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 goes beyond adapting or optimizing well known and proven techniques, and explores radically different 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 time redundancy through 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 rollback approaches the system's Mean Time Between Failures (MTBF), which leads to a significant drop in efficiency and increase in power [2], [3], [4]. A number of approaches have been proposed to address this shortcoming, focusing on hybrid in-memory and multilevel checkpointing methods [5], which rely both on coordinated and uncoordinated checkpointing. Some of these methods require that each process maintain a log of all incoming messages.

A second approach to resilience is replication, which exploits hardware redundancy by executing simultaneously multiple instances of the same task on separate processors [6]. 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 [7]. 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 [8], 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 *Rejuvenat-*

ing 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 the occurrence of 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 and improved performance.
- 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 [9]. Checkpointing/restart periodically saves the execution state to stable storage, with the anticipation that computation can be restarted from a saved checkpoint in the case of a failure. Assuming that all non-deterministic events can be identified, message logging protocols allow a system to recover beyond the most recent consistent checkpoint by combining checkpointing with logging of non-deterministic events [2]. Another way to cope with faults is proactive fault tolerance, which relies on a prediction model to forecast faults ahead of time and take preventive measures, such as task migration or checkpointing the application [10], [11]. Algorithm-based fault tolerance (ABFT) uses redundant information inherent of its coding of the problem to achieve resilience. Although the efficiency of ABFT can be orders of magnitude higher than that of general techniques, it lacks generality [9].

For the past 30 years, disk-based coordinated checkpointing has been the primary fault tolerance mechanism in production HPC systems [12]. To achieve a globally consistent state, all processes coordinate with one another to produce individual states that satisfy the "happens before" relationship [13]. Coordinated Checkpointing gains its popularity from its simplicity and ease of implementation. Its major drawback, however, is the lack of scalability [14]. Uncoordinated checkpointing allows processes to record their states independently, thereby reducing the overhead during fault free operation [15]. However, the scheme requires that each process maintain multiple checkpoints, necessary to construct a consistent state during recovery, and complicates the garbage collection scheme. It can also suffer the well-known domino effect [16].

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, semi-blocking checkpointing, in-memory checkpointing, and multi-level checkpointing [17], [18], [19], [20], [21], [22], [23]. 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 [24], [25]. In addition, full and partial replication have also been used to augment existing checkpointing techniques, and to guard against silent data corruption [26], [27], [28].

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 [29], [30], [31], [32], [22]. Several replication schemes are implemented in MPI, with the runtime overhead ranging from negligible to 70%, depending on the application communication patterns [33], [12], [28]. ULFM provides a set of MPI extensions that enable the deployment of application specific fault tolerance strategies [34].

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 [35]. Furthermore, we assume that the machine on which failure occurred can be rebooted (or equivalently, a spare machine is available) 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. When a main fails, its associated shadow increases its execution rate to complete the task. In essence,

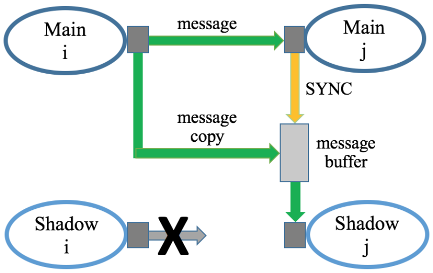


Figure 1. Consistency protocol for Rejuvenating Shadows.

if a main fails, its shadow is promoted to assume the role of the main and completes the task. In this work, however, we deviate from the original concept of shadow as a replica [8] 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. We designed a protocol, depicted in Figure 1, to enforce sequential consistency. In the context of shadowing, sequential consistency ensures that each shadow sees messages in the same order and operation results as its main. For each message, the sending main sends a copy of the message to both the receiving main and its shadow. Moreover, the shadow of the sender is suppressed from sending out messages. 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) [36], process collocation [8], or a combination of both. Because of the undesirable consequences entailed by the use of DVFS, such as impact on reliability, limited control granularity, and underutilization of hardware resources, we choose to use only collocation as the mechanism to control the shadows' execution rate in this work. The implementation details are presented in Section IV-B. When collocation is used, 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 shadows collocated on the same processing unit and their mains. Figure 2 shows an example with three shadowed sets and collocation ratio of 4.

B. Leaping

Leaping was initially proposed as a technique to boost the performance of lazy shadows [8]. Since shadows execute slower than their associated mains, failure recovery requires a shadow to catch up and thus delays the completion of the execution. Leaping takes advantage of the recovery time

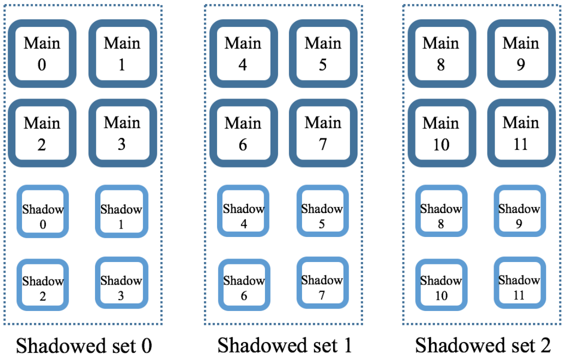


Figure 2. Logical organization of 12 mains and their shadows with every 4 shadows collocated on the same processing unit, resulting in 3 shadowed sets.

to synchronize the state of the shadows with the state of their non-faulty mains. As a result, shadows achieve forward progress with minimal overhead. Consequently, the recovery time of future failures, if any, is minimized.

In this paper, we extend the concept of leaping to the Rejuvenating Shadow model. The goal is to restore the failing main using the state of its associated shadow. Note that 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. In the case where the collocation ratio is greater than one, this is accomplished by terminating the execution of the other collocated shadows. As a result, however, only one instance of each task in the shadowed set remains. A subsequent fault in any task of this shadowed set will cause a system failure, 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, shadows collocated with the shadow of failed main need not to be terminated, but only temporarily suspended during the recovery process. Upon recovery, every main is always guaranteed to have an associated shadow. The problem, however, is that the newly spawned process will start from its initial state and may lag far behind the other processes. When the new process needs to synchronize with other processes, significant delay will be incurred as a result of the lag.

Leaping provides the solution, with minimum overhead, to eliminate the lag that a newly spawned process may incur.

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

Specifically, after a new process is started, leaping is used to synchronize the new process' state with the state of its associated shadow, after recovery.

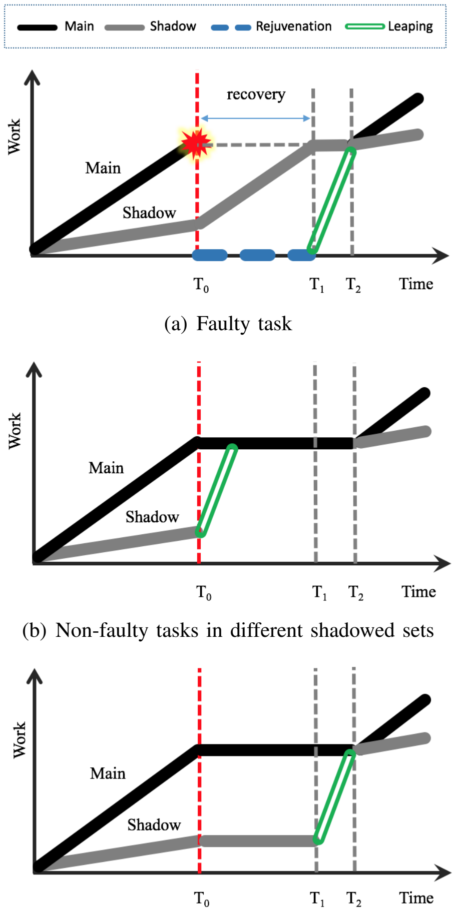
Figure 3 illustrates the failure recovery process with rejuvenation assuming that a main M_i fails at time T_0. In order for its shadow S_i to speed up and finish the recovery as soon as possible, we will temporarily suspend the execution of the other shadows that collocate with S_i. Meanwhile, the failed processor is rebooted and then used to launch a new process for M_i. When, at T_1, S_i catches up with the state of M_i before its failure, leaping can be used to advance the new process to the current state of S_i. The leaping is initiated after S_i catches up so as to avoid the need of message logging should M_i re-execute from its initial state.

Because of the failure of M_i, the other mains will be blocked when they arrive at the next synchronization point, which is assumed to be immediately after T_0. During the idle time, we can opportunistically perform a leaping and transfer state from each living main to its associated 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, since these shadows are suspended during the recovery. After the leaping finishes at T_2, all mains and shadow can 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 rebooting to finish and the new M_i to launch, and then perform the leaping. The first option requires a shadow to starting sending out messages, as well as 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.

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 [12], [28], 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. In this way, we not only can take advantage of existing MPI performance optimization that numerous researches have spent years on, but also achieve portability across all MPI implementations that conform to the MPI specification. 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) 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.

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, rsMPI will translate it into $2N+K$ processes, with the first N ranks being the mains, the next N ranks being the shadows, and the last K ranks being the coordinators. We also allow the user to specify the collocation of shadows using a configuration file. Figure 4 shows the mapping between MPI ranks and rsMPI processes for the 3 shadowed sets corresponding to Figure 2. 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.

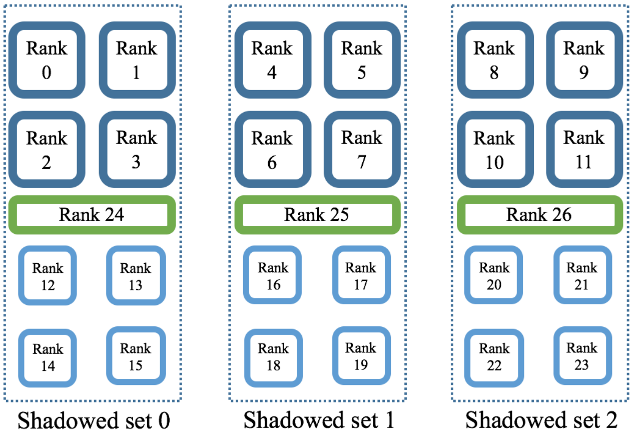


Figure 4. Mapping between MPI ranks and rsMPI processes for an example of 12 application-visible processes grouped into 3 shadowed sets. Dark blue squares are mains, light blue squares are shadows, and green rectangles are coordinators.

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 introduced to enforce determinism. For example, MPI_ANY_SOURCE may result in different message receiving orders between a main and its shadow. To deal with this, we serialize the receiving of MPI_ANY_SOURCE message by having the main first do the receiving and then use a SYNC message to forward the message source information to its shadow, which then issues a receiving from the specific source. Other operations, such as MPI_Wtime() and MPI_Probe(), are dealt with in a similar manner by forwarding the result from a main to its shadow.

E. Leaping

Checkpointing/restart requires each process to save its execution state, which can be used later to retrieve the state of the computation. Leaping is similar to saving the state in Checkpointing/restart, except that the state is transferred between a pair of main and shadow. To reduce the size of data involved in saving a process' state, we choose to implement leaping in the same way as application-level checkpointing [37]. rsMPI provides a routine for users to register any data as process state. Application developer could use domain knowledge to identify only necessary state data, or use compiler techniques to automate this [38]. Specifically, rsMPI provides the following API for process state registration:

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

For each piece of data to be registered, three parameters are needed: a pointer to the address of the data, the number of data items, and the datatype. Internally, rsMPI uses a linked list to keep track of all registered data. During leaping, the linked list is traversed to retrieve all registered data as the process state.

Coordination in leaping is simpler than in coordinated checkpointing, since leaping is always between a pair of main and shadow, while all processes need to coordinate for checkpointing. To synchronize the leaping between a main and a shadow, the coordinator in the corresponding shadowed set is involved. For example, when a main detects failure of another main and initiates a leaping, it will send a control message to its coordinator, which then uses a signal to notify the associated shadow to participate in the leaping.

Different from Checkpointing where the process state is saved, leaping directly transfers process state between a main and its shadow. Since MPI provides natural support for message passing between processes, rsMPI uses MPI messages to transfer process state. Although multiple pieces of data can be registered as a process' state, only a single message is needed to transfer the process state, 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.

A challenge in leaping lies in the need for maintaining state consistency. To make sure a pair of main and shadow stay consistent after a leaping, not only user-defined states should be transferred correctly, but also lower level states, such as program counter and message buffer, need to be updated correspondingly. Specifically, the roll forward process needs to satisfy two requirements. Firstly, after leaping the roll forward process should discard all obsolete messages before resuming normal execution. Secondly, the roll forward process should 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.

There is no straightforward way to discard obsolete messages since the message buffer is maintained by MPI runtime and not visible to rsMPI. Hence, rsMPI borrows the idea of ”determinants” used in rollback recovery to correctly discard all obsolete messages. Specifically, during normal execution, both the main and shadow record the meta data (i.e., MPI source, tag, and communicator) for all received messages in the receiving order. During leaping, the meta data at the main is transferred to the shadow, so that the shadow knows about the messages that have been received by its main but not by itself. Then the shadow combines MPI probe and MPI receive operations to remove the messages from MPI runtime buffers in the correct order.

To satisfy the second requirement (resume execution from the same state), 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 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. Then, the shadow will check the flag and compare the counter value at every possible 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 benchmark applications 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 the computations in a typical molecular dynamics application. MiniAero is an explicit unstructured finite volume code that solves the compressible Navier-Stokes equations. Both MiniFE and HPCCG are proxy applications for unstructured implicit finite element codes, but HPCCG uses MPI_ANY_SOURCE receive operations and can be used to demonstrate rsMPI's capability of handling MPI 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 checkpointing to compare with srMPI in the presence of failures. To be optimistic, we chose double in-memory checkpointing that is much more scalable than disk-based checkpointing [20]. Same as leaping in srMPI, our implementation provides an API for process state registration. This API requires the same parameters as leap_register_state(void *addr, int count, MPI_Datatype dt), 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 srMPI 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

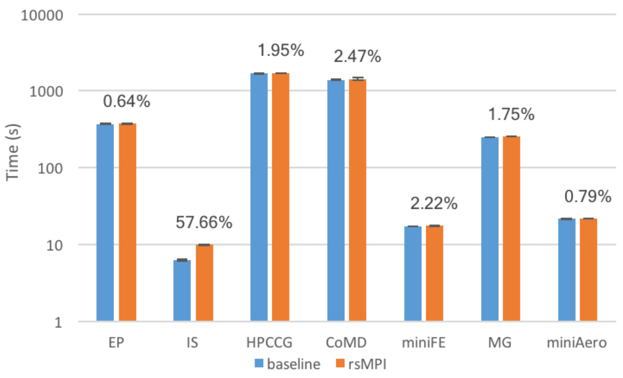


Figure 5. Comparison of execution time between baseline and rsMPI using 256 application-visible processes and collocation ratio of 2 for srMPI.

this overhead we ran each benchmark application linked to srMPI multiple times and compared the average execution time with the baseline, where each application runs with original OpenMPI.

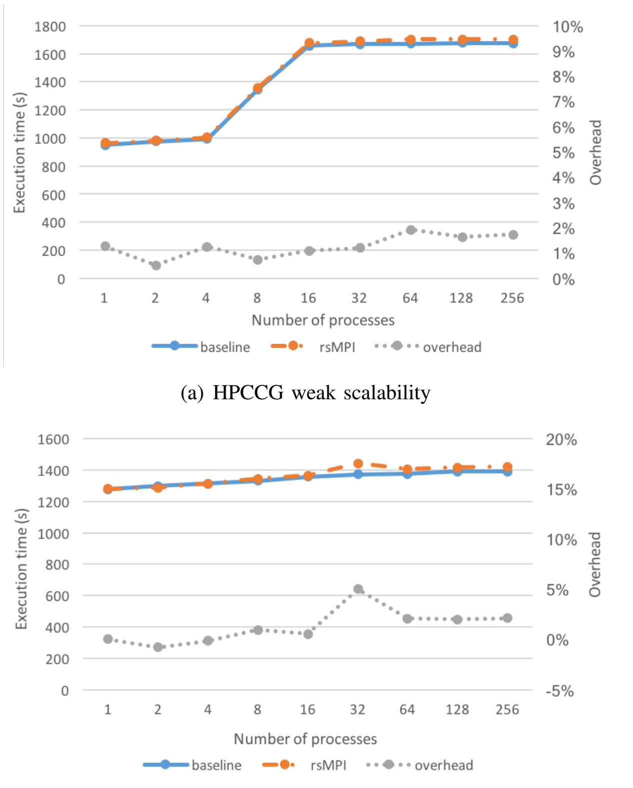
Figure 5 shows the comparison of the execution time between baseline and srMPI for the 7 applications in the absence of faults. All the experiments are conducted with 256 application-visible processes. That is, the baseline always uses 256 MPI ranks compiled with the unmodified OpenMPI library, while rsMPI uses 256 mains together with 256 shadows which are invisible to the application. Each result shows the average execution time of 5 runs, the standard deviation, and srMPI's runtime overhead. The baseline execution time varies from seconds to half an hour, so we plotted the time in log-scale.

From the figure we can see that srMPI 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 srMPI's runtime overheads are modest for scalable HPC applications that exhibit a fair communication-to-computation ratio.

B. *Scalability*

In addition to measuring the runtime overhead at fixed application-visible process count, we also varied the number of processes and 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 and perform 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 6.



(a) HPCCG weak scalability

(b) CoMD weak scalability

Figure 6. Scalability test for number of processes from 1 to 256. Collocation ratio is 2 for srMPI.

Comparing the baseline execution time between Figure 6(a) and Figure 6(b), it is obvious that HPCCG and CoMD have different weak scaling characteristics. Keeping the same problem size per process, the execution time for CoMD increases by 8.9% from 1 process to 256 processes, while 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. Note that each node in the cluster has 20 cores and we always use all the cores of a node before adding another node. Therefore, it is very likely that the node level contention leads to the substantial increase in execution time for HPCCG. The results from 16 to 256 processes show that both HPCCG and CoMD are weak scaling applications.

Similar to the results of the previous section, the runtime overhead for srMPI is modest. The maximum overhead observed is 5.0% when running CoMD with 32 processes. Excluding this case, the overhead is always below 2.1%. 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.

Failure detection is beyond the scope of srMPI. We assume that the underlying hardware platform has a Reliability, Availability and Serviceability (RAS) system that provides this functionality. In our test system, we emulate the RAS functionality by associating a signal handler with every main and shadow. 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 of the failure. Similar to ULFM, a process in srMPI can only detect failure when it posts an MPI receive operation. In a srMPI receive, a process checks for failure messages before it performs the actual receive operation.

The first step was to test the effectiveness of leaping. For each application, we identified the process state and register them with rsMPI. Figure 7 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, at an increment of 10%. The execution time is normalized to that of the fault-free baseline. The blue solid line and red dashed line represent srMPI with collocation ratio of 2 and 4, respectively. For simplicity, they are referred to as srMPI_2 and srMPI_4 in the following text.

As shown in Figure 7, srMPI's execution time increases with the failure occurrence time, regardless of the collocation ratio. The reason is that failure recovery time for srMPI is proportional to the amount of divergence between mains and shadows, and the divergence 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, srMPI_2 can recover faster than srMPI_4, and therefore achieves better execution time.

The results in Figure 7 show that recovery time in srMPI is proportional to the divergence between a main and its shadow, which confirms that srMPI is well 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. As the number of failure increases, the interval between failures decreases, thereby reducing the recovery time per failure.

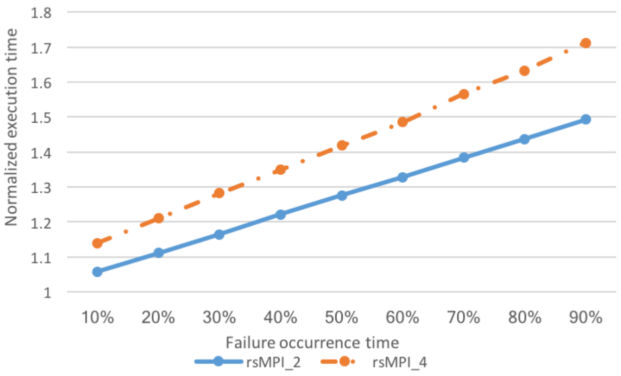


Figure 7. Execution time of HPCCG with a single failure injected at various time.

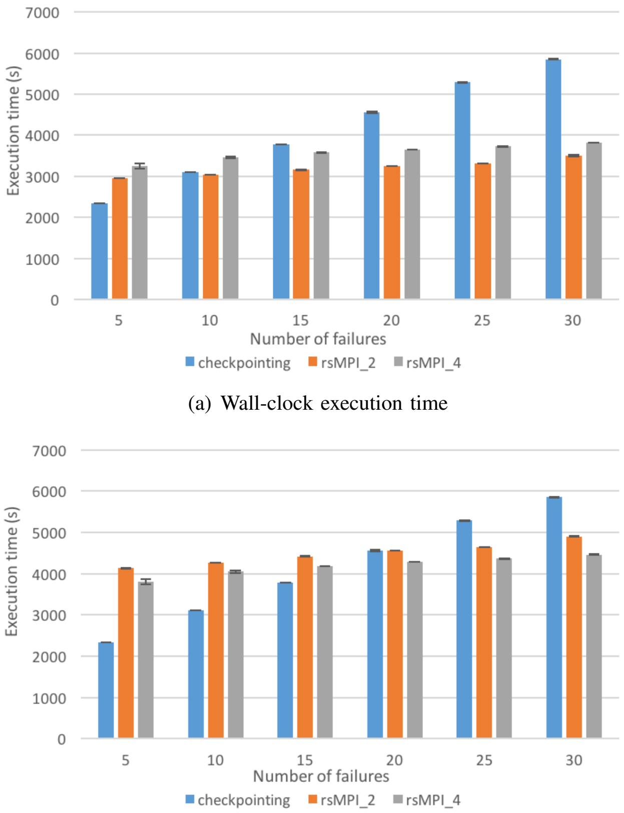
To demonstrate the above analysis, we compare rsMPI with checkpointing under various number of failures. To run the same number of application-visible processes, rsMPI needs more nodes than checkpointing to host the shadow processes. For fairness, we take into account the extra hardware cost by defining 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 srMPI_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 srMPI_4 as $1.25 \times \frac{1.88}{2} = 1.17$.

In this comparative analysis, we set the checkpointing interval to $0.1T$, where T is the total execution time. To emulate failures, for both checkpointing and srMPI, 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 256 processors and executing for 1700 seconds, injecting 10 faults corresponds to a processor's MTBF of 12 hours. However, for a system of $64,000$ processors executing over 4 hours, injecting 10 faults corresponds to a processor's MTBF of 3 years.

Figure 8 compares checkpointing and srMPI (srMPI_2 and srMPI_4), based on both wall-clock and weighted execution time. Without taking into consideration the hardware overhead, Figure 8(a) shows that, when the number of failures is small (e.g., 5 failures), checkpointing slightly outperforms srMPI, with respect to wall-clock execution time. As the number of failures increases, however, srMPI achieves significantly higher performance than checkpointing. For



(a) Wall-clock execution time

(b) Weighted execution time

Figure 8. Comparison between checkpointing and srMPI with various number of failures injected to HPCCG. 256 application-visible processes, 10% checkpointing interval.

example, when the number of failures is 20, srMPI_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, srMPI_4 reduces the execution time by 19.7% and 34.8%, when the number of failures are 20 and 30, respectively.

Careful analysis of Figure 8(a) reveals that, as the number of failures increases, checkpointing and srMPI exhibit different performance behaviors with respect to wall-clock execution time. As expected, the execution time for checkpointing increases proportionally with the number of failures. For srMPI, however, the increase is sub-linear. This is due to fact that as more failures occur, the interval between failures is reduced, and as a result, the recovery time per failure is also reduced. Although not shown in Figure 8(a), rebooting time will eventually dominate the recovery time when more failures occur, resulting in a linear increase in execution time for srMPI. This increase, however, occurs at a significantly slower rate than checkpointing.

Considering both execution time and hardware overhead, Figure 8(b) compares the weighted execution time of checkpointing and srMIP. As expected, checkpointing is better than srMPI when the number of failures is small (e.g., 5 failures). When the number of failures increases, however, checkpointing loses its advantage, as its wall-clock execution time increases significantly in comparison to srMPI. When the number of failures reaches 30, for example, the results show that srMPI_2 and srMPI_4 are 19.3% and 31.3% more efficient than checkpointing, respectively. Note that, when comparing srMPI_2 and srMPI_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, referred to as Rejuvenating Shadows. Either by using DVFS or collocating shadows to reduce the execution rate of the shadows, Rejuvenating Shadows enables a parameterized tradeoff between time and hardware redundancy. The proposed solution differs from traditional approaches in the type of faults it handles and the fault tolerance protocol it uses. This property leads to efficient and "tunable" resilience that takes into consideration the time-to-solution, the power constraints and the resilience level of the underlying application.

We discussed the design decisions and presented the details of a full-feature implementation of Rejuvenating Shadows in MPI. Careful design guarantees that Rejuvenating Shadows always achieves forward progress in the presence of failures, maintains consistent level of resilience regardless of the number of failures, 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 the increase in time between failures. 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. We will also explore how this mechanism will be implemented in srMPI.

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, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, 2008.

[2] 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.

[3] S. Kalaiselvi and V. Rajaraman, "A survey of checkpointing algorithms for parallel and distributed computers," *Sadhana*, vol. 25, no. 5, pp. 489–510, 2000.

[4] 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.

[5] S. Gao, B. He, and J. Xu, "Real-time in-memory checkpointing for future hybrid memory systems," in *Proceedings of the 29th International Conference on Supercomputing*. New York, NY, USA: ACM, 2015, pp. 263–272.

[6] 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.

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

[8] X. Cui, T. Znati, and R. Melhem, "Adaptive and power-aware resilience for extreme-scale computing," in *16th IEEE International Conference on Scalable Computing and Communications*, July 18-21 2016.

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

[10] S. Chakravorty and et. al., "Proactive fault tolerance in mpi applications via task migration," in *International Conference on High-Performance Computing*, 2006, pp. 485–496.

[11] 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.

[12] 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.

[13] K. Chandy and C. Ramamoorthy, "Rollback and recovery strategies for computer programs," *Computers, IEEE Transactions on*, vol. C-21, no. 6, pp. 546–556, June 1972.

[14] 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.

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

[16] B. Randell, "System structure for software fault tolerance," in *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM, 1975.

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

[18] J. Plank and K. Li, "Faster checkpointing with n+1 parity," in *Fault-Tolerant Computing*, June 1994, pp. 288–297.

[19] E. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit," *TC*, vol. 41, pp. 526–531, 1992.

[20] 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.

[21] X. Ni, E. Meneses, and L. V. Kalé, "Hiding checkpoint overhead in hpc applications with a semi-blocking algorithm," in *Cluster Computing*. IEEE, 2012, pp. 364–372.

[22] 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.

[23] D. Hakkarinen and Z. Chen, "Multilevel diskless checkpointing," *Computers, IEEE Transactions on*, vol. 62, no. 4, pp. 772–783, April 2013.

[24] 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.

[25] F. Cappello, "Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 212–226, 2009.

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

[27] 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.

[28] 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.

[29] S. Sankaran, J. M. Squyres, and et. al., "The lam/mpi checkpoint/restart framework: System-initiated checkpointing," *The International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, 2005.

[30] 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.

[31] 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.

[32] J. Hursey, J. M. Squyres, and et. al., "The design and implementation of checkpoint/restart process fault tolerance for open mpi," in *IPDPS'07*, pp. 1–8.

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

[34] 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.

[35] 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.

[36] 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.

[37] A. Beguelin, E. Seligman, and P. Stephan, "Application level fault tolerance in heterogeneous networks of workstations," *Journal of Parallel and Distributed Computing*, vol. 43, pp. 147–155, 1997.

[38] G. Bronevetsky, D. Marques, K. Pingali, S. McKee, and R. Rugina, "Compiler-enhanced incremental checkpointing for openmp applications," in *International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–12.