A Composable Runtime Recovery Policy Framework Supporting Resilient HPC Applications

Joshua Hursey*, Andrew Lumsdaine*
* Open Systems Laboratory, Indiana University, Bloomington, IN USA 47405
Email: {jjhursey,lums}@osl.iu.edu

Abstract—**An HPC application must be resilient to sustain itself in the event of process loss due to the high probability of hardware failure on modern HPC systems. These applications rely on resilient runtime environments to provide various resiliency strategies to sustain the application across failures. This paper presents the ErrMgr recovery policy framework providing applications the ability to compose a set of policies at runtime. This framework is implemented in the Open MPI runtime environment and currently includes three policy options: run-through stabilization, automatic process recovery, and preemptive process migration. The former option supports continuing research into fault tolerant MPI semantics while the latter two provide transparent, checkpoint/restart syle recovery. We discuss the impact on recovery policy performance for various stable storage strategies including staging, caching, and compression. Results indicate that some stable storage configurations designed to provide low overhead during failure-free execution often negatively impact recovery performance.**

I. Introduction

With the increasing size of High Performance Computing (HPC) systems, process loss due to hardware failure is becoming a limiting factor to application scalability. Resilient applications (i.e., applications that can continue to run despite process failure) depend on resilient runtime and communication environments to sustain the remaining live processes across the failure of a peer process. In typical HPC environments, communication is provided by an Message Passing Interface (MPI) implementation [1] and process launch, monitoring, and cleanup is provided either by the corresponding MPI runtime or by a system-provided runtime. Therefore a resilient MPI implementation depends on a stable and recoverable runtime environment that can sustain both the MPI implementation and the application. Unfortunately, resilient runtime environments and resilient MPI implementations are uncommon today. As a result, applications - even applications that are designed to be resilient - are forcibly removed from the system upon process failure.

This paper presents recent advancements in the Open MPI project providing the foundational components of a resilient runtime environment to support resilient MPI applications. Primary among these components is the Error Management and Recovery Policy (ErrMgr) framework in the Open MPI Runtime Environment (ORTE). The ErrMgr framework provides applications with a variety of individual recovery policy options that can be composed at runtime to form a tailored recovery policy solution to best support the resilient application. We discuss three currently available recovery policy options: *run-through stabilization*, *automatic process recovery*, and *preemptive process migration*. Run-through stabilization supports continuing research into fault tolerant MPI semantics allowing the application to continue running without requiring the recovery of lost processes. Automatic process recovery and preemptive process migration are transparent Checkpoint/Restart (C/R) supported recovery policies.

Although various individual recovery policies have been explored in previous research, the ErrMgr framework unites these techniques into a novel composable infrastructure. This unique framework allows policies to be composed in a customizable and interdependent manner so applications can choose from a wide range of recovery policy options instead of just one, as with most previous contributions. Additionally, the composable nature of this framework reduces the recovery policy development burden by supporting and encouraging code reuse. Unique to this framework is the ability of one policy to failover onto another policy. For example, this framework allows an application to compose a recovery policy from individual light-weight and a heavy-weight policies. Combining the two policies allows the light-weight approach to be used for small groups of failures, and then failover to the heavy-weight approach for large groups of failures.

A logical stable storage device provides recovery policies with a reliable location to place recovery information during normal execution that can be later used during recovery. The implementation of the stable storage device can have considerable impact on both the failure-free and recovery performance overhead of a recovery policy. Accordingly, this paper develops the Stable Storage (SStore) framework and analyses the performance tradeoffs for various stable storage strategies including staging, caching, and compression. Results indicate that some stable storage configurations designed to provide low overhead during failure-free execution often negatively impact recovery performance. Additionally, for some applications the use of compression and caching can greatly improve both the failure-free and recovery performance.

II. Related Work

Resilient applications depend on a resilient communication and runtime environments to enable continued operation across a process loss. Algorithm Based Fault Tolerance (ABFT) [2] and Natural Fault Tolerance [3], [4] techniques focus on modifying, or choosing alternative core algorithms to integrate fault tolerance techniques. Such algorithms often either contain support for diskless checkpointing [5], [6], [7], [8], [9] or exploit the ability of an algorithm to compute previous results from current data [10], [11], [12]. Such techniques are encouraging examples of how applications can adapt to become more resilient to process loss at an algorithmic level.

A. Fault Tolerance and MPI

To support resilient applications the underlying message passing environment must also be resilient to process failures. As the *de facto* standard message passing environment, MPI is often looked to for resilience support [1]. Unfortunately, the MPI standard does not specify the semantics regarding how an MPI implementation should behave after a process failure. Individual, high-quality implementations are left to define their own behaviors [13]. Most MPI implementations do not support such additional semantics for fear of added implementation complexity and the potential performance impact.

The research is limited with regards to MPI implementations providing the ability to continue operating across process failure. The most notable project that investigated such an implementation is the FT-MPI project [14], [15], [16]. FT-MPI is an MPI-1 implementation that extended the MPI communicator states and modified the MPI communicator construction functions. Resilient MPI applications use these extensions to stabilize MPI communicators and, optionally, recover failed processes by relaunching them from the original binary and rejoining them into the MPI communicator.

The Heterogeneous Adaptable Reconfigurable Networked SyStems (HARNESS) runtime environment supports the FT-MPI project by providing process fault detection and global recovery operations [17], [18]. The recovery procedure in the HARNESS project relies on a distributed, resilient election algorithm that chooses a leader that in turn determines the view of the system after a failure. In the Open MPI Runtime Environment (ORTE) we simplify this algorithm by requiring that the Head Node Process (HNP) survive all anticipated failures in the HPC system; thus the HNP becomes the recovery leader. This simplification reduces both the algorithmic complexity of the recovery protocol and the time required to stabilize the runtime environment after process loss. Similar to the HARNESS project, ORTE uses a fault notification service to distribute fault events throughout the runtime environment.

The ABARIS project [19] provided a fault tolerance framework for MPI implementations allowing a user to select from a variety of individual recovery policies at runtime. They supported the ability to ignore process loss, restart computation from the last checkpoint, and migrate processes using coordinated checkpointing. The ErrMgr framework in the ORTE allows for the dynamic selection of the recovery policy combining the dynamic recovery policy concept from ABARIS and the infrastructure lessons from the FT-MPI/HARNESS projects. Extending from the previous research, this framework allows recovery policy options to be both individually selectable and composable providing the application with a wider range of policy options.

B. Fault Recovery

Once a process failure has been detected and the runtime has stabilized, then the software stack can consider recovery options. The recovery policy may be as straightforward as removing references to the failed processes and continuing execution, often called running through failure or running while crippled. Alternatively, the application or runtime environment may choose to promote a replicated process, or restart the failed processes from the original binary or from a previously established checkpoint. As discussed in the ABARIS project [19], providing the application and runtime environment with a variety of recovery options allows it to choose the option best suited for the application given the state of the computing environment.

To recover from process loss, replication and rollback recovery policies return one or more processes to a previously established state [20]. Message logging and Checkpoint/Restart (C/R) represent the primary categories of rollback recovery techniques.

Process replication often either involves primary backup or active replica update techniques [21], [22]. Only a few MPI implementations have experimented with replication, often focusing on implementations transparent to the application. Notable implementations include MPI/FT [23], P2P-MPI [24], VolpexMPI [25], and rMPI [26]. Each implementation focuses on providing only a replication-based recovery policy. In this paper, we present an infrastructure that is able to support replication research as one of many composable recovery policies available in the system.

Message logging rollback recovery techniques restart the failed processes from either the beginning of execution or, when combined with uncoordinated C/R, from a previously established checkpoint. The failed process is then allowed to rely on a log of external events to guide the process execution back into a state consistent with the rest of the processes in the job [27], [28]. Notable implementations of various message logging protocols include Manetho [29], Egida [30], MPL* [31], RADICMPI [32] and MPICH-V [33], [34]. The MPICH-V project contributed a variety of implementations each investigating various message logging and C/R techniques [35]. The extensible infrastructure in Open MPI, including the ErrMgr framework, supports concurrent research into both C/R and message logging techniques. Early work on the integration of message logging techniques into Open MPI are presented in [36].

Checkpoint/Restart (C/R) rollback recovery is a technique used to reduce the amount of computation lost to process failure by restoring processes from a previously established point in the computation. Applications establish checkpoints during failure-free operation by writing them to a stable storage medium. A *stable storage* medium is a logical device that survives the maximum number of anticipated failures in the system [20]. Usually this is represented as a centralized file server able to recover all processes in a job, though peer based techniques can be used for partial job recovery.

When discussing the cost of checkpointing techniques, we must consider the effect on both the application and the system. The additional execution time required by the application as a result of taking a single checkpoint is called the *Checkpoint Overhead* [37], [38]. The *Checkpoint Latency* is the time required to create and establish a checkpoint on stable storage [37], [38]. If the application is suspended until the checkpoint is established then the checkpoint overhead is equal to the latency, which is the case for many direct to central storage techniques.

There are three broad categories of C/R implementations with varying degrees of complexity and consistency: uncoordinated, message induced, and coordinated [20]. Uncoordinated C/R protocols do not coordinate among processes when creating checkpoints, but instead apply algorithms to determine the set of consistent checkpoints on restart [39], [40], [41], [42]. Message induced (or communication induced) C/R protocols take both independent and forced checkpoints dependent upon state information piggybacked on messages in the system [43], [44], [45]. Coordinated C/R protocols require that all processes take a checkpoint at logically the same time [46]. Coordinated C/R protocols create a consistent, and often a strongly consistent or quiescent state, of the network before checkpoints are created to account for messages in-flight at the time of the checkpoint [41], [47]. Coordinated C/R protocols are a popular implementation choice due to its relative simplicity and strong consistency characteristics.

Although C/R is not part of the MPI standard, it is often provided as a (semi-)transparent service by MPI implementations including LAM/MPI [48], M^3 [49], CoCheck [50], MPICH-V [35], MVAPICH [51], and Open MPI [52], [53]. Most implementations provide support for checkpointing an MPI application and restarting it from a new job submission. Few implementations provide automatic, in-place recovery of the MPI application. In this paper we will present an implementation of automatic, in-place recovery using a coordinated C/R technique similar to that presented in [54].

Process migration moves a process from a *source* machine to a *destination* machine during execution for either fault tolerance or load balancing purposes. When process migration is used for fault tolerance, it is important that the technique used does not leave *residual dependencies* on the source machine. A *residual dependency* is a dependency on the source machine after the process has migrated to the destination machine. There are a variety of techniques for transferring a process from the source to the destination machine [55]. This paper presents an implementation of an *eager copy* process migration protocol where the computation is paused while the entire process image is transferred and restarted on the destination machine. Other work has investigated applying a *pre-copying* (a.k.a., syphoning or live migration) process migration protocol to reduce the migration overhead experienced by the application [56], [57].

C. Stable Storage

A logical stable storage device provides recovery policies with a reliable location to place recovery information during normal execution that is used during recovery. Due to the large amount of recovery data typically involved with C/R based recovery policies, the performance of stable storage accounts for the majority of the checkpoint latency.

Checkpoint compression is one way to reduce the checkpoint latency; reducing the amount of checkpoint data that is pushed to stable storage reduces the I/O required to store the checkpoint [58], [59], [60], [61]. Most research using compression employs in-line compression while writing the checkpoint [59]. Only a few researchers have looked at compression as part of a larger staging process at the stable storage level, as we will in this paper. Though compression may reduce the size of the checkpoint, it only improves the checkpoint latency "...if the speed of compression is faster than the speed of disk writes, and if the checkpoint is significantly compressed." [59].

Another technique used to reduce checkpoint latency is to use peer-based storage and/or multi-level staging techniques. Peer-based stable storage techniques remove the dependency on central storage by using node-local storage on peer systems to reduce the stress on the central file system [62]. The SCR [63] and stdchk [64] projects have shown considerable performance and system reliability improvements by moving to a peer-based stable storage environment for C/R. Staging and staggering techniques reduce the stress on the central storage device by managing the concurrent writes in response to the currently available bandwidth and system architecture [65], [66], [67], [68]. Caching files in node-local storage, reduces the stress on stable storage improving recovery performance [69], [70]. This paper explores the effect of node-local staging, caching, and compression techniques in the stable storage pipeline on recovery policy options.

III. Open MPI Architecture

Open MPI [71] is an open source, high performance, MPI-2 compliant implementation of the MPI standard [1]. Open MPI is built upon the Open MPI Runtime Environment (ORTE) project that provides runtime services customized at build and runtime to support Open MPI applications running on HPC systems.

A. Modular Component Architecture

Open MPI is designed around the Modular Component Architecture (MCA) [72]. The MCA provides a set of component *frameworks* to which a variety of point-to-point, collective, and other MPI and runtime-related algorithms can be implemented. The MCA allows for runtime selection of the best set of *components* (implementations of the framework interfaces) to properly support an MPI application in execution.

MCA frameworks in Open MPI are divided into three distinct layers: Open Portable Access Layer (OPAL), Open MPI Runtime Environment (ORTE), and Open MPI (OMPI). OPAL is composed of frameworks that are concerned with portability

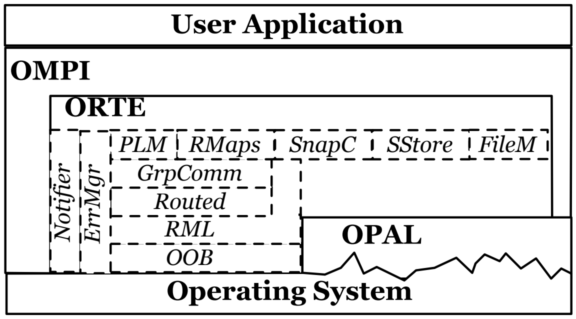


Fig. 1. Open MPI MCA layers including some ORTE layer frameworks.

across various operating systems and system configurations along with various software development support utilities (e.g., linked lists). ORTE is composed of frameworks that are concerned with the launching, monitoring, and cleaning up of processes in the HPC environment. The OMPI layer is composed of frameworks to support the MPI interfaces exposed to the application layer. Figure 1 illustrates this layering.

Many frameworks are combined to form the ORTE layer. The communication frameworks in ORTE combine to provide a resilient, scalable out-of-band communication path for the runtime environment. The Out-Of-Band (OOB) framework provides low-level interconnect support (currently TCP/IP based). The Runtime Messaging Layer (RML) framework provides a higher-level point-to-point communication interface including basic datatype support. The Group Communication (GrpComm) framework provides group communication, collective-like operations among various processes active in the runtime environment. The Routing Table (Routed) framework provides a scalable routing topology for the GrpComm framework.

The Process Lifecycle Management (PLM) framework is responsible for launching, monitoring, and terminating processes in the runtime environment. The ORTE Daemon Local Launch Subsystem (ODLS) framework provides the same services as the PLM, but on a local node level. The Resource Mapping Subsystem (RMapS) framework is responsible for mapping a set of processes onto the currently available resources.

The ErrMgr framework is accessible throughout the ORTE and OMPI layers. This framework provides a central reporting location for detected or suspected process, or communication failure. In this paper, we have extended the ErrMgr to include support for a composable set of recovery policies, discussed in Section IV. The Notifier framework works with the ErrMgr framework and provides an interface for processes to send and receive reports on abnormal events in the system, including process failure or communication loss.

B. Checkpoint/Restart Infrastructure

The checkpoint/restart infrastructure integrated into Open MPI [52], [53] is defined by process and job levels of control. These two levels of control work in concert to create stable job-level checkpoints (called *global snapshots*) from each individual process-level checkpoint (called *local snapshots*).

Job-level control is composed of two original frameworks and one new framework. The Snapshot Coordination (SnapC) and File Management (FileM) frameworks are part of the original design. This paper introduces the SStore framework into the C/R infrastructure in Open MPI. The SnapC framework controls the checkpoint life-cycle, from distributing the checkpoint request to all processes, to monitoring their progress, to synchronizing the final snapshots to stable storage. The FileM framework focuses on the movements of individual local snapshots to and from storage devices possibly across file system and node visibility boundaries.

The new SStore framework is a stable storage framework that abstracts the SnapC framework from the underlying mechanism of how snapshots are established on the stable storage medium. A global snapshot is said to be *established* on stable storage when the snapshot is able to be used to recover the application from stable storage up to the number of anticipated failures in the system. For example, in a centralized stable storage environment (e.g., SAN) that is able to handle the loss of the entire job, the global snapshot is established when all of the local snapshots have successfully been written to the centralized stable storage environment. In a peer-based, node-local stable storage medium that replicates local snapshots among *N* peers, the global snapshot is established upon verified completion of the replication stage.

The SStore framework currently supports two components: central and stage. The central component stores local snapshots directly to the logically centralized stable storage device (e.g., SAN, parallel file system). The application is stalled until all of the local snapshots have been established to the storage device. The stage component uses node-local storage (e.g., local disk, RAM disk) as a staging location for moving local snapshots back to the logically centralized stable storage device. The application is allowed to continue execution once all of the local snapshot have been written to the node-local storage devices. The ORTE daemon then concurrently moves the local snapshots back to the logically centralized stable storage device, overlapping snapshot movement with application execution often improving the checkpoint overhead.

To improve the performance of C/R based automatic recovery and process migration, discussed in Section IV, we have also implemented local snapshot caching and compression in the stage component. Local snapshot caching requires processes to keep a copy of the last *N* local snapshots (default 2) on node-local storage. This improves the performance of automatic recovery since all of the non-failed processes are not moved in the system, and can use the locally cached copy of the snapshot instead of going to the stable storage device.

The compression feature adds one more step in the staging pipeline. After the application writes the local snapshot to node-local storage it is able to continue execution. Once all of the process on the node have finished checkpointing the ORTE daemon compresses the local snapshot using one of a variety of compression utilities (e.g., bzip, gzip, zlib). After the compression stage, the compressed local snapshot is moved to the logically centralized stable storage device. Moving forward with this framework, we plan to support other checkpoint-specific stable storage file systems (e.g., stdchk [64]).

IV. Error Management and Recovery Policy Framework

The Error Management and Recovery Policy (ErrMgr) framework was originally designed to provide Open MPI processes and daemons a stable interface to report process and communication failure so that the proper job abort procedure could be taken. This paper extends the ErrMgr framework to support a variety of recovery policies. However, since recovery is a non-standard feature this framework preserves, by default, the termination of the job upon process failure. The ErrMgr framework is a *composite* framework which allows more than one recovery policy component to be active at the same time. The ability to compose a recovery policy from multiple ErrMgr components enables the application to tailor the recovery solution to their application and system requirements. Additionally, the composable nature of this framework reduces the recovery policy development burden by supporting and encouraging code reuse.

In the composite ErrMgr framework components stack themselves in priority order and pass a status vector between active components. To facilitate cross-component collaboration, the status vector allows one component to indicate to the next component any action it may have taken as a result of an anticipated or detected process fault. For example, this framework allows for two process recovery components to work together to provide a light-weight and heavy-weight recovery policy. The light-weight recovery policy may handle small groups of concurrent failures, and then defer to the heavy-weight recovery policy that is able to handle a larger groups of concurrent failures.

An MPI process detects process failure by way of communication timeouts or errors and calls the `process_fault()` interface of the ErrMgr to report the fault. The ErrMgr works in concert with the Notifier framework to distribute the fault event throughout the runtime environment. These frameworks are able to recognize and handle multiple reports of the same failure. The ErrMgr in the MPI process is able to stabilize locally while waiting for a global ruling on how to proceed by the same ErrMgr component active in the HNP, serving as the global leader.

Fault detection inside the ORTE environment is currently communication and, optionally, heartbeat based. If one process is unable to communicate with a peer it declares that process as failed and notifies the ErrMgr framework for recovery. If there is no recovery policy defined by the ErrMgr component then the job aborts. The local ErrMgr component can receive fault notification events from external fault detection/prediction sources through the Notifier framework (including support for CIFTS FTB [73]). Though the current components decide globally how to recover from a failure, the framework interface is general enough to support a more localized or distributed recovery policy and notification implementations.

A. Runtime Stabilization

The default behavior of the ErrMgr framework is to terminate the entire MPI job upon the detection of process failure. The sustain ErrMgr component provides the ability to choose to run-through the process failure by, instead of terminating the job, simply stabilizing the runtime environment and continuing execution.

The Routed, GrpComm, and RML frameworks have been extended to include interfaces for the ErrMgr to notify them of process failure for framework- and component-level stabilization. The individual components are then responsible for recovering from the failure. If the component cannot recover from the loss, indicating an unrecoverable runtime, it can return an error value which will terminate the job.

Since MPI processes always route ORTE layer out-of-band communication through their local daemon, the daemon contains the bulk of the recovery logic for rerouting, delaying or dropping communication around recovery from process loss reducing the per process recovery overhead. Dropped communication will return as a communication error to the sending or receiving process.

As part of the stabilization an *up-call* is made available to the OMPI layer from the ErrMgr component to indicate that a process has been lost and that the OMPI layer stabilization and recovery procedures should be activated. Stabilization at the OMPI layer often includes, but is certainly not limited to, flushing communication buffers involving the failed peer(s), activating error handlers and error reporting paths back to the application, and stabilizing communicator and other opaque MPI data structures. The semantics for how MPI functions behave across process failure is still an active area of research and is currently under consideration by the Fault Tolerance Working Group in the MPI Forum. The ErrMgr framework was designed to support this effort by providing well-defined stabilization procedures for the runtime environment. The OMPI layer builds upon the stabilized runtime to support research into MPI fault tolerance semantics.

B. Automatic Recovery

Instead of just running through a failure, the application may choose to recover from the loss by automatically recovering from the last established global snapshot of the application by enabling the autor ErrMgr component. The autor component is notified of process failure via the `process_fault()` interface. By default, the autor component places a failed process on a different node than the one it resided on before the failure. This avoids repeated failures due to node specific component failure that may have caused the original process failure.

Since this implementation of automatic recovery is based in a coordinated C/R implementation, all processes must be restarted from a previously established global snapshot in order to provide a consistent state on recovery. Depending on the SStore component active in the application, the local snapshots may be pulled directly from logically centralized stable storage or staged to node-local storage before restart. If there is a locally cached copy of the local snapshot, a process can improve recovery time by using it to reduce the performance bottleneck on central storage.

C. Process Migration

When a process or node failure is anticipated, the ErrMgr components are notified via the `predicted_fault()` interface. This interface provides the ErrMgr components with a list of anticipated process and node faults supported by an external fault prediction service or system administrator. The external fault prediction service can express with each prediction both an assurance level, and an estimated time bounds. The estimated time bounds allow the process migration ErrMgr component, crmig, to tell if it has enough time to migrate the processes or if it should defer to the autor component for failure recovery taking advantage of the composible nature of the ErrMgr framework. The `ompi-migrate` tool provides a command line interface for end users to request a process migration within a running MPI application.

Additionally, this command line interface allows an external service the ability to provide a suggested list of target nodes to use in replacement for the affected nodes. The RMapS framework uses the `suggest_map_targets()` interface to allow the ErrMgr components the opportunity to suggest nodes for each recovering process. The ability to suggest destination nodes allows a system administrator to move processes from a set of nodes going down for maintenance to a set of nodes dedicated to the process for the duration of the maintenance activity. This also allows users to experiment with using process migration for load balancing since they can also specify specific process ranks instead of just nodes for migration.

The crmig ErrMgr component implements an *eager copy* process migration protocol without residual dependencies based on the coordinated C/R infrastructure in Open MPI. Only the migrating processes are checkpointed, all other processes are paused after the checkpoint coordination. Once the migrating processes have been successfully restarted on their replacement nodes, the non-migrating processes are released and computation is allowed to resume.

If an unexpected failure occurs during process migration, then the migration is canceled and the autor component is allowed to recover the job from the last fully established global snapshot. If the autor component is not active, and no other recovery policy is enabled, then the job will terminate.

V. Results

This section presents an analysis of the affect of stable storage configuration on recovery policy performance. In these tests we are using the Odin cluster at Indiana University. Odin is an 128 node, Dual AMD 2.0 GHz Dual-Core Opteron machine with 4 GB of memory per compute node. Compute nodes are connected with gigabit Ethernet and InfiniBand. It is running RedHat Linux 2.6.18-53, BLCR 0.8.1, and a modified version of Open MPI. In our analysis we ran the benchmarks and applications using two of the four cores on each of the compute nodes, as to alleviate some of the memory contention on the nodes. In this experiment the local snapshots were compressed (using `gzip`) on the same node as the application process. Future studies will assess the benefits of using an intermediary node to assist in the compression process.

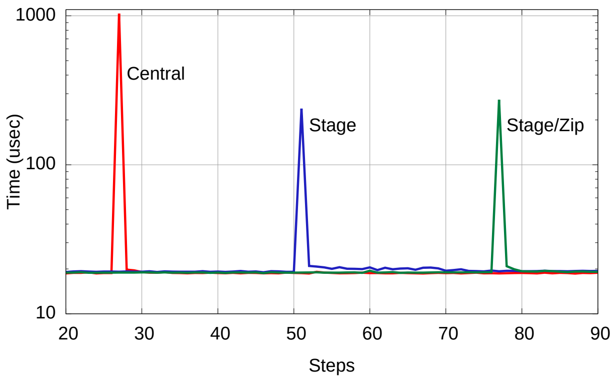


Fig. 2. Performance affect of SStore components on application latency.

For the automatic recovery, process migration, and stable storage overhead discussions we will primarily be using a `noop` program that can be configured with a variable sized random matrix to emulate various process sizes. In these experiments, the `noop` program was given a 10MB random matrix per process.

Additionally, we assessed the impact of SStore configurations, including compression, on the Parallel Ocean Model (POP) [74], Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [75], and High-Performance Linpack (HPL) [76] HPC software packages. In our analysis of POP, we used the *bench01.tacc* benchmark over 5 days of simulation. In our analysis of LAMMPS, we used a scaled version of the metal benchmark *eam* involving 11 million atoms over 400 steps. In our analysis of HPL, we used a variety of problem sizes: 40,000 for 64 process, 55,000 for 128 processes, and 70,000 for 192 processes.

A. SStore Overhead

In this section we assess checkpoint overhead by measuring the impact of various stable storage strategies on application performance. We used a continuous latency test that measures the time taken for an 8KB message to travel around a ring of 64 processes. With this test we are able to both illustrate the impact of the stable storage strategy on the application, and to measure the impact of the checkpoint overhead. Figure 2 illustrates the impact of using the following SStore components:

- central
- stage
- stage with compression enabled

The central SStore component adds approximately 1018 microseconds of half roundtrip point-to-point latency overhead across the checkpoint operation which takes 11.1 seconds. The stage SStore component adds approximately 244 microseconds of overhead spread over 21 steps of computation. The checkpoint took 8.3 seconds, spending only 0.8 seconds establishing the local snapshot on the local disk, and 7.4 seconds staging the local snapshots back to stable storage. The compression enabled stage SStore component adds approximately 257 microseconds of overhead spread over 4 steps of computation. The checkpoint took 4.1 seconds, spending only 0.7 seconds establishing the local snapshot, and 2.7 seconds staging the local snapshots back to stable storage.

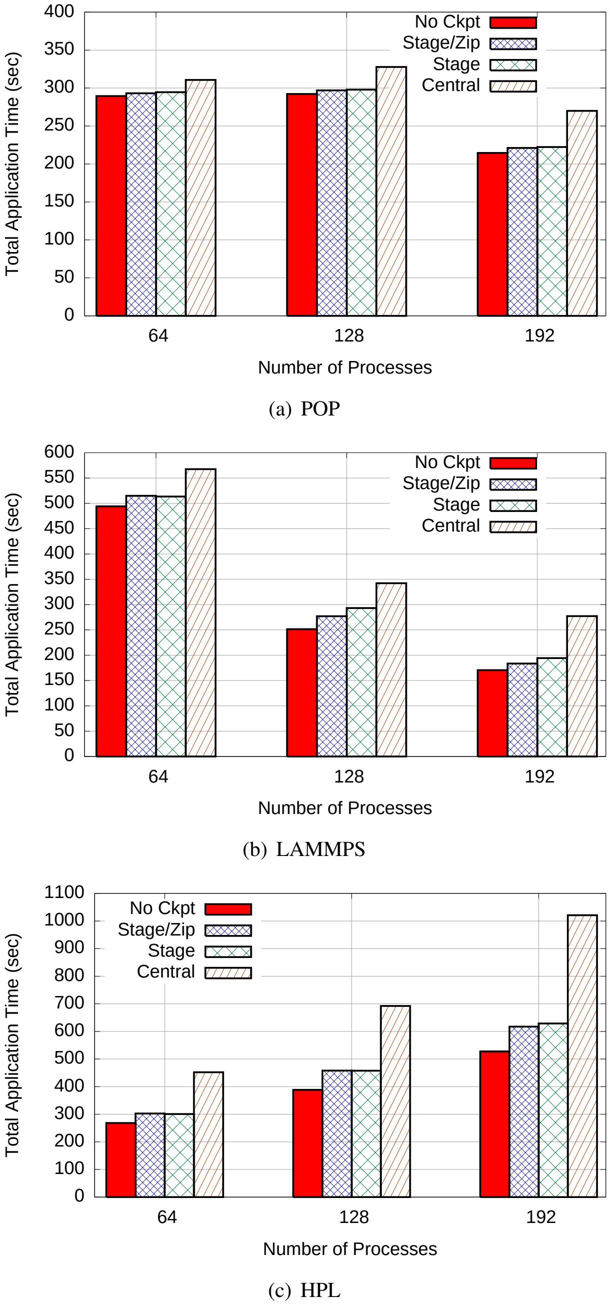
Compression adds slightly more to the checkpoint overhead in comparison with the default stage component, but reduces the duration of the effect. The checkpoint latency is reduced from 8.3 to 4.1 seconds by enabling compression, since this application is highly compressible as it resembles the compression rate of the `noop` application presented in Table I(a).

The checkpoint latency is reduced from 11.1 to 8.3 seconds by switching from the direct central storage (i.e., central) to the staging protocol. Often this reduction in checkpoint latency is caused by the flow control in the stage SStore and rsh FileM components which constrains the number of concurrent files in flight to 10, by default. The flow control focuses the write operations so that only a subset of the nodes are using the bandwidth to stable storage at the same time instead of all nodes fighting for the same exhausted bandwidth.

The performance impact on the application runtime for the various SStore component configurations is shown in Figure 3(a) for POP, Figure 3(b) for LAMMPS, and Figure 3(c) for HPL. All of these figures demonstrate that the stage SStore component is an improvement over the central component, especially for large checkpoint sizes. Interestingly, the compression enabled stage component may slightly improve the checkpoint overhead in comparison with the default stage component. Since the compression occurs on-node and competes for computational cycles, one might expect the opposite effect. However, if the application is sufficiently compressible, the overhead involved in checkpointing is regained by reducing the time to establish the checkpoint to stable storage, reducing the overall impact of checkpointing on the network and application.

B. Automatic Recovery

In this section we assess the performance of the automatic recovery ErrMgr component, autor, for various SStore component configurations. For this assessment, we are using the `noop` application to focus our investigation. The `noop` application is a naturally quiescent application (since it does not communicate) with a fixed process size allowing us to focus the analysis on the automatic recovery specific overheads. In this experiment failed processes are placed on different nodes than the ones they resided on before the failure. Processes are forcibly terminated by sending `SIGKILL` to the target processes from an external agent.



(a) POP

(b) LAMMPS

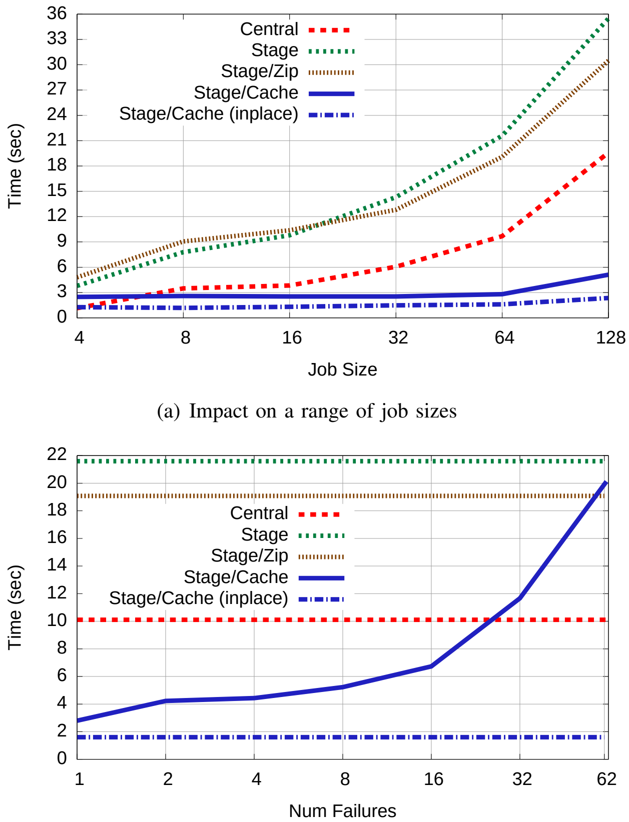
(c) HPL

Fig. 3. Checkpoint overhead impact of various SStore components.

First, we will assess the performance implications of automatically recovering from a single process failure for various job sizes restarting on spare machines in the allocation. Figure 4(a) presents the effect of using the following configurations of SStore components:

- central
- stage
- stage with compression enabled
- stage with caching enabled

From Figure 4(a), we can see that cache enabled stage component has drastic performance benefits as the job size



(a) Impact on a range of job sizes

(b) Impact on a range of concurrent process failures in a 64 process job.

Fig. 4. Performance impact of SStore components on automatic recovery.

increases providing constant recovery time of approximately 3 seconds. Caching reduces the recovery pressure on stable storage by allowing the non-failed processes to restart from the node-local storage while only the failed processes are forced to stage-in their local snapshots from stable storage.

The central component outperforms the cache enabled stage component for small job sizes. This is because even with a caching enabled stage component the failed processes must first copy the local snapshot to the node-local storage then restart from it. This is in contrast to the central component which avoids the copy to node-local storage by directly referencing the local snapshot. Notice also that the compression enabled stage component begins to outperform the default stage configuration at larger job sizes since it is reducing the amount of data being transferred between the stable storage device and node-local storage.

Figure 4(b) presents checkpoint latency for a variety of concurrent failures in a fixed size job, in this case 64 processes. The time to recover the job is not changed by the number of failures since for all of the non-cache enabled SStore components the entire job is terminated and recovered from stable storage. The main variable in Figure 4(b) is the recovery time when caching is enabled. We can see that caching continues to provide performance benefits up to about half of the job failing at which point the central component begins to perform better.

Interestingly, even up to 62 concurrent process failures a caching enabled stage component still performs better than the default stage component. This indicates that even with a few processes taking advantage of the node-local cache, in this case 2 processes, there are still performance benefits to not further stressing the stable storage device.

If we allow failed processes to be restarted on the node in which they previously failed, the benefits of caching becomes even more significant. The time to restart becomes approximately 1.5 seconds regardless of the number of concurrent failures or the job size. This is a slightly unrealistic use case for typical deployments since processes failing due to node failure often cannot access the node to restart from since it has crashed. However, this is an interesting data point for future investigations into peer-based, node-local storage that eliminates or reduces the need for logically centralized stable storage devices, confirming much of the previous literature [63], [64].

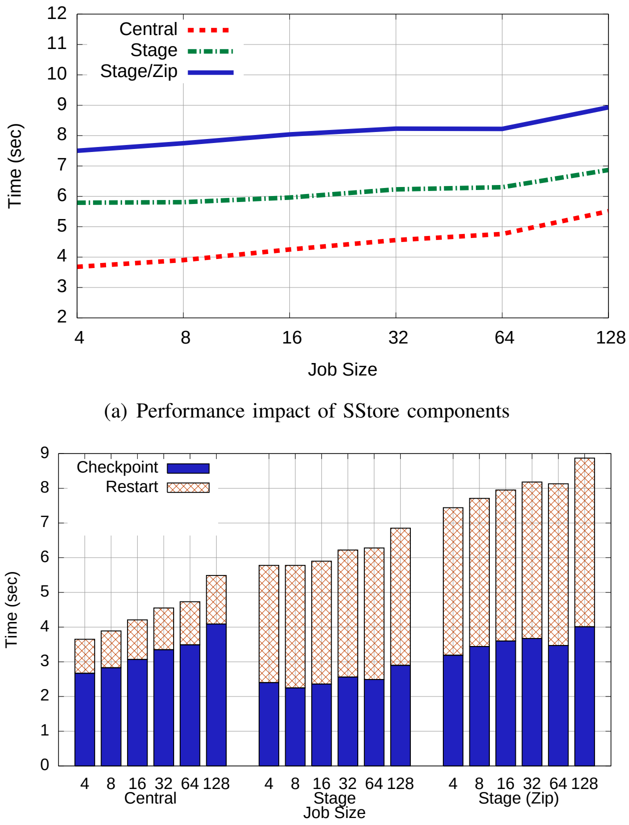
C. Process Migration

In this section we assess the performance of the eager copy process migration ErrMgr component, crmig. As in Section V-B, we are using the noop application to focus the analysis of the performance overheads involved in process migration. In this experiment, processes were migrated from a source set of machines to a destination set of machines that were distinct from the source set. So in this experiment, caching will not provide any benefit since the source node is never the same as the destination node for any of the migrating processes.

The common use case for proactive fault tolerance is to move an entire nodes worth of processes in anticipation of a node failure. First, we will assess the performance impact of migrating two processes while varying the size of the MPI job.

In Figure 5(a), we can see that the central component performs better than the either of the stage component configurations. Looking at the breakdown in Figure 5(b) we are able to see that the time to restart the processes remains fairly constant regardless of the job size among each SStore component. This is due to the relatively low bandwidth requirement of pulling two local snapshots from stable storage. The signifiant difference comes in the time to stage the local snapshot to and from stable storage. The reduction in the checkpoint overhead is beneficial when checkpointing for fault recovery, but contributes additional overhead to the process migration performance.

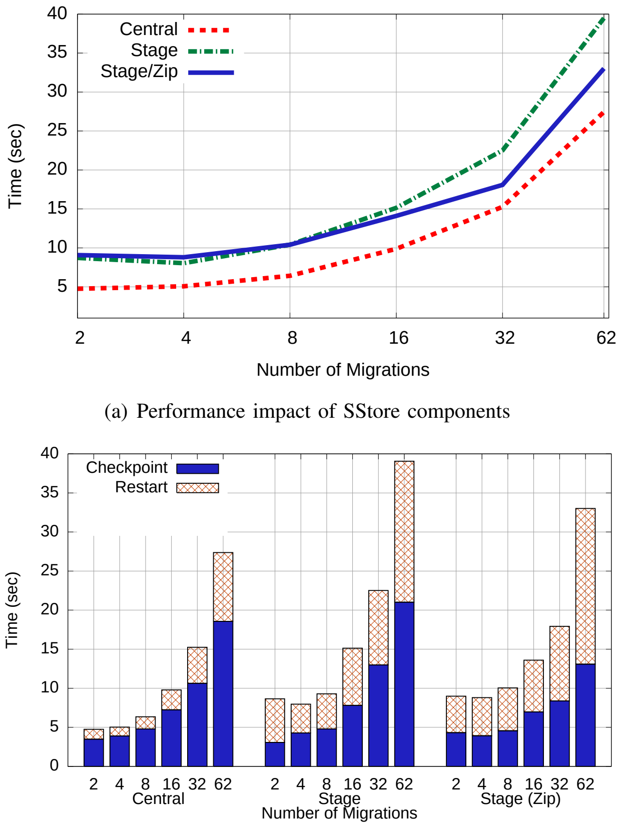
Process migration is limited by the time to move the checkpoint from the source to the destination system. Both the central and stage SStore components rely on a logically centralized stable storage device. As such, they copy the local snapshot to the stable storage device from the source machine then immediately copy it back from stable storage to the destination machine. For future work, we are investigating direct copy techniques that remove the logically centralized stable storage device from the process migration procedure.



(a) Performance impact of SStore components

(b) Breakdown of performance impact of SStore components.

Fig. 5. Performance impact of SStore components on process migration on a range of job sizes.



(a) Performance impact of SStore components

(b) Breakdown of performance impact of SStore components.

Fig. 6. Performance impact of SStore components on process migration on a range of migrating processes in a 64 process job.

Our discussion in this paper focuses on the performance implications of two common stable storage techniques provided by C/R enabled MPI implementations. So for a small number of migrating processes, the central component requires the least number of copies during the migration, and given the limited bandwidth requirements of migrating only two processes this SStore component is the best performance option.

Next we assess the performance implications of process migration by varying the number of migrating processes for a fixed size job, in this case 64 processes. Figure 6(a) shows that as the number of processes migrating are increased the compression enabled stage component begins to outperform the default stage component, and approaches the performance of the central SStore component. If we look at the breakdown of the migration overhead in Figure 6(b), we can see that the time to checkpoint increases as we increase the number of migrating processes, since we are checkpointing more processes and putting more pressure on the bandwidth to stable storage. Reciprocally, we can see the time to restart the migrating processes increases as the number of migrating processes increases for the same reasons. So it makes sense that the compression enabled stage component begins to approach the performance of the central component since it reduces the amount of data traveling over the network to and from stable storage.

D. Compression

Checkpoint compression can improve checkpoint latency by reducing the amount of data that needs to traverse the network to and from stable storage. The benefits of compression, in terms of improving checkpoint latency, is determined by how well the processes address space represented in the local snapshot can be compressed. If the checkpoint does not compress well this can negate much or all of the benefits of including compression in the staging pipeline. The checkpoint overhead may also increase since, if the compression occurs on the same machine as the process in execution, the two processes could compete for CPU cycles. In order to access the impact of compression on the checkpoint overhead and latency we looked at four applications.

As a baseline number we looked at benchmarking the `noop` program with 0 additional bytes of data, effectively a "hello world" style MPI program. Table I(a) presents the experimental data showing considerable improvements in the checkpoint latency when enabling compression. Since the checkpoint is 92% compressible the checkpoint latency is reduced by 71.4% for a 192 process MPI job. Since the `noop` program waits until it is signaled to finish the Application Performance numbers are not meaningful for this application.

(a) noop Compression Data

NP	Application Performance (sec.)					Checkpoint Latency (sec.)					Compression Rate (MB)		
	Central	Stage (%)		Zip (%)		Central	Stage (%)		Zip (%)		Normal	Zip	%
64	N/A	N/A	(– %)	N/A	(– %)	10.3	7.6	(25.8 %)	4.4	(57.5 %)	258.5	21.0	91.9 %
128	N/A	N/A	(– %)	N/A	(– %)	21.8	17.7	(18.8 %)	6.9	(68.4 %)	593.8	48.3	91.9 %
192	N/A	N/A	(– %)	N/A	(– %)	40.9	28.9	(29.5 %)	11.7	(71.4 %)	1167.4	93.3	92.0 %

(b) LAMMPS Compression Data

NP	Application Performance (sec.)					Checkpoint Latency (sec.)					Compression Rate (MB)		
	Central	Stage (%)		Zip (%)		Central	Stage (%)		Zip (%)		Normal	Zip	%
64	567.5	513.5	(9.5 %)	515.0	(9.3 %)	69.1	77.3	(-11.9 %)	38.0	(45.1 %)	4029.4	1474.6	63.4 %
128	342.3	293.1	(14.4 %)	277.1	(19.0 %)	87.1	90.7	(-4.1 %)	37.4	(57.1 %)	4638.7	1694.7	63.5 %
192	277.3	194.3	(30.0 %)	183.7	(33.8 %)	107.1	104.1	(2.7 %)	35.8	(66.6 %)	5427.2	1786.9	67.1 %

(c) POP Compression Data

NP	Application Performance (sec.)					Checkpoint Latency (sec.)					Compression Rate (MB)		
	Central	Stage (%)		Zip (%)		Central	Stage (%)		Zip (%)		Normal	Zip	%
64	310.7	294.6	(5.2 %)	293.1	(5.7 %)	19.7	13.8	(29.9 %)	6.1	(69.0 %)	609.3	106.2	82.6 %
128	327.7	297.9	(9.1 %)	297.0	(9.4 %)	35.3	27.9	(20.8 %)	8.5	(75.8 %)	1105.9	144.5	86.9 %
192	270.0	222.3	(17.7 %)	221.1	(18.1 %)	53.6	39.8	(25.9 %)	14.8	(72.3 %)	1802.2	205.9	88.6 %

(d) HPL Compression Data

NP	Application Performance (sec.)					Checkpoint Latency (sec.)					Compression Rate (MB)		
	Central	Stage (%)		Zip (%)		Central	Stage (%)		Zip (%)		Normal	Zip	%
64	452.2	300.8	(33.5 %)	302.7	(33.1 %)	182.4	218.3	(-19.7 %)	237.2	(-30.1 %)	13557.8	13020.2	4.0 %
128	692.3	457.6	(33.9 %)	458.1	(33.8 %)	301.8	371.6	(-23.1 %)	347.4	(-15.1 %)	24581.1	23234.6	5.5 %
192	1020.8	628.7	(38.4 %)	617.1	(39.5 %)	493.8	561.8	(-13.8 %)	551.0	(-11.6 %)	40105.0	37411.8	6.7 %

TABLE I
EFFECTS OF STAGING AND COMPRESSION ON APPLICATION PERFORMANCE AND CHECKPOINT OVERHEAD.

Next we consider the effect of compression on the metal benchmark for the LAMMPS software package. Table I(b) shows that compression can reduce the size of the LAMMPS global snapshot by up to 67% from 5.3 GB to 1.7 GB reducing not only the checkpoint latency by 67%, but also the amount of stable storage disk space required to store the checkpoint.

Next we consider the effect of compression on the *bench01.tacc* benchmark of the POP software package. Table I(c) shows that compression reduces the size of the global snapshot by up to 89% and the checkpoint latency by up to 76% constituting considerable savings for this application.

Finally we consider the effect of compression on the HPL software package. Table I(d) shows that compression only reduces the size of the global snapshot by up to 7%. Due to the low compression rate and the large checkpoint size the checkpoint latency increases, but the checkpoint overhead is reduced by up to 40%. All of the application studies show that the checkpoint overhead can be reduced by using a staging approach to stable storage in place of a direct approach.

VI. CONCLUSION

This paper presents recent advancements in the Open MPI project providing the foundational components of a resilient runtime environment to support resilient MPI applications. We discuss our extensions to the ErrMgr framework to provide composible, application specified recovery policies. In addition to run-though recovery, we discuss two additional ErrMgr recovery components that provide transparent, C/R based process migration and automatic recovery policies. We present a new stable storage framework, SStore, that provides the recovery policies with an abstract interface to the stable storage device. Additionally, we present an analysis of the impact on recovery policy performance for various stable storage strategies including staging, caching, and compression. The techniques and recovery policies discussed in this paper will be available in the Open MPI 1.5 release series.

Moving forward from this work we intend to continue research into providing a resilient MPI interface required to support resilient MPI applications. We also intend to investigate alternative recovery techniques to the ErrMgr framework including live-migration, replicated processes, and message logging based recovery. The initial results of using compression in the stable storage pipeline has encouraged us to continue investigation into better models of the effects of checkpoint compression on a wider range of applications. We anticipate investigating peer-based, node-local storage techniques as part of the SStore framework. This will include investigation into direct source to destination checkpointing to support process migration. Additionally, this will include adding a support for specialized C/R file systems (e.g., stdchk) within the SStore framework.

ACKNOWLEDGMENTS

This work was supported by grants from the Lilly Endowment; National Science Foundation NSF ANI-0330620, and EIA-0202048; and U.S. Department of Energy DE-FC02-06ER25750^A003.

REFERENCES

[1] Message Passing Interface Forum, “MPI: A Message Passing Interface,” in *Proceedings of Supercomputing ’93*. IEEE Computer Society Press, November 1993, pp. 878–883.

[2] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. 33, no. 6, pp. 518–528, 1984.

[3] A. Geist and C. Engelmann, "Development of naturally fault tolerant algortihms for computing on 100,000 processors," *Journal of Parallel and Distributed Computing*, 2002.

[4] C. Engelmann and A. Geist, "Super-scalable algorithms for computing on 100,000 processors," in *Proceedings of International Conference on Computational Science (ICCS)*, vol. 3514, no. 1, May 2005, pp. 313–320.

[5] Y. Kim, J. S. Plank, and J. J. Dongarra, "Fault tolerant matrix operations for networks of workstations using multiple checkpointing," *International Conference on High Performance Computing and Grid in Asia Pacific Region*, pp. 460–465, 1997.

[6] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, October 1998.

[7] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, "Fault tolerant high performance computing by a coding approach," in *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM Press, 2005, pp. 213–223.

[8] Z. Chen, M. Yang, G. Francia, and J. Dongarra, "Self adaptive application level fault tolerance for parallel and distributed computing," *IEEE International Parallel and Distributed Processing Symposium*, pp. 1–8, 2007.

[9] Z. Chen and J. Dongarra, "Algorithm-based fault tolerance for fail-stop failures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 12, pp. 1628–1641, 2008.

[10] Y. Du, P. Wang, H. Fu, J. Jia, H. Zhou, and X. Yang, "Building single fault survivable parallel algorithms for matrix operations using redundant parallel computation," *International Conference on Computer and Information Technology*, pp. 285–290, 2007.

[11] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra, "Recovery patterns for iterative methods in a parallel unstable environment," *SIAM Journal of Scientific Computing*, vol. 30, no. 1, pp. 102–116, 2007.

[12] H. Ltaief, E. Gabriel, and M. Garbey, "Fault tolerant algorithms for heat transfer problems," *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 663–677, 2008.

[13] W. Gropp and E. Lusk, "Fault tolerance in message passing interface programs," *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 363–372, 2004.

[14] G. E. Fagg and J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," in *Lecture Notes in Computer Science: Proceedings of EuroPVM-MPI 2000*, 2000, pp. 346–353.

[15] G. E. Fagg and J. J. Dongarra, "Building and using a fault-tolerant MPI implementation," *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 353–361, 2004.

[16] G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. Dongarra, "Extending the MPI specification for process fault tolerance on high performance computing systems," in *Proceedings of the International Supercomputer Conference (ICS) 2004*. Primeur, 2004. [Online]. Available: http://www.netlib.org/utk/people/JackDongarra/PAPERS/isc2004-FT-MPI.pdf

[17] J. Dongarra, G. Fagg, A. Geist, J. Kohl, P. Papadopoulos, S. Scott, V. Sunderam, and M. Magliardi, "HARNESS: Heterogeneous Adaptable Reconfigurable NEtworked SyStemS," *International Symposium on High-Performance Distributed Computing*, 1998.

[18] G. E. Fagg, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic, and J. J. Dongarra, "Scalable fault tolerant MPI: Extending the recovery algorithm," in *Proceedings of Recent Advances in Parallel Virtual Machine and Messaging Passing Interface Users' Group Meeting Euro PVMMPI*. Springer Heidelberg, Lecture Notes in Computer Science, 2005, pp. 67–75. [Online]. Available: http://www.springerlink.com/openurl.asp?genre=article\&id=doi:10.1007/11557265_13

[19] H. Jitsumoto, T. Endo, and S. Matsuoka, "ABARIS: An adaptable fault detection/recovery component framework for MPIs," *International Parallel and Distributed Processing Symposium*, 2007.

[20] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[21] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," in *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 562–570.

[22] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer*, vol. 30, no. 4, pp. 68–74, 1997.

[23] R. Batchu, A. Skjellum, Z. Cui, M. Beddhu, J. P. Neelamegam, Y. Dandass, and M. Apte, "MPI/FT: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing," in *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2001.

[24] S. Genaud and C. Rattanapoka, "P2P-MPI: A peer-to-peer framework for robust execution of message passing parallel programs on grids," *Journal of Grid Computing*, vol. 5, no. 1, pp. 27–42, 2007.

[25] T. LeBlanc, R. Anand, E. Gabriel, and J. Subhlok, "VolpexMPI: an MPI library for execution of parallel applications on volatile nodes," *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, September 2009.

[26] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, T. Kordenbrock, and R. Brightwell, "Increasing fault resiliency in a message-passing environment," Sandia National Laboratories, Tech. Rep. SAND2009-6753, October 2009.

[27] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Transactions Computer Systems*, vol. 3, no. 3, pp. 204–226, 1985.

[28] L. Alvisi and K. Marzullo, "Message logging: Pessimistic, optimistic, causal, and optimal," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 149–159, 1998.

[29] E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit," *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 526–531, May 1992.

[30] S. Rao, L. Alvisi, and H. M. Vin, "Egida: An extensible toolkit for low-overhead fault-tolerance," in *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*. Washington, DC, USA: IEEE Computer Society, 1999, p. 48.

[31] J. C. de Kergommeaux, M. Ronsse, and K. D. Bosschere, "MPL*: Efficient record/play of nondeterministic features of message passing libraries," in *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK: Springer-Verlag, 1999, pp. 141–148.

[32] A. Duarte, D. Rexachs, and E. Luque, "An intelligent management of fault tolerance in cluster using RADICMPI," *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, vol. 4192, pp. 150–157, 2006.

[33] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–18. [Online]. Available: http://portal.acm.org/citation.cfm?id=762815

[34] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello, "MPICH-V project: A multiprotocol automatic fault-tolerant MPI," in *International Journal of High Performance Computing Applications*, vol. 20, no. 3. Sage Publications, Inc., 2006, pp. 319–333.

[35] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006.

[36] A. Bouteiller, G. Bosilca, and J. Dongarra, "Retrospect: Deterministic replay of MPI applications for interactive distributed debugging," *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 297–306, 2007.

[37] J. S. Plank and W. R. Elwasif, "Experimental assessment of workstation failures and their impact on checkpointing systems," in *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*. Washington, DC, USA: IEEE Computer Society, 1998.

[38] N. H. Vaidya, "Impact of checkpoint latency on overhead ratio of a checkpointing scheme," *IEEE Transactions on Computers*, vol. 46, no. 8, pp. 942–947, 1997.

[39] R. H. B. Netzer and J. Xu, "Necessary and sufficient conditions for consistent global snapshots," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 2, pp. 165–169, 1995.

[40] D. Manivannan, R. H. B. Netzer, and M. Singhal, "Finding consistent global checkpoints in a distributed computation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 6, pp. 623–627, 1997.

[41] R. H. B. Netzer and Y. Xu, "Replaying distributed programs without message logging," in *HPDC '97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 1997.

[42] Y.-M. Wang, "Consistent global checkpoints that contain a given set of local checkpoints," *IEEE Transactions on Computers*, vol. 46, no. 4, pp. 456–468, 1997.

[43] J.-M. Helary, A. Mostefaoui, and M. Raynal, "Preventing useless checkpoints in distributed computations," in *SRDS '97: Proceedings of the 16th Symposium on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 1997.

[44] L. Alvisi, S. Rao, S. A. Husain, A. de Mel, and E. Elnozahy, "An analysis of communication-induced checkpointing," in *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*. Washington, DC, USA: IEEE Computer Society, 1999.

[45] D. Manivannan and M. Singhal, "Quasi-synchronous checkpointing: Models, characterization, and classification," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 703–713, 1999.

[46] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.

[47] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[48] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," *International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, Winter 2005.

[49] H. Jung, D. Shin, H. Han, J. W. Kim, H. Y. Yeom, and J. Lee, "Design and implementation of multiple fault-tolerant MPI over Myrinet (M^3)," *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, November 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1105760.1105798

[50] G. Stellner, "CoCheck: Checkpointing and process migration for MPI," in *10th International Parallel Processing Symposium (IPPS '96)*, 1996. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/IPPS.1996.508106

[51] Q. Gao, W. Yu, W. Huang, and D. K. Panda, "Application-transparent checkpoint/restart for MPI programs over InfiniBand," *International Conference on Parallel Processing*, pp. 471–478, Jan 2006.

[52] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for Open MPI," in *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, March 2007.

[53] J. Hursey, T. I. Mattox, and A. Lumsdaine, "Interconnect agnostic checkpoint/restart in Open MPI," in *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC 2009)*, June 2009.

[54] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "A job pause service under LAM/MPI+BLCR for transparent fault tolerance," *International Parallel and Distributed Processing Symposium*, 2007.

[55] M. R. Eskicioglu, "Design issues of process migration facilities in distributed systems," in *Scheduling and Load Balancing in Parallel and Distributed Systems*, 1995, pp. 414–424.

[56] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive fault tolerance for HPC with Xen virtualization," in *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 23–32.

[57] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration in HPC environments," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[58] C.-C. J. Li, E. M. Stewart, and W. K. Fuchs, "Compiler-assisted full checkpointing," *Software: Practice and Experience*, vol. 24, no. 10, pp. 871–886, 1994.

[59] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in *Usenix Winter Technical Conference*, January 1995, pp. 213–223.

[60] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, 2007. [Online]. Available: http://stacks.iop.org/1742-6596/78/012022

[61] L. M. Silva, J. G. Silva, S. Chapple, and L. Clarke, "Portable checkpointing and recovery," in *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing (HPDC '95)*. Washington, DC, USA: IEEE Computer Society, 1995.

[62] J. S. Plank, "Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques," in *SRDS '96: Proceedings of the 15th Symposium on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 1996.

[63] G. Bronevetsky and A. Moody, "Scalable I/O systems via node-local storage: Approaching 1 TB/sec file I/O," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-415791, 2009.

[64] S. Al-Kiswany, M. Ripeanu, S. S. Vazhkudai, and A. Gharaibeh, "stdchk: A checkpoint storage system for desktop grid computing," *International Conference on Distributed Computing Systems*, pp. 613–624, 2008.

[65] N. H. Vaidya, "Staggered consistent checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 694–702, 1999.

[66] H. Jin and K. Hwang, "Distributed checkpointing on clusters with dynamic striping and staggering," in *ASIAN '02: Proceedings of the7th Asian Computing Science Conference on Advances in Computing Science*. London, UK: Springer-Verlag, 2002, pp. 19–33.

[67] R. Oldfield, "Investigating lightweight storage and overlay networks for fault tolerance," *High Availability and Performance Computing Workshop 2006 (HAPCW06)*, October 2006.

[68] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth, "Modeling the impact of checkpoints on next-generation systems," in *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 30–46.

[69] V. Cate and T. Gross, "Combining the concepts of compression and caching for a two-level filesystem," *SIGPLAN Notices*, vol. 26, no. 4, pp. 200–211, 1991.

[70] P. Sobe, "Stable checkpointing in distributed systems without shared disks," in *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 214.2.

[71] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[72] J. M. Squyres and A. Lumsdaine, "The component architecture of Open MPI: Enabling third-party collective algorithms," in *Proceedings of the 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, V. Getov and T. Kielmann, Eds. St. Malo, France: Springer, July 2004, pp. 167–185.

[73] R. Gupta, P. Beckman, H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra, "CIFTS: A coordinated infrastructure for fault-tolerant systems," *International Conference on Parallel Processing (ICPP)*, 2009.

[74] P. W. Jones, P. H. Worley, Y. Yoshida, J. B. White, III, and J. Levesque, "Practical performance portability in the parallel ocean program (POP): Research articles," *Concurrency and Computation: Practice & Experience*, vol. 17, no. 10, pp. 1317–1327, 2005.

[75] S. J. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, pp. 1–19, 1995.

[76] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. (2008, September) HPL: A portable implementation of the High-Performance Linpack benchmark for distributed-memory computers. [Online]. Available: http://www.netlib.org/benchmark/hpl/