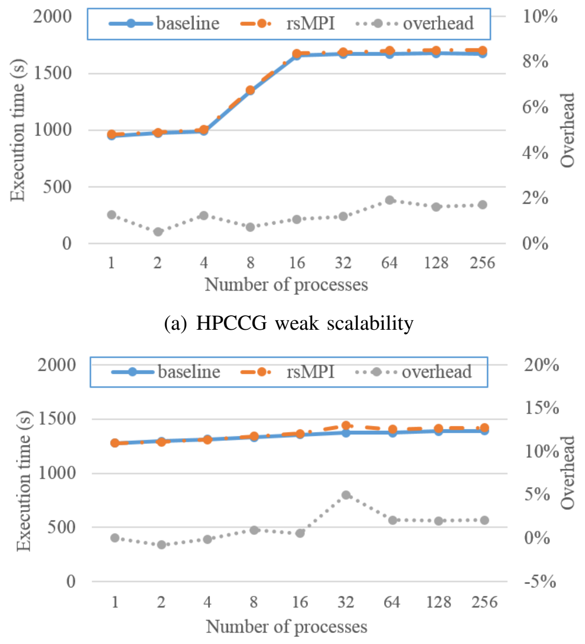


Figure 4. Comparison of execution time between baseline and rsMPI using 256 application-visible processes and collocation ratio of 2 for rsMPI.



(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.

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 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 produce failures in a manner similar to naturally occurring process failures,

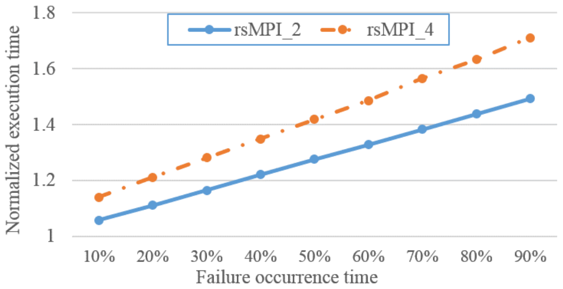


Figure 6. Execution time of HPCCG with a single failure injected at various time, normalized to that of the failure-free baseline.

the 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.

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 failure-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.

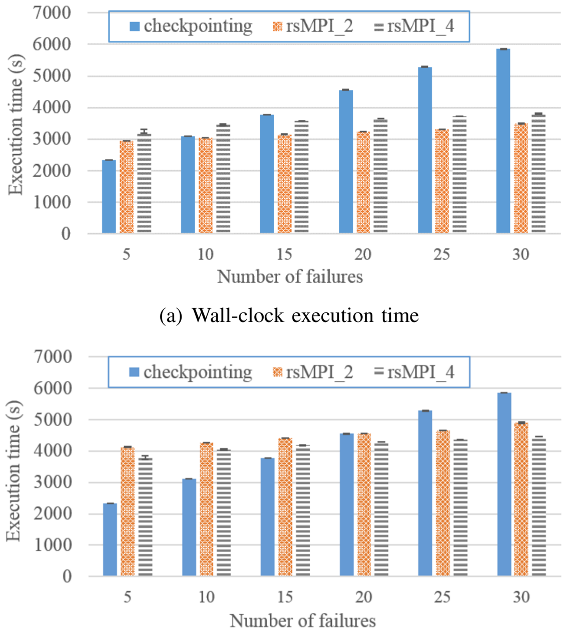
As shown in Figure 6, rsMPI's execution time increases with the failure occurrence time, regardless of the collocation ratio. The reason is that recovery time in rsMPI is proportional to the amount of divergence between mains and shadows, which grows as the execution proceeds. Another factor that determines 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 $1.25 \times \frac{1.88}{2} = 1.17$.



(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.

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 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, which guarantees forward progress in the presence of failures, maintains consistent level of resilience, and minimizes implementation complexity and runtime overhead. Empirical experiments demonstrated that the Rejuvenating Shadows model outperforms in-memory 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 further study this approach to determine what behavior triggers forced leaping in order 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] R. E. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems*, vol. 3, pp. 204–226, 1985.

[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] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

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

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

[23] 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.

[24] 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.

[25] 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.

[26] S. Eyerman and L. Eeckhout, "Fine-grained dvfs using on-chip regulators," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 1, pp. 1:1–1:24, Feb. 2011.

[27] 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.

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