Design and Implementation for Checkpointing of Distributed Resources using Process-level Virtualization

Kapil Arya
Mesosphere, Inc.
San Francisco, CA, USA
kapil@mesosphere.io

Rohan Garg*
Northeastern University
Boston, MA, USA
rohgarg@ccs.neu.edu

Artem Y. Polyakov
Mellanox Technologies
Sunnyvale, CA, USA
artemp@mellanox.com

Gene Cooperman†
Northeastern University
Boston, MA, USA
gene@ccs.neu.edu

***Abstract*—System-level checkpoint-restart is a critical technology for long-running jobs in high-performance computing. Yet, only two approaches to checkpointing MPI applications continue to survive in wide use today. One approach is to use the kernel module-based BLCR in combination with an MPI checkpoint-restart service particular to the MPI implementation in use. Unfortunately, this lacks support for some important Linux system services such as SysV IPC (e.g., shared memory objects). A second approach has been to use the original 2009 DMTCP implementation (herein referred to as DMTCP-09) for transparent, system-level checkpointing. Unfortunately, DMTCP-09 lacked support for checkpointing many of the necessary features found by MPI in a modern batch environment. These include: ssh; the InfiniBand network; process migration (restarting an MPI application on different cluster nodes); and modified file path prefixes on restart (typically due to a changing current directory, mount points, library paths, etc.).**

This work presents DMTCP-PV, a new user-space transparent checkpointing system based on the concept of *process virtualization*. This approach separately models the state of each local or distributed subsystem while decoupling it from the core checkpointing engine. By separating these concerns, a domain expert can extend checkpointing into a new domain without any knowledge of the core checkpointing engine. This allowed DMTCP-PV to address the deficiencies noted above and many others. It is shown that the runtime overhead of DMTCP-PV is generally less than 1%, and the checkpointing time is dominated by the time to write an image file to stable storage.

***Index Terms*—checkpoint-restart, virtualization, fault tolerance, DMTCP**

I. Introduction

Checkpoint-restart is a critical technology for fault tolerance and long-running jobs in high-performance computing. This work argues for the need for system-level checkpointing that transparently adapts to the many existing and emerging distributed resources. Examples of such distributed resources that are not easily checkpointed include ssh and shared memory segments (both BSD-style shared memory and system V shared memory objects), and virtualization of the various existing and emerging network constructs: TCP sockets, InfiniBand, Cray GNI, Intel Omni-Path Architecture (OPA), etc.

For more moderately sized HPC clusters, both users and administrators face a more immediate need for transparent, system-level checkpointing. For users, there is the dilemma of what to do if a computation will run for days or weeks, since a typical batch reservation slot has a limit of 24 hours. For administrators, there is the dilemma of what to do when an administrator must bring down a cluster for system maintenance. One can stop accepting jobs in a 24-hour batch queue at least 24 hours prior to system shutdown. But this limits the flexibility and throughput of a smoothly running computer center.

The current state of the art in system-level checkpointing of MPI in batch environments has many deficiencies (see Section I-A). We are not arguing that the other approaches to checkpointing, such as BLCR, CRIU and ZapC, could not be extended to support the requirements for distributed checkpointing for modern MPI implementations. Rather, we argue only that those competing approaches would require additional specialized routines for each new distributed construct.

Here, we describe DMTCP-PV (Distributed MultiThreaded CheckPointing with Process Virtualization), a novel checkpointing system based on a layered software model. The use of layers reflects many of the same layers found in the design of a typical operating system. However, it also includes some higher layers that reflect user-space system services, such as ssh, InfiniBand, and an interface with the batch queue, which is typically SLURM (see Figure 3). The principles of this work have served as a foundation for several widely used checkpointing applications. (See Section II-C.)

DMTCP-PV can be compared with the original 2009 implementation of DMTCP, described in [1] and herein referred to as DMTCP-09. DMTCP-09 was based on a single, monolithic checkpointing model, as is the case for all preceding checkpointing packages.

In *process virtualization*, virtualization of system ids (such as pids, mount points, file and socket ids, etc.) occurs entirely in user space within the application process. The goal of process virtualization is to decouple the specialized expertise of the checkpointing package developers from the domain expertise of distributed resources. One uses a simple virtualization model and API to write resource-specific code concerning the checkpointing and restoration of the state of a relevant subsystem. The process virtualization model is introduced in Section II and described in detail in Section IV.

* This work was partially supported by the National Science Foundation under Grant ACI-1440788.

† This work was partially supported by the IDEX “Chaire d’attractivité” program of the Université Fédérale Toulouse Midi-Pyrénées under Grant 2014-345.

2168-9253/16 $31.00 © 2016 IEEE
DOI 10.1109/CLUSTER.2016.55

IEEE computer society

The contributions of this work are:

1) the principles of a simple, uniform computational model for checkpoint-restart based on process virtualization; and
2) examples that support some resources (external agents) that could not previously be checkpointed using earlier monolithic checkpoint systems;

A. Alternative checkpointing approaches for HPC

Application-level checkpointing has been employed by many users instead of system-level checkpointing, due to the difficulty of using system-level checkpoint-restart at many sites. This has two problems in comparison with transparent, system-level checkpointing. First, it does not decouple application expertise from checkpointing expertise. Each time a module of a large application is enhanced, an application developer familiar with the application must update the checkpointing routine to take account of the modified state in the updated module. (However, see [2] for a mixed-level approach to mediate this burden.) Second, application-level checkpointing is typically executed only at the end of a computation phase. Hence, when an administrator wishes to bring down the cluster for system maintenance without the ability to trigger a system-level checkpoint, applications may be terminated well after their last checkpoint. In that case, any work since the last checkpoint will be lost.

Another alternative for checkpointing of parallel or distributed computations over InfiniBand has been the use of an MPI checkpoint-restart service [3], [4], [5] in combination with BLCR [6], [7]. At this time, the primary example in common use is the MPICH checkpoint-restart service [8] (Nemesis channel for InfiniBand) for support of BLCR. The MPI implementations derived from MPICH (e.g, MVAPICH2 and Intel MPI) inherit this support. (As an example of the difficulty of this approach, the Open MPI developers have temporarily dropped support for their independent, BLCR-based checkpoint-restart service as of version 1.7 of Open MPI due to a lack of a maintainer for that code [9].)

Finally, PGAS has been defined as a key element in the path to exascale computing [10]. Hence, checkpointing of PGAS is critical. PGAS languages [11] are often implemented on top of MPI, which then require shared memory objects. Many MPI implementations (e.g., MVAPICH2 and Open MPI) have added support for hybrid computations such as those using both MPI and OpenSHMEM [12]. Unfortunately, since BLCR does not support shared memory objects, the current MPI checkpoint-restart services also do not support these implementations. See Section VIII(a) for a further discussion.

B. Organization of paper

Section II motivates the need for a model of process virtualization with a simple example concerning process ids. In particular, Section II-C describes some successes of DMTCP-PV using this model. Section III discusses the basic requirements for checkpointing distributed resources while Section IV presents a process virtualization based mechanism for checkpointing such resources. Sections V and VI present SSH and batch-queue plugins that are critical to checkpointing in a modern batch system. Section VII presents the performance. Section VIII summarizes the related work, and Section IX presents the conclusion.

II. User-Space Process Virtualization

Often, application processes violate an implicit closed-world assumption, in that, they must interact either with privileged external agents (e.g., ssh daemons) or even external hardware (e.g., the HCA adapter for InfiniBand). We refer to the interface between the "core" of the process and the external agents/subsystems as an *application surface* (see Figure 1).

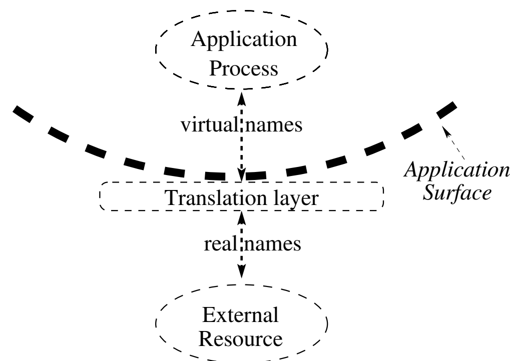


Fig. 1: *Application surface of a running process. The* ***virtual*** *names lie inside the application surface, whereas the* ***real*** *names lie outside the surface.*

When restarting from a checkpoint image, the recreated objects derived from external systems/services may not be the same as their pre-checkpoint version. This is due to the changing execution environment across a checkpoint-restart boundary.

User-space process virtualization finds a surface that is at least as large as the application surface, such that any virtualized view of an object lies inside this surface and any real view lies outside this surface. On restart, it links the recreated objects with their virtualized view inside the application surface. Thus, the application view of the objects does not change across checkpoint and restart.

In Subsection II-A, a simple example of process virtualization is presented, with the goal of describing the core ideas. In Subsection II-C, a brief overview is given of several successes using this type of process virtualization. With this motivation, the programming model for process virtualization is presented in Section IV.

A. Process Virtualization: A simple example with pids

As discussed in the introduction, restoring pids is the first challenge of checkpoint-restart that distinguishes the different approaches. In a process virtualization approach, pids are virtualized by creating a virtualization layer between the process and the O/S kernel. The virtualization layer hides

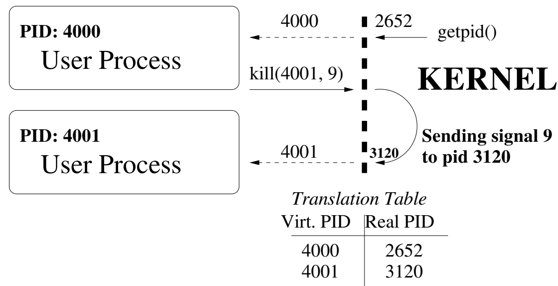


Fig. 2: *Process virtualization for pids. The application binary is inside the application surface, however, libc and the kernel are outside the application surface.*

the *real* pid (assigned by the kernel) from the user process and instead assigns it a *virtual* pid. Further, it maintains a translation table for translating between virtual and real pids as shown in Figure 2. Finally, at the time of restart, the translation table is updated with the real pids that were assigned to the restarted processes by the O/S kernel. This is achieved by having each process share its *real* pid with the peer processes through a publish/subscribe mechanism.

A virtualization layer at this application surface can be created using function wrappers. The application always calls the wrapper function, which will eventually call the "real" function inside libc. DMTCP-PV allows for composition of plugins: if two plugins contain wrappers for the same functions, the two wrappers are nested. This is the key to supporting virtualization *layers*.

Listing 1 illustrates such a wrapper which translates the *virtual* pid passed by an application to a `real` pid. In this example, the user has also chosen to make the wrapper an atomic transaction, by disabling checkpoints during its execution.

```
WRAPPER int kill(pid_t pid, int sig) {
  disable_ckpt();
  real_pid = virt_to_real(pid);
  int ret = REAL_kill(real_pid, sig);
  enable_ckpt();
  return ret;
}
```

Listing 1: *A function wrapper for pid virtualization*

B. Finding an application surface

There can be more than one possible application surface. Typically one chooses an application surface close to a well-known API for the sake of stability and maintainability. Recall that a typical application interacts with the execution environment through various libraries. For example, the libc runtime library provides access to the kernel resources, a device driver library provides access to the underlying device hardware, and so on. Thus, one can imagine virtualizing a resource by intercepting the relevant library calls. A wrapper around any call to the API updates both the virtual and the real view in a consistent manner. This allows us to inspect and modify the behavior of the underlying subsystem as seen by the application.

Choosing the application surface at a library API layer demonstrates a *caller-virtualized* approach, where we virtualize at the call site of the unprivileged caller. This is in contrast to the more common case of virtualizing at the site of the callee (*callee-virtualized*). For example, in an O/S container, pid virtualization takes place within the kernel.

The callee-virtualized mechanism may seem like the ideal solution since being close to the resource being virtualized gives us a better handle over the involved components. For example, the pid namespace in the Linux kernel solves the pid virtualization problem by keeping a translation table inside the kernel. However, the callee-virtualized approach fails for resources involving multiple hosts (e.g., network, InfiniBand) without a coordination protocol. The caller-virtualized approach, on the other hand, allows us to get similar results by virtualizing at the application surface to decouple the application process from the external subsystem. This eliminates the need to modify the callee or the user application.

C. Systems based on DMTCP-PV

DMTCP-PV is freely available through http://dmtcp.sourceforge.net and https://github.com/dmtcp. DMTCP-PV has been used as part of the HPC environment at the Center for Computational Research of the University at Buffalo, a large HPC center supporting 8,000 CPU cores, and integration with the SLURM resource manager is undergoing testing. DMTCP-PV also provides its own process virtualization module for SLURM launch and restart scripts. In this way, an end user may use checkpointing with SLURM batch jobs on an MPI implementation of their choice even though SLURM had not previously been configured to work with DMTCP.

Next, we provide some intuition on how process virtualization has been used in several critical technologies for using MPI in a batch environment. The following section then presents an overview of the process virtualization model, itself.

A more complete description of ssh is found in Section V. The key here is that checkpointing an ssh connection requires one to recreate a connection to a remote ssh daemon on restart. The ssh daemon is a privileged agent. Hence, the ssh plugin in DMTCP-PV is decoupled from the core of DMTCP-PV. The ssh plugin uses only the domain expertise concerning how the file descriptors for stdin/stdout/stderr are transmitted through the ssh daemon.

The details of support for the RC (reliable connection) mode for InfiniBand are covered in [13]. DMTCP-PV currently also supports UD (unreliable datagram mode). Modern MPI implementations require UD. The plugin now virtualizes a remote LID, which refers to an InfiniBand HCA adapter on a remote node. In fact, it is required to virtualize a pair consisting of an LID and a remote queue pair.

IP addresses are virtualized in a standard way, but they also require the publish/subscribe feature of DMTCP-PV (see Section IV) in order to correctly virtualize the remote IP addresses known to an application on restart. It was the lack

of virtualization of IP addresses in DMTCP-09 that prevented the earlier DMTCP-09 from restarting an MPI application on a new set of nodes. Path prefixes are virtualized in a plugin that adds a wrapper function around every system call in the C runtime library (libc) that refers to a pathname.

III. Checkpointing Distributed Resources

Distributed applications often have resources that are shared between multiple processes. Further, some resources such as file descriptors and SysV IPC objects are visible only within a compute node whereas resources such as socket and InfiniBand connections are visible over multiple nodes. Capturing and restoring the state while keeping a consistent virtualized view of a given shared resource is a non-trivial problem. Here we discuss the steps required to successfully checkpoint/restart distributed resources.

A. Checkpoint-leader election

In a distributed computation, resources may be shared explicitly (e.g., the dup() system call creates a duplicate file descriptor) or implicitly (by creating a child process; the child process gets a copy of all the file-descriptors, shared memory etc. automatically). However, only one of the several processes should be allowed to save/restore the state of the underlying resource. This is required for two reasons: (i) for some resources, part of the state to be checkpointed can be read only once. This is the case with data in kernel buffers or network data; and (ii) if multiple processes recreate the resource during restart, it may no longer be shared. In some situations, it is impossible to recreate a resource (e.g. sockets) by multiple processes, while in other cases, recreating a resource multiple times is permitted but results in incorrect behavior (e.g. same file can be opened by multiple processes resulting in loss of semantics).

B. Capture local and global state

A checkpoint-leader deploys domain-specific techniques to capture the state of an underlying resource. For example, the state of node-local resources such as open file handles, can be captured by querying the kernel. In the case of a network socket, one needs to save the in-flight network data by draining the network buffers as shown by [1]. Similarly, one has to drain various completion queues for capturing the state of InfiniBand connection as shown by [13]. The captured state is then checkpointed as part of process memory.

C. Recreate/restore resource state

On restart, the checkpoint-leader must recreate a semantically-equivalent copy of a given resource from the checkpointed state. For example, to restore a file handle, a new file descriptor is created and the file offset is restored to the pre-checkpoint state. For global resources such as socket connections, the peer processes need to exchange the current network addresses in order to recreate a new socket connection. Once the socket is recreated, the in-flight data is restored by pushing data back onto the network socket.

D. Update the virtualized view

As discussed in Section II-A, on restart, the virtual to real name mappings need to be updated to reflect the current names. Without it, a process will try to reach the underlying resource using the pre-checkpoint names.

E. Re-share resource

Once the shared resource has been recreated, it must be re-shared with peer processes. The kernel-based implementations can directly modify the kernel data structures of the target processes. DMTCP-09 shared the resources by restoring shared resources in a parent process and then forking child peer processes. This poses an inherent problem if the combined resources exceed the resource limit of a single process.

IV. Design and Implementation

A real-world application typically contains several types of interdependent distributed resources. In order to successfully checkpoint a distributed computation, one has to look at two axes of coordination: a horizontal coordination for checkpointing of a single distributed resource type (as discussed in the previous section); and vertical coordination for handling interdependent resources in a single process.

In this section, we present an architecture based on the idea of virtualization layers where each extra software layer is used to virtualize a resource at a lower layer. We then discuss the techniques that enable seamless horizontal coordination at every layer. This allows for a simple, intuitive programming model for extending the checkpointing system to adapt to a new external resource/subsystem. The concept follows the well-developed principle of using layers to develop and manage the complex code of a large operating system kernel.

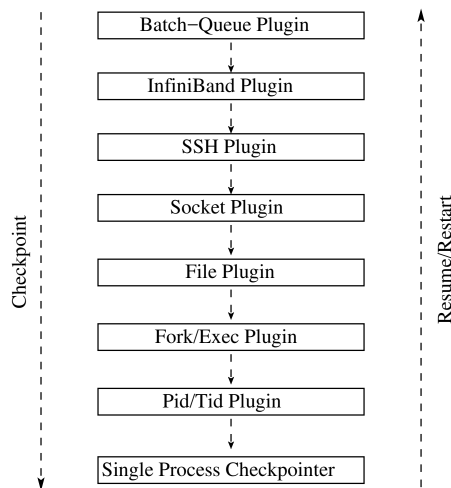


Fig. 3: *Virtualization layers in a distributed computation*

A. Vertical coordination using a plugin architecture

DMTCP-PV uses a plugin architecture to implement the layers. Each plugin is implemented as a dynamic library and

corresponds to a single virtualization layer and is responsible for checkpointing relevant resources at that layer. Figure 3 illustrates some of the plugins/layers in a distributed computation. Note that the layers may not require a total order. The total ordering in Figure 3 is an artifact of the dynamic library implementation.

In the current implementation, it is up to the end user to specify the ordering of virtualization layers upon each computation. Enhancements are possible, such as fields within each dynamic layer that specify "provides" and "requires" (e.g., requires: pid; provides: infiniband). It can be further automated by using library inspection tools to lookup symbols to determine a potential ordering constraint between layers.

1) Event callbacks: There are three important events in a checkpointing system: checkpoint time; resume time (resuming the original process after writing a checkpoint image); and restart time (restarting a computation as a new process from a checkpoint image file).

A straightforward approach is to have each virtualization layer register three callback functions for the three events (checkpoint, resume, and restart). On checkpoint, the callback functions are then called in order of the layers from top to bottom (since saving a higher layer may require calls to system services at a lower layer). On resume or restart, the callback functions are called in the opposite order, from bottom to top (since re-building a higher layer after restart may require the use of system services from a lower layer).
Remark: The first callback for the topmost layer takes place after the core checkpointing system has quiesced all user threads of the process to prevent them from making any changes to the computation state.

B. Horizontal coordination using barriers and publish/subscribe

While event callbacks provide a way to coordinate between multiple virtualization layers in a single process, a barrier allows for coordination between multiple processes at the virtualization layer corresponding to a single resource type. The proposal here is to integrate the two techniques with a publish/subscribe interface to checkpoint a distributed computation across multiple hosts, or multiple processes that use several types of interdependent resources.

In the unified model, each virtualization layer must still register for the three standard events (checkpoint, resume, and restart). But instead of registering for a single callback function, one registers several *sub-callbacks* with a barrier between two consecutive sub-callbacks.

We use a stateless centralized coordinator process to implement barriers. The coordinator also maintains a key-value database for providing publish/subscribe services. A virtualization may use the publish/subscribe service during the sub-callbacks for distributed coordination. Listing 2 uses this unified model to implement checkpoint-leader election.

In a more complex example, similar techniques are used to discover current addresses of a remote peer for restoring

```
sub_callback1() {
  for each id in file_id_list:
    publish(id, getpid());
}
sub_callback2() {
  for each id in file_id_list:
    if (subscribe(id) &=&  getpid):
      // We are ckpt leader;
      ckpt(id)
}
```

Listing 2: *Checkpoint leader election with the unified model.*

socket connection during restart. During checkpoint, each peer process publishes a (local socket-address, socket-id) pair and subscribes to <remote socket-address> for a given socket and saves the <socket-id> of the remote peer. On restart, each peer publishes a (socket-id, current-socket-address) pair and subscribes for the <remote socket-id> to receive the current socket-address of the remote process.
Remark: Note that the "ids" play a critical role in publish/subscribe and hence it is important to avoid any conflicts. There are several mechanisms to achieve this such as reading from `/dev/random` or the `libuuid` library.

1) Local and global barriers: As an optimization, barriers and publish/subscribe service are provided with two levels of scope: node-local and global. Node-local service is used for plugins that need to share information among processes on the same compute node (e.g., pid, file handles, etc.). Global service is needed for plugins related to distributed applications (e.g., socket, InfiniBand, etc.).

C. Discussion

Here we discuss two potential limitations of the wrapper-based process virtualization approach. It should be noted that the authors or the users of DMTCP-PV haven't yet experienced these limitations. Finally, the current design of plugin architecture supports well behaving and cooperating plugins only. Faulty/buggy plugins are not supported either.

a) Inline system calls: Wrapper mechanism can catch library function calls only. If some code makes system calls using inline assembly, the call won't be wrapped and the artifact related to the system call will not be virtualized or save-restored. A potential solution is to modify the application source code, replacing the inline assembly with a library function. Another solution is to use tools like PIN [14] which can dynamically instrument the binary and replace the inline assembly with desired function calls.

b) Certain assembly instruction: `RDTSC` is an assembly instruction used to read the time stamp counter. If one were to virtualize the clock, it would be necessary to replace RDTSC with some library function call. As with inline system calls, one can recompile the binary or use PIN.

V. CHECKPOINTING SSH

Recall that Secure Shell (SSH) allows two processes to securely communicate over an insecure network. The SSH daemon (`sshd`) is a privileged process. Checkpointing it by

an unprivileged user process is not possible since the user cannot recreate `sshd` on restart. Further, if shared connections are enabled, multiple connections to the remote host will share the same `sshd` process.

In order to checkpoint an SSH session, we need to find an application surface that would allow us to decouple the SSH daemon from the application processes. Further, the SSH client communicates with the SSH daemon process over a socket and shares some state such as session-id with the daemon. Thus, both `sshd` and the SSH client also needs to be outside the application surface.

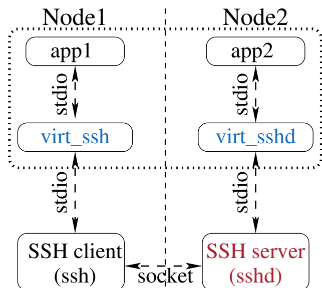


Fig. 4: *Virtualizing an SSH connection. The SSH client and server processes are not checkpointed.*

a) Launching remote process under checkpoint control: In order to keep both `ssh` and `sshd` processes outside the application surface, the SSH plugin intercepts the `exec` call to create two helper processes, `virt_ssh` and `virt_sshd`, when establishing the ssh connection as shown in Figure 4. These helper processes are inside the application surface and are checkpointed and restarted as part of the computation.

b) Checkpoint: At the time of checkpoint (see Figure 4), only processes `app1`, `app2`, `virt_ssh`, and `virt_sshd` are checkpointed. The `ssh` and `sshd` process are not under checkpoint control and are not checkpointed. Further, the `virt_ssh` and `virt_sshd` directly "drain" any in-flight network data that has not yet reached its destination. Thus, they act as buffers to hold network data prior to resume or restart. To guarantee that all of the in-flight messages have reached their destination, a "magic cookie" is sent across the write-end of each connection. The read-end is then polled until the arrival of magic cookie. A more elaborate solution would involve keeping account of the number of bytes sent and received for each connection. The `virt_ssh` and `virt_sshd` processes can then exchange this information out-of-band to find the status of in-flight data.

c) Restart: During restart, we need to reestablish the ssh connection between `virt_ssh` and `virt_sshd`. Since the application processes may be restarted on a different set of compute nodes, the `virt_ssh` processes must discover the current network address of `virt_sshd` process using publish-subscribe. It then launches a helper process on the remote node to reestablish the ssh connection. Finally, `virt_ssh` and `virt_sshd` processes write the previously drained network data to the corresponding pipes before resuming the computation.

VI. Checkpointing in Batch Environment

In high-performance computing, jobs to be run on clusters are typically submitted to a batch queue, and then executed on a cluster. Software components in this environment include: (i) an MPI library for inter-process communication; (ii) batch queue utilities (e.g., SLURM, Torque PBS, or LSF); (iii) a process manager (e.g., PMI or Hydra); and (iv) a low-latency network (e.g., InfiniBand, the Cray Aries interconnect, or the Intel Omni-Path Architecture (OPA)). Fault tolerance is an essential consideration for long-running HPC applications. In this section, we address the difficulties in checkpointing a distributed computation running in a batch environment.

A. MPI library

The vendor's MPI library is viewed as entirely within the application surface. Thus, any interposition of library calls will be done at a lower layer. This has the advantage of providing a solution that is independent of the particular MPI implementation.

B. Batch-queue utilities

Normally, an MPI application is launched through batch queue utilities. Scripts are developed based on the batch queue utility (e.g., for SLURM, Torque PBS, or LSF).

C. Resource manager

A resource manager provides the distributed application with a list of nodes and adjusts the standard output and error file descriptors to point to the log files. The application has no control over which nodes are allocated. In this case, the application surface includes the entire distributed application along with the MPI libraries and excludes the resource manager. Since the resource manager is in charge of communication among the distributed MPI processes, it presents a situation analogous to that of SSH.

a) Checkpoint: In particular, the resource manager launches user programs through: tm_spawn in the case of TORQUE PBS [15]; $lsb_launch()$ in the case of Load Sharing Facility (LSF) [16]; and the standalone $srun$ and $sbatch$ commands in the case of SLURM [17]. The current work supports the TORQUE and SLURM resource managers (with and without the PMI interface). Similar to SSH, the batch-queue layer intercepts these calls to inject a call to `dmtcp_launch` to bring the remote process under checkpoint control.

b) Restart: At the time of restart, the resource manager considers the restarting application a new job that is unrelated to any previous jobs. Thus, it allocates a new set of compute nodes (which may be different from the original set). The batch queue layer must remap the restarting processes onto the new set of nodes. The socket layer then takes care of reconnecting socket connections between processes, resulting in a transparent restart.

c) Communication between an MPI application and the external Process Management Interface (PMI): Most modern MPI implementations use or support the *Process Management Interface* (PMI) [18]. The PMI model comprises three entities:

the MPI library, the PMI library and the process manager. As mentioned earlier, the MPI library lies within the application surface. The PMI library and the external process manager both lie outside the application surface.

Currently there are several implementations of process manager entities, including the standalone Hydra package [19], and the PMI server of the SLURM resource manager. SLURM requires an MPI process to communicate with the privileged SLURM job step daemon, which is not under checkpoint control. In this case, the batch queue plugin finalizes the PMI session before checkpointing and recreates it afterward.

D. Changing mount points

In a batch environment, a computation might be restarted on a different set of nodes which might have different mount points for users home/work directories. We need to translate between the pre-checkpoint mount points and the current mount points for a successful restart. As discussed in Section II-C, a path virtualization plugin translates paths remembered by the application into correct paths as per the new mount points.

VII. Performance and Evaluation

For evaluating performance, four types of experimental data is provided: the scalability of DMTCP-PV (Section VII-B; the overhead for the SSH and Batch-queue virtualization layers (Section VII-C); the overhead for single-host applications (Section VII-D); and overhead of wrappers (Section VII-E). Finally, a programming model should also be measured according to the burden on the programmer. We use the number of lines of code in each plugin as a proxy for the programming burden in Table I.

Plugin	Lines of code	Wrappers	Plugin	Lines of code	Wrappers
Batch Queue	1,715	13	Socket	2,156	17
SSH	1,021	3	KVM	749	2
InfiniBand	2,500	34	SysV IPC	1,154	14
IB2TCP	1,000	31	Tun/Tap	351	3

*: Uses additional 899 lines of shared common code.

TABLE I: *Statistics for various plugins.*

A. Experimental setup

In this performance evaluation, the scalability experiments were conducted on Stampede [20] at TACC (Texas Advanced Computing Center). Stampede is currently the #10 supercomputer on the Top500 list [21]. Each computer node at Stampede has 16 cores, consisting of a dual-CPU Xeon ES-2680 configuration with 32 GB of RAM. Experiments use the Lustre parallel filesystem version 2.5.5 on Stampede.

The overhead runs testing the SSH virtualization layer were conducted on a cluster at the Massachusetts Green High-Performance Computing Center (MGHPCC). We reserved eight nodes with Intel Xeon E5-2650 CPUs running at 2 GHz. Each node was dual-CPU, for a total of 16 cores per node. The operating system was RedHat Enterprise Linux 6.4 with Linux kernel version 2.6.32.

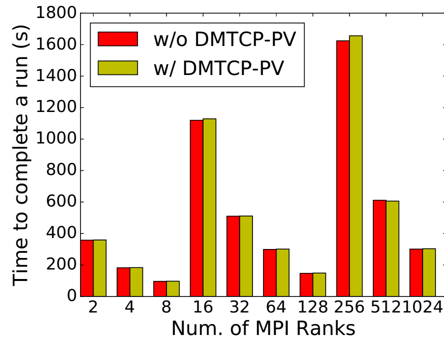


Fig. 5: *Runtime overhead on NAS LU benchmark with DMTCP-PV. The numbers are averaged over three runs. LU class C was used for 2, 4, and 8 MPI ranks. LU class D was used for 16, 32, 64, and 128 MPI ranks. LU class E was used for the runs with 256 and higher MPI ranks.*

The batch queue layer evaluation was conducted on a cluster with nodes equipped with two Intel Xeon CPUs running at 2.60 GHz for a total of 16 cores per node. The operating system was Scientific Linux 6.4 with Linux kernel version 2.6.32.

All single host application benchmarks, except MATLAB, were conducted on a dual-core Intel i7-2640M laptop computer with 8 GB RAM and Intel 320 series solid state disk running OpenSUSE 13.1 with Linux kernel 3.11.6. The MATLAB test was carried out on a 16-core 1.80 GHz Opteron Processor 8346 HE (4 quad-core sockets) with 128 GB RAM and 7200RPM hard disk drive running Ubuntu 13.04 with kernel 3.8.0-19-generic.

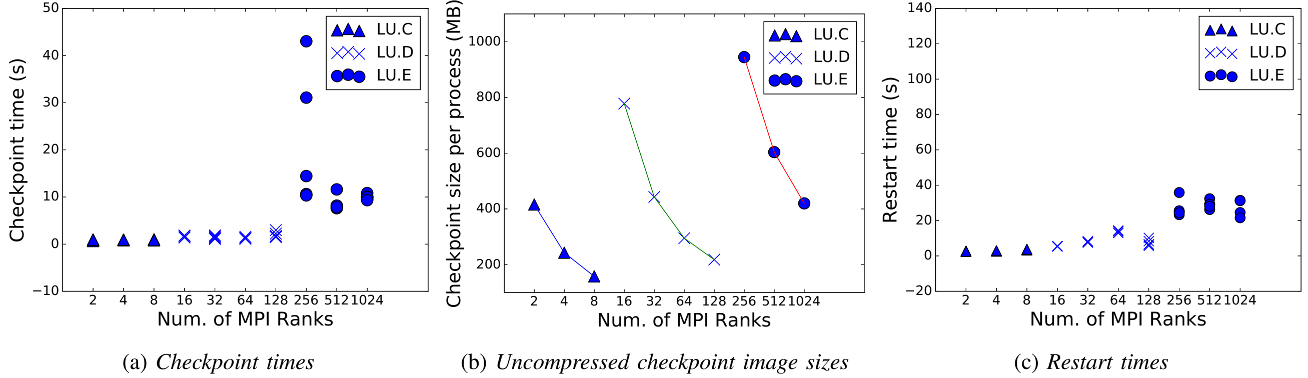
The NAS parallel benchmarks [22] were used to test scalability. Note that a given class, such as class E, represents a fixed problem size independently of the number of MPI processes.

B. Scalability

The NAS LU benchmark running under MVAPICH-2.0 was used to measure the scalability of DMTCP-PV. In particular, we measure the runtime overhead, the checkpoint overhead, and the restart overhead as we scale up. We also measure the cost of the barriers used by the different virtualization layers, including the cost to write to and read from stable storage.

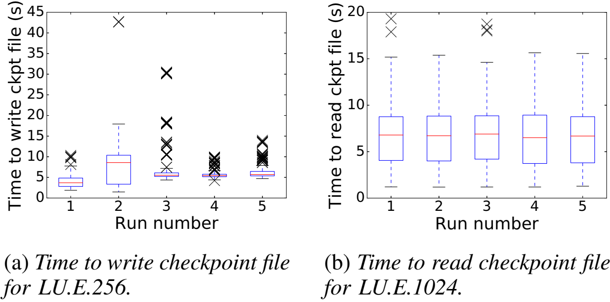
Figure 5 shows that the average runtime overhead imposed by DMTCP-PV. It is less than 1% in most cases. For a given number of MPI ranks, six runs were conducted – three under DMTCP-PV and three without DMTCP-PV – on the same set of nodes. The average overhead goes up to 1.9% in the case of LU.E.256, which we attribute to system noise.

Checkpoint times for the NAS LU benchmark are shown in Figure 6a. Five successive checkpoints were taken for a given number of MPI ranks on the same set of nodes. In the case of LU.E.256, we observe a large variation in the checkpoint time. We speculate that this was due to the congestion on the backend network and Lustre when the runs were conducted.



(a) *Checkpoint times* (b) *Uncompressed checkpoint image sizes* (c) *Restart times*

Fig. 6: *Checkpoint/restart times along with checkpoint size for NAS LU benchmark with DMTCP-PV. For a given number of MPI ranks, five successive checkpoints and restarts were done. LU class C was used for 2, 4, and 8 MPI ranks. LU class D was used for 16, 32, 64, and 128 MPI ranks. LU class E was used for the runs with 256 and higher MPI ranks.*



(a) *Time to write checkpoint file for LU.E.256.* (b) *Time to read checkpoint file for LU.E.1024.*

Fig. 7: *Time to write/read checkpoint file to/from stable storage. Each marker (indicated by "X") represents a single checkpoint file. LU.E.NNN implies NNN processes and so NNN checkpoint files. The upper horizontal edge of a box represents the first quartile (Q1). The lower horizontal edge of the box represents the third quartile (Q3). The horizontal line inside the box represents the median. The horizontal lines (whiskers) outside the box mark 1.5 times the interquartile range (IQR = Q3-Q1) beyond Q1 and Q3. The dots beyond the horizontal lines represent the outliers.*

Figure 6b shows checkpoint image size of each process for the NAS LU benchmark at different scales. In the case of LU, the image size for a given class size decreases with scale since the total data across all processes is fixed.

The restart times for the NAS LU benchmark are shown in Figure 6c. The average restart time roughly increases with the amount of checkpointing data read from the disk.

The InfiniBand plugin publishes key-value data at the time of restart. The data is then used to virtualize the InfiniBand queue-pairs for the application. The average key-value data published to the central checkpointing coordinator for each process varies from 401 bytes to 427 bytes.

a) Sources of checkpoint overhead: To quantify the contribution of different sources to the checkpointing overhead, we note the time taken by each process to write its checkpoint image to the disk and to execute the checkpointing barriers (as described in Section IV).

Figure 7a shows the distribution of the times to write a checkpoint image file to the disk reported by each of the 256 processes for the NAS LU.E.256 benchmark. The x-axis shows the distribution for five successive checkpoints, as seen in figure 6a. Note the unusually high times of 43 seconds and 30 seconds taken by some processes during the second and the fifth checkpoints. The straggling processes raise the overall checkpoint time of the entire computation.

For the second checkpoint for the NAS LU.E.256 benchmark, we observe that the time spent by a process waiting at a checkpointing barrier is less than 1 millisecond for all barriers. The time taken to execute a barrier is less than a microsecond. This shows that the time to checkpoint is dominated by the time to write to the disk.

b) Sources of restart overhead: Next, to quantify the contribution of different sources to the restart overhead, we note the time taken by each process to read its checkpoint image from the disk and the time taken by each process to execute the different restart-resume barriers.

Figure 7b shows the distribution of the time to read a checkpoint image from the disk reported by each of the 1024 processes for the NAS LU.E.1024 benchmark. The x-axis shows the distribution for five different restarts, as seen in Figure 6c.

We pick the worst of the five runs, run #1, for further analysis. We observe that a process can wait up to 26.3 seconds at each global barrier. The global barriers are used by the InfiniBand plugin on restart to propagate a consistent view of the state across all the restarted processes. Although, the state of the system published by the plugin (represented by the total key-value data at the central coordinator) is less than 410KB, the congestion on the network results in a high cost.

C. Runtime overhead for SSH and batch-queue plugins

a) SSH virtualization layer over TCP/IP: MPICH version 3.1 was run on the NAS MPI benchmarks to evaluate the performance of the SSH virtualization layer. When used outside a resource manager, MPICH uses SSH by default to launch daemons and jobs on remote nodes. The InfiniBand

network was not sued for these tests. The experiments were run by launching `mpiexec` itself under checkpoint control. The SSH layer ensured that the remote processes launched under SSH by MPICH were launched under DMTCP-PV checkpoint control.

For both native and checkpoint-enabled runs, each case was repeated five times in the shorter experiments for the sake of reproducibility, and three times for those experiments taking more than ten minutes. The median result of each case is reported. As shown in Table II, the runtime overhead is often close to zero with the highest overhead being 1.5%. In some cases, the performance with the SSH layer was measured as slightly better than running natively without the SSH layer. This is attributed to random interference of other jobs during the experiments.

Bench-mark	Native (s)	w/ SSH layer (s)	Overhead (%)	Ckpt (s)	Rst (s)
CG.C.8	52.21	52.75	1.0	0.24	0.92
CG.C.16	96.36	96.82	0.4	0.74	0.56
CG.C.32	209.91	211.64	0.8	1.34	0.17
CG.C.64	1839.89	1834.93	0.0	1.59	0.25
LU.C.8	132.30	130.59	0.0	1.43	0.92
LU.C.16	80.30	81.57	1.5	0.57	0.16
LU.C.32	68.49	68.11	0.0	0.58	0.22
LU.C.64	71.26	70.91	0.0	0.60	0.21

TABLE II: *Runtime overhead while using SSH layer*

b) Batch-queue and InfiniBand virtualization layers: NAS benchmarks running under Open MPI were used to measure the runtime overhead of checkpoint-enabled runs with the batch queue layer. The experiments used the SLURM resource manager and ran over InfiniBand.

Table III presents the runtime overhead of the checkpoint-enabled runs with the batch queue (BQ) layer. Similarly to the case of the SSH layer, the overhead of the batch queue layer is often zero with the highest overhead being 1.4%.

Benchmark	Native (s)	w/ BQ layer	Overhead (%)
CG.C.8	37.8	37.8	0.2
CG.C.32	9.5	9.4	0.0
CG.C.64	6.1	6.2	1.4
CG.D.128	106.3	106.9	0.5
LU.C.8	150.9	151.6	0.4
LU.C.64	28.0	27.9	0.0
LU.D.128	242.8	242.8	0.0

TABLE III: *Runtime overhead with batch queue layer*

D. Single-host application benchmarks

Table IV shows the runtime overhead of several applications when running with checkpointing support. The checkpoint and restart times along with checkpoint image sizes are also shown.

The applications are:

- Schedbench and syncbench are two micro-benchmarks from EPCC OpenMP Microbenchmarks V2.0 [23].
- The Regex-dna benchmark [24] matches DNA 8-mers and substitute nucleotides for IUB codes. The algorithm was implemented in three programs written in Javascript, Java and Python. The programs were passed 100 MB of data through stdin.
- The MATLAB LINPACK benchmark [25] carries out the LINPACK benchmark using MATLAB's "backslash" operator.

Application Benchmarks	Execution Time (s)			Ckpt time (s)	Rst time (s)	Ckpt size (MB)
	Native	w/ ckpt	Overhead (%)			
OpenMP-schedbench	9.5	9.5	0.4	0.075	0.058	42
OpenMP-syncbench	7.1	7.1	0.0	0.073	0.059	42
Regex-dna (Javascript)	7.8	7.9	1.2	0.519	0.273	877
Regex-dna (Java)	14.2	14.3	0.7	1.357	0.357	1160
Regex-dna (Python)	16.3	16.4	0.6	0.371	0.230	632
MATLAB (LINPACK)	21.8	22.1	1.3	3.343	0.786	1318

TABLE IV: *Checkpoint-restart times for various benchmarks. As we can see, the checkpoint restart times correspond to the checkpoint image size and are thus reflect the dominance of memory write operation.*

These applications were run with the socket, file, event, SysV IPC, timer, pid, and malloc virtualization layers present. For most virtualization layers, the runtime overhead is negligible and thus, the overhead was not measured for individual virtualization layers.

DMTCP-PV did not employ its default dynamic compression on the fly in creating the checkpoint images. However, DMTCP-PV always skips writing of zero pages to the checkpoint image. One could reduce the checkpoint image size further with DMTCP-PV compression (`-gzip`), but at the cost of increased checkpoint times and slightly increased restart times. The increase in checkpoint-restart times in using the gzip on-the-fly compression directly reflects the cost of compressing the image.

One could further decrease the checkpoint times significantly by using DMTCP-PV options such as forked-checkpointing — a forked process uses the copy-on-write mechanism to create the checkpoint image in the background while the actual process resumes computation. Similarly, restart times can be improved by using demand-paging of memory areas via the mmap system call, instead of reading in the entire checkpoint image at once.

For lightly loaded computers, the checkpoint and restart times may sometimes be close to zero. This can be attributed to caching in the buffers maintained by the operating system. Thus, the checkpoint cost has the potential to be close to the cost of memory-to-memory copy.

Note that while forked-checkpointing and buffer caches improve checkpoint times, there is an inherent danger of a node failure before the entire checkpoint image has been persisted to a stable storage. Thus, these techniques should be avoided if the application requires checkpoint images to be valid before continuing the execution.

E. Micro benchmarks

a) Wrapper overhead: To measure the overhead imposed by the wrapper functions in different virtualization layers, we

conducted an experiment with a program that opens and closes the "/dev/null" device in a loop. We measure the time taken to execute a million open and close calls with and without DMTCP-PV.

We observed that it takes 65 microseconds on average to execute each open call under DMTCP-PV. The overhead is dominated by the string comparison functions used in the open wrapper functions to virtualize filepaths. Note that this cost is amortized over the runtime of a process as shown in Sections VII-B and VII-C.

b) scp: To measure the overhead imposed by the SSH virtualization layer, we conducted a single experiment of transferring a 5 GB file over scp to the same host. We chose the destination to be the same as the source to eliminate noise due to network congestion. We ran the scp command natively and under DMTCP-PV. We did not observe any statistically significant difference in the throughput.

VIII. Related Work

Egwutuohaet al. [26] provide a survey of various checkpoint-restart implementations as of 2013. However, in practice, today, BLCR [6], [7] is the only other widely used implementation of transparent, system-level checkpoint-restart for MPI.

a) BLCR and MPI-based checkpoint-restart services: BLCR supports only single-node standalone checkpointing. In particular, it does not support checkpointing of TCP sockets, InfiniBand connections, open files, or SysV shared memory objects.

The commonly used MPI implementations today either use DMTCP-PV for transparent, system-level checkpointing or else support a checkpoint-restart service that employs BLCR. As stated in Section I-A, most of those MPI implementations are derived from MPICH and Open|MPI. As mentioned in Section I-A, the MPICH-derived implementations typically use the BLCR checkpointing support over InfiniBand, as provided by MPICH's Nemesis-IB channel (Nemesis channel with InfiniBand support). (MVAPICH had also developed an earlier BLCR-based checkpoint-restart service [5], but Nemesis-IB appears to be the preferred mechanism at this time.)

Open MPI has temporarily dropped support for the BLCR-based checkpoint-restart service due to a current lack of a maintainer. Quoting from the Open MPI website:

> "The checkpoint/restart support was last released as part of the v1.6 series. The v1.7 series and the Open MPI master do not support this functionality …. This feature is looking for a maintainer." [9]

Note that the current dependence of MPI checkpoint-restart services on BLCR implies that the MPI mechanisms do not support checkpointing of shared memory. Quoting from the BLCR User's Guide:

> "However, certain applications are not supported because they use resources not restored by BLCR: … Applications which use System V IPC mechanisms including shared memory, semaphores and message queues." [27]

Further, although BLCR is actively maintained, it is no longer under active development. This can be seen by observing that there have been just two minor releases over the last five years [28]: BLCR-0.8.4 (released Aug., 2011); and BLCR-0.8.5 (released Jan., 2013).

The lack of BLCR support for shared memory affects the use of OpenSHMEM [29] in HPC. Both MVAPICH2 (under the name MVAPICH2-X) and Open MPI support the hybrid MPI+OpenSHMEM model, but their BLCR-based checkpoint-restart services are unable to checkpoint under that model due to limitations of BLCR.

This has implications for checkpointing PGAS languages, such as UPC (Unified Parallel C/C++) [30], which are implemented on top of shared memory abstractions and are often based on top of MPI. PGAS will be a key technology for programmability in the exascale generation [10]. If the software stack relies on PGAS over MPI over BLCR, checkpointing will not be possible, since support for System V shared memory objects is lacking.

b) DMTCP-09: DMTCP-09 operates solely in user-space and can checkpoint MPI in a non-batch environment, but the computation cannot be restarted transparently on a different set of nodes. Support for restarting on a different set of nodes and SSH are critical for modern batch systems. The lack of virtualization support prevents it from supporting various mechanism such as interprocess communication via signals (requires pid virtualization) and SysV IPC objects (also requires virtualization). DMTCP-09's checkpoint-leader election is limited to resources associated with file descriptors (due to the use of fcntl system call). Thus, it cannot checkpoint other shared resources such as SysV IPC objects and BSD/POSIX-style shared-memory regions. Further, the lack programmable barriers and publish/subscribe makes it harder to implement and maintain support for new shared resources that require horizontal coordination for checkpointing.

DMTCP-PV [31], on the other hand, doesn't have any of these limitations as demonstrated by the support for checkpointing InfiniBand [13], network of virtual machines [32], and 3-D graphics [33] as discussed in Section II-C.

c) CRIU, ZapC and CRUZ: CRIU and ZapC/CRUZ represent two other checkpointing approaches. However, neither package supports the broad set of distributed resources needed for a modern MPI implementation. However, neither is usually used in checkpointing for HPC. CRIU [34] leverages Linux namespaces for transparently checkpointing containers on a single host, but lacks support for distributed computations.

ZapC [35] and CRUZ [36] were earlier efforts to support distributed checkpointing, by modifying the kernel to inserting hooks into the network stack using netfilter to translate source and destination addresses. ZapC and CRUZ are no longer in active use. They were designed to virtualize primarily two resources: process ids and IP network addresses. They did not support SSH, InfiniBand, System V IPC, or POSIX timers, all of which are commonly used in modern MPI implementations.

IX. Conclusion

A novel concept of process virtualization in user space has been introduced, which unifies events, barriers, and publish/subscribe. The events, barriers and publish/subscribe are used at the time of checkpoint-restart. Interposition through wrapper functions is also used. It is distinguished from older approaches by no longer requiring a monolithic approach to checkpoint-restart. Plugins implement layers in a manner analogous to typical implementations of operating systems. Several successes using process virtualization are also reviewed. These successes demonstrate the novel ability to checkpoint the state of an external privileged agent (e.g., an ssh daemon) and external hardware (e.g., the HCA adapter for InfiniBand). The new approach is shown to have extremely low overhead.

Acknowledgment

We would like to thank Alex Garthwaite, Jérôme Vienne and anonymous reviewers for their thoughtful comments and feedback on this work.

References

[1] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *23rd IEEE Intl. Parallel and Distributed Processing Symposium*, 2009, pp. 1–12.

[2] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, "Recent advances in checkpoint/recovery systems," in *20th Int. Parallel and Distributed Processing Symposium*. IEEE, 2006, pp. 8–15.

[3] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdain, "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) / 12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems*. IEEE Computer Society, March 2007.

[4] 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*, ser. HPDC '09. New York, NY, USA: ACM, 2009, pp. 49–58.

[5] Q. Gao, W. Yu, W. Huang, and D. K. Panda, "Application-transparent checkpoint/restart for MPI programs over InfiniBand," in *Int. Conf. on Parallel Processing, (ICPP'06)*. IEEE, 2006, pp. 471–478.

[6] J. Duell, P. Hargrove, and E. Roman, "The design and implementation of berkeley Lab's Linux checkpoint/restart (BLCR)," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-54941, 2003.

[7] P. Hargrove and J. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters," *Journal of Physics Conference Series*, vol. 46, pp. 494–499, Sep. 2006.

[8] MPICH team, "Checkpointing implementation - MPICH," Sep. 2010, https://wiki.mpich.org/mpich/index.php/Checkpointing_implementation , https://wiki.mpich.org/mpich/index.php/Checkpointing.

[9] Open MPI team, "Does Open MPI support checkpoint and restart of parallel jobs (similar to LAM/MPI)?" accessed May, 2016, https://www.open-mpi.org/faq/?category=ft#cr-support.

[10] W. Gropp, "MPI at exascale: Challenges for data structures and algorithms," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 16th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science **5759**. Springer-Verlag, Sep. 2009, pp. 3–3.

[11] PGAS, "PGAS — Partitioned Global Address Space languages," accessed May, 2016, http://www.pgas.org/.

[12] B. Chapman, T. Curtis, S. Pophale, S. e. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing OpenSHMEM: SHMEM for the PGAS Community," in *Proceedings of the Fourth Conference on Partitioned Global Addres s Space Programming Model*, ser. PGAS '10. New York, NY, USA: ACM, 2010, pp. 2:1–2:3.

[13] J. Cao, K. Arya, and G. Cooperman, "Transparent checkpoint-restart over InfiniBand," in *Proc. of ACM Symp. on High-Performance Parallel and Distributed Computing (HPDC'14)*. ACM Press, 2014.

[14] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation," in *Proc. of 37th Annual IEEE/ACM International Symposium on Microarchitecture*, Portland, Oregon, 2004, pp. 81–92.

[15] G. Staples, "Torque resource manager," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188464

[16] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: a load sharing facility for large, heterogeneous distributed computer systems," *Software: practice and Experience*, vol. 23, no. 12, pp. 1305–1336, 1993.

[17] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *9th International Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.

[18] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur, "PMI: A scalable parallel process-management interface for extreme-scale systems," in *Proc. of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, 2010, pp. 31–41.

[19] Hydra team, "Hydra process management framework." [Online]. Available: http://wiki.mcs.anl.gov/mpich2/index.php/Hydra_Process_Management_Framework

[20] "TACC Stampede user guide - TACC user portal," https://portal.tacc.utexas.edu/user-guides/stampede, accessed Apr., 2016, 2016.

[21] "TOP500 supercomputer sites," http://top500.org/lists/2015/11/, Nov. 2015.

[22] NASA Advanced Supercomputing Division, "NAS parallel benchmarks," http://www.nas.nasa.gov/publications/npb.html, accessed May, 2016.

[23] J. M. Bull and D. O'Neill, "A microbenchmark suite for OpenMP 2.0," *ACM SIGARCH Computer Architecture News*, vol. 29, no. 5, pp. 41–48, 2001.

[24] Regex-dna, "The computer language benchmarks game," Feb. 2015. [Online]. Available: http://benchmarksgame.alioth.debian.org/

[25] J. Burkardt, "The LINPACK benchmark using MATLAB's backslash," accessed May, 2016, http://people.sc.fsu.edu/~jburkardt/m_src/linpack_bench_backslash/linpack_bench_backslash.html.

[26] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, Sep. 2013.

[27] BLCR team, "Berkeley Lab Checkpoint/Restart (BLCR) user's guide," accessed May, 2016, https://upc-bugs.lbl.gov/blcr/doc/html/BLCR_Users_Guide.html.

[28] ——, "Berkeley Lab Checkpoint/Restart for Linux (BLCR) downloads," accessed May, 2016, http://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/berkeley-lab-checkpoint-restart-for-linux-blcr-downloads/.

[29] OpenSHMEM team, "Openshmem," accessed May, 2016, http://openshmem.org/site/Links#imp.

[30] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," IDA Center for Computing Sciences, Technical Report CCS-TR-99-157, 1999, http://upc.lbl.gov/publications/upctr.pdf.

[31] K. Arya, "User-space process virtualization in the context of checkpoint-restart and virtual machines," Ph.D. dissertation, Northeastern University, 2014.

[32] R. Garg, K. Sodha, Z. Jin, and G. Cooperman, "Checkpoint-restart for a network of virtual machines," in *Proc. of 2013 IEEE Computer Society Int. Conf. on Cluster Computing*. IEEE Press, 2013, 8 pages, electronic copy.

[33] S. Kazemi Nafchi, R. Garg, and G. Cooperman, "Transparent checkpoint-restart for hardware-accelerated 3D graphics," http://arxiv.org/abs/1312.6650v2, arXiv, Tech. Rep., 2014.

[34] CRIU team, "CRIU," accessed May, 2016, http://criu.org/.

[35] O. Laadan, D. Phung, and J. Nieh, "Transparent checkpoint-restart of distributed applications on commodity clusters," in *Cluster Computing, 2005. IEEE International*, Sep. 2005, pp. 1–13.

[36] G. Janakiraman, J. Santos, D. Subhraveti, and Y. Turner, "Cruz: Application-transparent distributed checkpoint-restart on standard operating systems," in *International Conference on Dependable Systems and Networks, 2005. DSN 2005. Proceedings*, Jun. 2005, pp. 260–269.