Transparent Checkpoint-Restart over InfiniBand

Jiajun Cao* Gregory Kerr* Kapil Arya* Gene Cooperman*
College of Computer and Information Science
Northeastern University
Boston, MA 02115 / USA
jiajun@ccs.neu.edu, kerrgi@gmail.com, kapil@ccs.neu.edu, gene@ccs.neu.edu

ABSTRACT

Transparently saving the state of the InfiniBand network as part of distributed checkpointing has been a long-standing challenge for researchers. The lack of a solution has forced typical MPI implementations to include custom checkpoint-restart services that "tear down" the network, checkpoint each node in isolation, and then re-connect the network again. This work presents the first example of transparent, system-initiated checkpoint-restart that directly supports InfiniBand. The new approach simplifies current practice by avoiding the need for a privileged kernel module. The generality of this approach is demonstrated by applying it both to MPI and to Berkeley UPC (Unified Parallel C), in its native mode (without MPI). Scalability is shown by checkpointing 2,048 MPI processes across 128 nodes (with 16 cores per node). The run-time overhead varies between 0.8% and 1.7%. While checkpoint times dominate, the network-only portion of the implementation is shown to require less than 100 milliseconds (not including the time to locally write application memory to stable storage).

Keywords

checkpoint/restart; InfiniBand; MPI; UPC

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability—*checkpoint/restart*

1. INTRODUCTION

InfiniBand is the preferred network for most of high performance computing and for certain Cloud applications, due to its low latency. Historically, transparent (system-initiated) checkpoint-restart has typically been the first technology that one examines in order to provide fault tolerance during long-running computations. *Checkpoint-restart* is the process of saving to stable storage (such as disk or SSD) the state of the processes in a running computation, and later re-starting from stable storage. The checkpoint-restart is *transparent* if no modification of the application is required. This is sometimes called *system-initiated* checkpointing.

Since transparent checkpoint-restart had not previously been available for distributed computations over InfiniBand, support for this important case had been based on: (i) "tearing down" the InfiniBand network connection; (ii) checkpointing the processed on each single computer node in isolation; and (iii) then re-building the network connection. Such schemes are typically implemented within each MPI implementation [15, 16, 25, 26], while using the BLCR kernel module [9, 14] for single-node checkpointing. However, such MPI-based implementations carry the overhead of waiting for completion of pending MPI messages, while blocking the sending of any new messages.

We present a new approach to checkpointing over InfiniBand. This is the first efficient and transparent solution for *direct* checkpoint-restart over the InfiniBand network (without the intermediary of an MPI checkpoint-restart service that is implementation-specific). This also extends to other language implementations over InfiniBand, such as Unified Parallel C (UPC [10]).

The new approach for InfiniBand provides at least three advantages:

1. Resuming after a checkpoint can be faster if there is no need to tear down and re-connect the network. (Section 4.2.2 shows the network-only portion of checkpointing to be two orders of magnitude faster than the older approach of Open MPI/BLCR.)

2. PGAS languages (e.g., UPC) often include two implementations of InfiniBand support: a direct implementation for greater network performance, and a refactoring on top of MPI in order to gain the advantage of MPI-based checkpointing. This work provides checkpoint-restart support in the direct case, thus supporting both speed and fault tolerance within a single implementation.

3. The use of the popular BLCR kernel module implies that the restart cluster must use the same Linux kernel as on the original checkpoint cluster. The new work eliminates this restriction.

The current work is implemented as a plugin on top of DMTCP (Distributed MultiThreaded CheckPointing) [1].

*This work was partially supported by the National Science Foundation under Grants OCI-0960978 and OCI 1229059, and by a grant from Intel Corporation.

HPDC'14, June 23–27, 2014, Vancouver, BC, Canada.

ACM 978-1-4503-2749-7/14/06 ...$15.00.
http://dx.doi.org/10.1145/2600212.2600219.

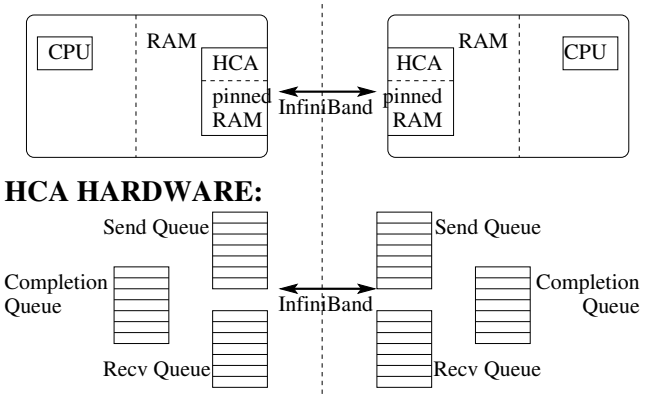

Figure 1: InfiniBand Concepts

The experimental evaluation demonstrates DMTCP-based checkpointing of Open MPI for the NAS LU benchmark and others. For 512 processes, checkpointing to a local disk drive, occurs in 232 seconds, whereas it requires 36 seconds when checkpointing to back-end Lustre-based storage. Checkpointing of up to 2,048 MPI processes (128 nodes with 16 cores per node) is shown to have a run-time overhead between 0.8% and 1.7%. This overhead is shown to be a little less than the overhead when using the checkpoint-restart of Open MPI using BLCR. Tests were also carried out on Berkeley UPC [5] over GASNet's ibv conduit [3], with similar results for checkpoint times and run-time overhead.

A particular advantage of the older MPI-based approach using network "tear-down" is the possibility to checkpoint on an InfiniBand network and restart on a TCP network (or vice versa). Such an interconnection-agnostic possibility was presented for the checkpoint-restart service of Open MPI [15]. The InfiniBand plugin design is intended to also be compatible with a future interconnection-agnostic capability for the new approach. An early proof of principle is presented here, showing an additional IB2TCP plugin, capable of checkpointing over InfiniBand and restarting over TCP in the special case of an MPI program over two nodes.

Finally, the number of lines of code of an implementation is a useful indicator of the likely effort required for maintenance. The primary InfiniBand plugin consists of 2,700 lines of code (while the additional IB2TCP plugin comprises 1,000 lines of code).

Organization of Paper.

Section 2 covers the background on the InfiniBand Verbs API and DMTCP. Section 3 provides design principles for checkpointing over InfiniBand. An experimental evaluation is presented in Section 4. Limitations and possible future directions are presented in Section 5. Finally, the related work (Section 6) and conclusions (Section 7) are presented.

2. BACKGROUND

Section 2.1 reviews some concepts of InfiniBand, necessary for understanding the checkpointing approach described in Section 3. Section 2.2 describes the use of plugins in DMTCP.

2.1 InfiniBand Verbs API

In order to understand the algorithm, we review some concepts from the Verbs API of InfiniBand. While there are several references that describe InfiniBand, we recommend one of [19, 2] as a gentle introduction for a general audience.

Recall that the InfiniBand network uses *RDMA* (remote DMA to the RAM of a remote computer). Each computer node must have a Host Channel Adapter (HCA) board with access to the system bus (memory bus). With only two computer nodes, the HCA adapter boards may be connected directly to each other. With three or more nodes, communication must go through an InfiniBand switch in the middle. Note also that the bytes of an InfiniBand message may be delivered out of order.

Figure 1 reviews the basic elements of an InfiniBand network. A hardware host channel adapter (HCA) and the software library and driver together maintain at least one *queue pair* and a *completion queue* on each node. The queue pair consists of a send queue and a receive queue. Sending a message across a queue pair causes an entry to be added to the completion queue on each node. However, it is possible to set a flag when posting a work request to the send queue, such that no entry is added to the completion queue on the "send" side of the connection.

Although not explicitly introduced as a standard, libibverbs (provided by the Linux OFED (OpenFabrics Enterprise Distribution)) is the most commonly used InfiniBand interface library. We will describe the model in terms of the functions prefixed by `ibv_` for the *verbs library* (libibverbs). Many programs also use OFED's convenience functions, prefixed by `rdma_*`. OFED also provides an optional library, librdmacm (RDMA connection manager) for ease of connection set-up and tear-down in conjunction with the verbs interface. Since this applies only to set-up and tear-down, this library does not affect the ability to perform transparent checkpoint-restart.

We assume the reliable connection model (end-to-end context), which is by far the most commonly used model for InfiniBand. There are two models for the communication:

- Send-receive model
- RDMA (remote DMA) model (often employed for efficiency, and serving as the inspiration for the one-sided communication of the MPI-2 standard)

Our InfiniBand plugin supports both models, and a typical MPI implementation can be configured to use either model.

2.1.1 Send-Receive Model

We first describe the steps in processing the send-receive model for InfiniBand connection. It may be useful to examine Figure 1 while reading the steps below.

1. Initialize a hardware context, which causes a buffer in RAM to be allocated. All further operations are with respect to this hardware context.
2. Create a protection domain that sets the permissions to determine which computers may connect to it.
3. Register a memory region, which causes the virtual memory to be pinned to a physical address (so that the operating system will not page that memory out).
4. Create a completion queue for each of the sender and the receiver. This completion queue will be used later.
5. Create a queue pair (a send queue and a receive queue) associated with the completion queue.

6. An end-to-end connection is created between two queue pairs, with each queue pair associated with a port on an HCA adapter. The sender and receiver queue pair information (several ids) is exchanged, typically using either TCP (through a non-InfiniBand side channel), or by using an rdmacm library whose API is transport-neutral.

7. The receiver creates a work request and posts it to the receive queue. (One can post multiple receive buffers in advance.)

8. The sender creates one or more work requests and posts them to the send queue.

9. The application must ensure that a receive buffer has been posted before it posts a work request to the send queue. It is an application error if this is not the case.

10. The transfer of data now takes place between a posted buffer on the send queue and a posted buffer on the receive queue. The posted send and receive buffers have now been used up, and further posts are required for further messages.

11. Upon completion, work completions are generated by the hardware and appended to each of the completion queues, one queue on the sender's node and one queue on the receiver's node.

12. The sender and receiver each poll the completion queue until a work completion is encountered. (A blocking request for work completion also exists as an alternative. A blocking request must be acknowledged on success.)

13. Polling causes the work completion to be removed from the completion queue. Hence, further polling will eventually see further completion events. Both blocking and non-blocking versions of the polling calls exist.

We also remark that a work request (a WQE or Work Queue Entry) points to a list of scatter/gather elements, so that the data of the message need not be contiguous.

2.1.2 RDMA Model

The RDMA model is similar to the send-receive model. However, in this case, one does not post receive buffers. The data is received directly in a memory region. An efficient implementation of MPI's one-sided communication (MPI_Put, MPI_Get, MPI_Accumulate), when implemented over InfiniBand, will typically employ the RDMA model [18].

As a consequence, Step 9 of Section 2.1.1 does not appear in the RDMA model. Similarly, Steps 11 and 12 are modified in the RDMA model to refer to completion and polling solely for the send end of the end-to-end connection.

Other variations exist, which are supported in our work, but not explicitly discussed here. In one example, an InfiniBand application may choose to send several messages without requesting a work completion in the completion queue. In these cases, an application-specific algorithm will follow this sequence with a message that includes a work completion. In a second example, an RDMA-based work request may request an immediate mode, in which the work completion is placed only in the remote completion queue and not in the local completion queue.

2.2 DMTCP and Plugins

DMTCP is a transparent, checkpoint-restart package that supports third-party plugins. The current work on InfiniBand support was implemented as a DMTCP plugin [8]. The plugin is used here to virtualize the InfiniBand resources exposed to the end user, such as the queue pair struct (ibv_qp) (see Figure 1). This is needed since upon restart from a checkpoint image, the plugin will need to create a new queue pair for communication. As a result, the InfiniBand driver will create a new queue pair struct at a new address in user space, with new ids.

Plugins provide three core features to support virtualization:

1. wrapper functions around functions of the InfiniBand library: these wrappers translate between virtual resources (seen by the target application) and real resources (seen within the InfiniBand library, driver and hardware). The wrapper function also records changes to the queue pair and other resources for later replay during restart.

2. event hooks: these hooks are functions within the plugin that DMTCP will call at the time of checkpoint and restart. Hence, the plugin is notified at the time of checkpoint and restart, so as to update the virtual-to-real translations, to recreate the network connection upon restarting from a checkpoint image, and to replay some information from the logs.

3. a publish/subscribe facility: to exchange ids among plugins running on the different computer nodes whenever new device connections are created. Examples of such ids are local and remote queue pair numbers and remote keys of memory regions.

3. DESIGN PRINCIPLES

The InfiniBand completion queue is the most complex of the subsystems being checkpointed. The key difficulty here is that at the time of checkpoint, the plugin needs to "drain" the notifications in the InfiniBand completion queue, and then re-insert those notifications at the time of resume or restart.

In this section, we will describe two orthogonal issues. First, Section 3.1 describes some important principles needed for draining the completion queue. Understanding of those underlying principles will be more enlightening than pseudo-code for a detailed implementation of an algorithm for draining the queue.

Second, Section 3.2 describes how the DMTCP plugin virtualizes the InfiniBand ids (e.g., rkey, qp_num, lid, pd). This is a key issue since the ids are shared among distributed processes. InfiniBand will typically assign new ids, when DMTCP restarts from a checkpoint image. Since the InfiniBand library and application code may have already cached the pre-checkpoint ids, the plugin uses its wrapper functions to interpose and pass on only virtual ids to the application and libraries. The plugin maintains an internal table of virtual and real ids. This table must be consistently updated across all processes on restart.

Figure 2 presents an overview of the virtualization of a queue pair. Observe that the DMTCP plugin library interposes between most calls from the target application to the InfiniBand ibverbs library. This allows the DMTCP InfiniBand plugin to intercept the creation of a queue pair by the

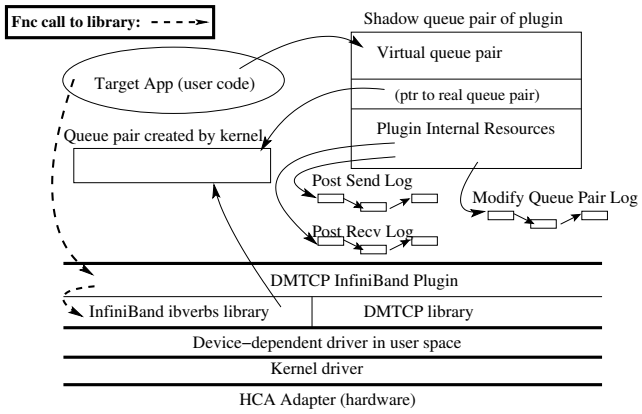

Figure 2: Queue pair resources and their virtualization. (The plugin keeps a log of calls to post to or to modify the queue pair.)

InfiniBand kernel driver, and to create a shadow queue pair. The target application is passed a pointer only to the virtual queue pair created by the plugin. Thus, any further ibverbs calls to manipulate the queue pair will be intercepted by the plugin, and appropriate fields in the queue pair structure can be appropriately virtualized before the real ibverbs call.

Similarly, any ibverbs calls to post to the send or receive queue, or to modify the queue pair, are intercepted and saved in a log. This log is used for internal bookkeeping by the plugin, to appropriately model work requests as they evolve into the completion queue.

Note also a subtle corner case: a call to ibv_post_send may request that no work completion entry be entered for that one call. The log must account for this through later calls that provide a completion entry, similarly to typical application code that works with InfiniBand.

In this work, we always use the three terms checkpoint, resume and restart as follows. *Checkpoint* refers to saving the state, *resume* refers to the original process resuming the computation, and *restart* refers to launching a new process that will restart from stable storage.

3.1 Draining the Completion Queue

As the user base code makes calls to the verbs library, we will use DMTCP plugin wrapper functions around these library functions to interpose. Hence the user call is first received by our DMTCP plugin library. We then extract parameters describing how the resources were created, before passing on the call to the verbs library, and later passing back the return value. This allows us to recreate semantically equivalent copies of those same resources on restart *even if we restart on a new computer*. In particular, we record any calls to `modify_qp` and to `modify_srq`. On restart, those calls are replayed in order to place the corresponding data structures in a semantically equivalent state to pre-checkpoint.

While the description above appears simple, several subtleties arise, encapsulated in the following principles.

Principle 1: Never let the user see a pointer to the actual InfiniBand resource.

A verbs call that creates a new InfiniBand resource will typically create a struct, and return a pointer to that struct. We will call this struct created by the verbs library a *real struct*. If the end user code creates an InfiniBand resource, we interpose to copy that struct to a a new *shadow struct*, and then pass back to the end user the pointer to this shadow struct. Some examples of InfiniBand resources for which this is done are: a context, a protection domain, a memory region, and a queue pair.

The reason for this is that many implementations of InfiniBand libraries contain additional undocumented fields in these structs, in addition to those documented by the corresponding "man page". When we restart after checkpoint, we cannot pass the original pre-checkpoint struct to the verbs library. The undocumented (hidden) fields would not match the current state of the InfiniBand hardware on restart. (New device-dependent ids will be in use after restart.)

So, on restart, we create an entirely new InfiniBand resource (using the same parameters as the original). This new struct should be semantically equivalent to the pre-checkpoint original, and the hidden fields will correspond to the post-restart state of the hardware.

This is a form of virtualization. The user is passed a pointer to a *virtual struct*, the shadow struct. The verbs library knows only about the *real struct*. So, we will guarantee that the verbs library only sees real structs, and that the end user code only sees virtual structs.

To do this, we interpose our DMTCP plugin library function if a verbs library function refers to one of these structs representing InfiniBand resources. If the end user calls a verbs library function that returns a pointer to a real struct, then our interposition will replace this and return a pointer to a corresponding virtual struct. If the user code passes an argument pointing to a virtual struct, we will replace it by a pointer to a real struct before calling the verbs library function.

Remark: In the OFED ibverbs implementation, some of the apparent library calls to the verbs library are in fact inline functions. A DMTCP plugin cannot easily interpose on inline functions. Luckily, these inline functions are often associated with possibly device-dependent functions. However, each of the important OFED inline functions expands to a dispatch through a global function pointer. So, the plugin resets the global function pointer to a plugin function, which wraps a call to the original function pointer.

Principle 2: Carry out bookkeeping on posts of work queue entries to the send and receive queue.

As work requests are entered onto a send queue or receive queue, the wrapper functions of the DMTCP plugin record those work requests (which have now become work queue entries). When the completion queue is polled, if a completion event corresponding to that work queue entry is found, then the DMTCP plugin records that the entry has been destroyed. At the time of checkpoint, there is a log of those work queue entries that have been posted and not yet destroyed. At the time of restart, the send and receive queues will initially be empty. So, those work queue entries are re-posted to their respective queues. (In the case of resume, the send and receive queues continue to hold their work queue entries, and so no special action is necessary.)

Principle 3: At the time of checkpoint, "drain" the completion queue of its completion events.

At the time of checkpoint, and after all user threads have been quiesced, the checkpoint thread polls the completion

queue for remaining completion events not yet seen by the end user code. A copy of each completion event seen is saved by the DMTCP plugin. Note that we must drain the completion queue for each of the sender and the receiver. Recall also that the verbs library function for polling the completion queue will also remove the polled completion event from the completion queue as it passes that event to the caller.

Principle 4: At the time of restart or resume, "refill" a virtual completion queue.

At the time of restart or resume and before any user threads have been re-activated, we must somehow refill the completion queue, since the end user has not yet seen the completion events that were drained (see previous principle). To do this, the DMTCP plugin stores the completion events of the previous principle in its own private queue. The DMTCP plugin library then interposes between any end user calls to a completion queue and the corresponding verbs library function. If the end user polls the completion queue, the DMTCP wrapper function passes back to the end user the plugin's private copy of the completion events, and the verbs library function for polling is never called. Only after the private completion queue becomes empty are further polling calls passed on to the verbs library function. Hence, the plugin's private queue becomes part of a *virtual completion queue*.

Principle 5: Any InfiniBand messages still "in flight" can be ignored.

If data from an InfiniBand message is still in flight (has not yet arrived in the receive buffer), then InfiniBand will not generate a completion event. Note that the InfiniBand hardware may continue to transport the data of a message, and even generate a completion event *after all user threads have been quiesced for checkpoint*. Nevertheless, a simple rule operates.

If our checkpoint thread has not seen a completion event that arrived late, then we will not have polled for that completion event. Therefore, our bookkeeping in Principle 2 will not have removed the send or receive post from our log. Further, this implies that the memory buffers will continue to have the complete data, since it was saved on checkpoint and restored on restart. Therefore, upon restart (which implies a fresh, empty completion queue), the checkpoint thread will issue another send or receive post (again following the logic of Principle 2).

Remark: Blocking requests for a completion event (`ibv_get_cq_event`) and for shared receive queues create further issues. While those details add some complication, their solution is straightforward and is not covered here.

3.2 Virtualization of InfiniBand Ids

A number of InfiniBand objects and associated ids will change on restart. All of these must be virtualized. Among these objects and ids are ibv contexts, protection domains, memory regions (the local and remote keys (lkey/rkey) of the memory regions), completion queues, queue pairs (the queue pair number, qp_num), and the local id (lid) of the HCA port being used. Note that the lid of an HCA port will not change if restarting on the same host, but it may change when restarting on a new host, which may have been configured to use a different port.

In all of the above cases, the plugin assigns a virtual id and maintains a translation table between virtual and real id. The application sees only the virtual id. Any InfiniBand calls are processed through the plugin, where virtual ids are translated back to real ids.

On restart, the InfiniBand hardware/driver may assign new real ids for a given InfiniBand object. In this case, the real ids are updated within the translation tables maintained by the plugin.

3.2.1 Virtualization of remote ids: rkey, qp_num and lid

A more difficult issue occurs in the case of remote memory keys (rkey), queue pair numbers (qp_num) and local ids (lid). In all three cases, an InfiniBand application must pass these ids to a remote node for communication with the local node. The remote need will need the qp_num and lid when calling `ibv_modify_qp` to initialize a queue pair that connects to the local node. The remote node will need the rkey when calling `ibv_post_send` to send a message to the local node.

Since the plugin allows the application to see only virtual ids, the application will employ a virtual id when calling `ibv_modify_qp` and `ibv_post_send`. The plugin will first replace the virtual id by the real id, which is known to the InfiniBand hardware. To do this, the plugin within each remote node must contain a virtualization table to translate all virtual ids by real ids.

Next, we recall how a remote node received a virtual id in the first place. The InfiniBand specification solves this bootstrapping problem by requiring the application to pass these three ids to the remote node through some out-of-band mechanism. When the application employs this out-of-band mechanism, the remote node will "see" the virtual ids that the plugin passed back to the application upon completion of an InfiniBand call.

The solution chosen for the InfiniBand plugin is that it assigns a virtual id, which is the same as the real id at the time of the initial creation of the InfiniBand object. After restart, the InfiniBand hardware may assign a new real id. At the time of restart, the plugin uses the DMTCP coordinator and the publish-subscribe feature to exchange the new real ids, associated with a given virtual id. Since the application continues to see only the virtual ids, the plugin can continue to translate between virtual and real ids through any wrapper by which the application communicates to the InfiniBand hardware (see Figure 2). (A subtle issue can arise if a queue pair or memory region is created after restart. This is a rare case. Although we have not seen this in the current work, Section 5 discusses two possible solutions.)

3.2.2 Virtualization of rkeys

Next, the case of rkeys (remote memory region keys) poses a particular problem that does not occur for queue pair numbers or local ids. This is because an rkey is guaranteed unique by InfiniBand only with respect to the protection domain within which it was created. Thus, if a single InfiniBand node has received rkeys from many remote nodes, then the rkeys for two different remote nodes may conflict.

Normally, InfiniBand can resolve this conflict because a queue pair must be specified in order to send or receive a message. The local queue pair number determines a unique queue pair number on the remote node. The remote queue pair number then uniquely determines an associated protec-

tion domain *pd*. With the remote *pd*, all rkeys are unique. Hence, the InfiniBand driver on the remote node uses the (*pd*, rkey) pair, to determine a unique memory address on the remote node.

In the case of the InfiniBand plugin, the vrkey (*virtual rkey*) and rkey are identical if no restart has taken place. (It is only after restart that the rkey may change, for a given vrkey). Hence, prior to the first checkpoint, translation from vrkey to rkey is trivial.

After a restart, the InfiniBand plugin must employ a strategy motivated by that of the InfiniBand driver. In a call to `ibv_post_send`, the target application will pass the required parameters, including both a virtual queue pair number and a virtual rkey (vrkey). Unlike InfiniBand, the plugin must translate the vrkey into the real rkey on the local node. However, during a restart, each node has published its locally generated rkey, the corresponding *pd* (as a globally unique id; see above), and the corresponding vrkey. Similarly, each node has published the virtual queue pair number and corresponding *pd* for any queue pair generated on that node. Each node has also subscribed to the above information published by all other nodes.

Hence, the local node is aware of the following through publish-subscribe during restart:

$$(\text{virtualqp\_num}, \text{pd})$$
$$(\text{vrkey}, \text{pd}, \text{realrkey})$$

The call to `ibv_post_send` provides the (local) virtual qp_num, and the vrkey. The previous InfiniBand calls building the connection had provided the corresponding remote virtual qp_num. The first of the publish-subscribe tuples above yields the globally unique *pd*. The *pd* and *vrkey* together are then enough to use the second tuple to derive the necessary rkey, which is used when calling the InfiniBand hardware.

4. EXPERIMENTAL EVALUATION

The experiments are divided into four parts: scalability with more nodes in the case of Open MPI (Section 4.1); comparison between BLCR and DMTCP for MPI-based computations (Section 4.2); tests on Unified Parallel C (UPC) (Section 4.3); and demonstration of migration from InfiniBand to TCP (Section 4.4).

Experimental Configuration.

Two clusters were employed for the experiments described here. For scalability tests with up to 2048 cores, a large cluster was reserved for our sole use (Section 4.1). This was the Massachusetts Green High-Performance Computing Center (MGHPCC), with Intel Xeon E5-2650 CPUs running at 2 GHz. Each node is dual-CPU, for a total of 16 cores per node. It employs Mellanox HCA adapters. In addition to the front-end InfiniBand network, there is a Lustre back-end network. The operating system is RedHat Enterprise Linux 6.4 with Linux kernel version 2.6.32. (Section 4.4 also used this cluster in small tests, not as the sole user.)

Sections 4.2 and 4.3 refer to a cluster at the Center for Computational Research at the University of Buffalo. It uses SLURM as its resource manager, and a common NFS-mounted filesystem. Each node is equipped with either a Mellanox or QLogic (now Intel) HCA, although a given partition under which an experiment was run was always homogeneous (either all Mellanox or all QLogic). The operating system is RedHat Enterprise Linux 6.1 with Linux kernel version 2.6.32.

In Section 4.2, experiments were run using one core per computer. Hence, the MPI rank was equal to the number of computers, and each MPI process was on a separate computer node. Not all computer nodes were identical. For reproducibility, a uniform memory limit per CPU was set at 3 GB. The CPUs had clock rates ranging from 2.13 GHz to 2.40 GHz.

In all cases, we used Open MPI 1.6, DMTCP 2.1 (or a pre-release version in some cases), and BLCR 0.8.3, respectively. Open MPI was run in its default mode, which used the RDMA model for InfiniBand, rather than the send-receive model. Although DMTCP version 2.1 was used, the plugin included some additional bug fixes appearing after that DMTCP release. For the applications, we used Berkeley UPC (Unified Parallel C) version 2.16.2 and NAS Parallel Benchmark version 3.1.

Tests of BLCR under Open MPI were run by using the Open MPI checkpoint-restart service [15]. Tests of DMTCP for Open MPI did not use the checkpoint-restart service. For DMTCP, all checkpoints are saved to a local disk (local to the given computer node), except as noted. Open MPI/BLCR uses the same strategy, except that it copies each local checkpoint image to a central coordinator process. Unfortunately, this serializes part of the parallel checkpoint. Hence, checkpoint times for BLCR are not directly comparable to those for DMTCP.

In case of DMTCP, the experimental timings reported here did not employ any particular tuning techniques and were run using the default DMTCP parameters. Thus, there are opportunities to reduce the run-time overhead by reducing the copying of buffers.

DMTCP also supports a faster forked checkpointing mode (taking advantage of checkpointing a forked child process under copy-on-write), and a fast restart using mmap to overlap running and reading in the remaining pages. Checkpoint times can also be sped up by omitting the DMTCP default on-the-fly gzip compression. See [12] for experiments exploring these extra options.

4.1 Scalability of InfiniBand Plugin

Table 1, and its graphical representation in Figure 3, present a study of scalability for the InfiniBand plugin. The NAS MPI test for LU is employed. For a given number of processes, each of classes C, D, and E are tested provided that the running time for the test is of reasonable length. The overhead for DMTCP is analyzed further in Table 2.

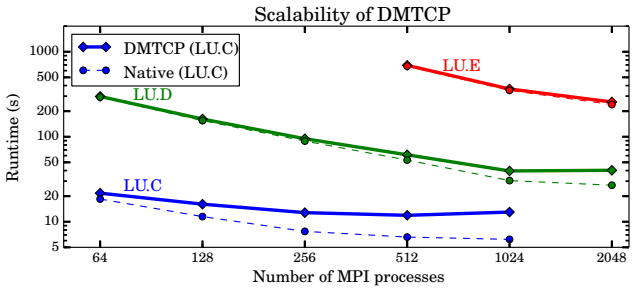

Figure 3: Plot based on Table 1.

NAS benchmark	Num. of processes	Runtime (s) (natively)	Runtime (s) (w/ DMTCP)
LU.C	64	18.5	21.7
LU.C	128	11.5	16.1
LU.C	256	7.7	12.8
LU.C	512	6.6	11.9
LU.C	1024	6.2	13.0
LU.D	64	292.6	298.0
LU.D	128	154.9	161.6
LU.D	256	89.0	94.8
LU.D	512	53.2	61.3
LU.D	1024	30.5	39.6
LU.D	2048	26.9	40.3
LU.E	512	677.2	691.6
LU.E	1024	351.6	364.9
LU.E	2048	239.3	256.4

Table 1: Demonstration of scalability: running times without DMTCP (natively) and with DMTCP; The corresponding plot is in Figure 3.

Table 1 and Figure 3 show runtimes decreasing with more MPI processes. This is because NAS experiments are based on *strong scalability*: each benchmark consists of a fixed amount of work. Hence, the runtime decreases with an increasing number of MPI nodes. The DMTCP plugin shows small overhead compared to native runs, except for cases where the runtime is below about 50 s. In these cases, startup overhead becomes a significant percentage of the total runtime (see Table 2).

# processes (running LU)	NAS classes	Startup overhead (s)	Slope (runtime overhead in %)
64	C, D	3.1	0.8
128	C, D	4.4	1.5
256	C, D	5.0	0.9
512	D, E	7.6	1.0
1024	D, E	8.7	1.3
2048	D, E	12.9	1.7

Table 2: Analysis of Table 1 showing derived breakdown of DMTCP overhead into startup overhead and runtime overhead. (See analysis in text.)

In Table 2, the overhead derived from Table 1 is decomposed into two components: startup overhead and runtime overhead. Given a NAS parallel benchmark, the total overhead is the difference of the runtime with DMTCP and the native runtime (without DMTCP). Consider a fixed number of processes on which two different classes of the same benchmark are run. For example, given the native runtimes for two different classes of the LU benchmark (e.g., t_1 for LU.C and t_2 for LU.D), and the total overhead in each case (o_1 and o_2), one can derive an assumed startup overhead s in seconds and runtime overhead ratio r, based on the formulas:

$$o_1 = s + rn_1$$
$$o_2 = s + rn_2.$$

Table 2 reports the derived startup overhead and runtime overhead using the formula above. In cases where three classes of the NAS LU benchmark were run for the same number of nodes, the largest two classes were chosen for analysis. This decision was made to ensure that any timing perturbations in the experiment would be a small percentage of the native runtimes.

NAS benchmark	Number of processes	Ckpt time (s)	Ckpt size (MB)
LU.E	128×4	70.8	350
LU.E	64×8	136.6	356
LU.E	32×16	222.6	355
LU.E	128×16	70.2	117

Table 3: Checkpoint times and image sizes for the same NAS benchmark, under different configurations. The checkpoint image size is for a single MPI process.

The runtime overhead shown in Table 2 remains in a narrow range of 0.8% to 1.7%. The startup overhead grows as the cube root of the number of MPI processes.

Table 3 shows the effects on checkpoint time and checkpoint image size under several configurations. Note that the first three tests hold constant the number of MPI processes at 512. In this situation, the checkpoint size remains constant (to within the natural variability of repeated runs). Further, in all cases, the checkpoint time is roughly proportional to the total size of the checkpoint images on a single node. A checkpoint time of between 20 MB/s and 27 MB/s was achieved in writing to local disk, with the faster times occurring for 16 processes per node (on 16 core nodes).

Next, a test was run to compare checkpoint times when using the Lustre back-end storage versus the default checkpoint to a local disk. As expected, Lustre was faster. Specifically, Table 4 shows that checkpoint times were 6.5 times faster with Lustre, although restart times were essentially unchanged. Small differences in checkpoint image sizes and checkpoint times are part of normal variation between runs, and was always limited to less than 5%.

Disk type	Ckpt size (MB)	Ckpt time (s)	Restart time (s)
local disk	356	232.3	11.1
Lustre	365	35.7	10.9

Table 4: Comparison with checkpoints to local disk or Lustre back-end. Each case was run for NAS LU (class E), with 512 processes (32 nodes × 16 cores per node).

Finally, a test was run in which DMTCP was configured not to use its default gzip compression. Table 5 shows that this makes little difference both for the checkpoint image size and the checkpoint time. The checkpoint time is about 5% faster when gzip is not invoked.

Program and processes	Ckpt size (MB)	Ckpt time (s)	Restart time (s)
with gzip	117	70.2	23.5
w/o gzip	116	67.3	23.2

Table 5: Comparison of checkpointing with and without the use of gzip for on-the-fly compression by DMTCP.

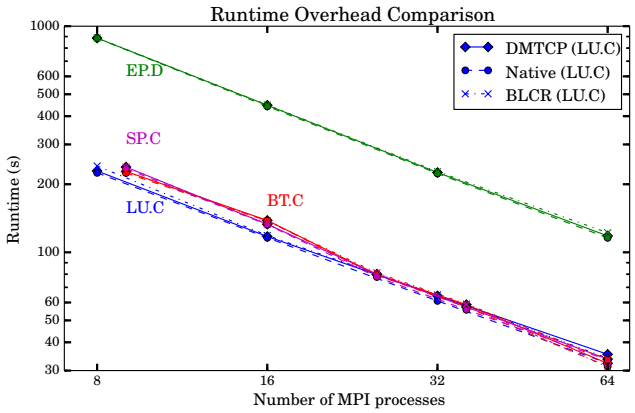

Figure 4: Comparison of running natively or with DMTCP or BLCR

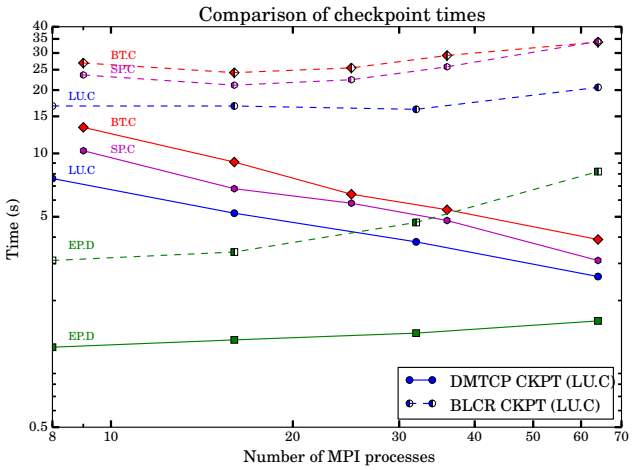

Figure 5: Comparison of checkpoint times

4.2 Comparison between DMTCP and BLCR

The NAS Parallel Benchmarks using Open MPI provide a test of performance across a broad test suite. For the sake of comparability with previous tests on the checkpoint-restart service of Open MPI [15], we emphasize the previously used NAS tests: LU.C, EP.D, BT.C and SP.C. An analysis of the performance must consider both runtime overhead and times for checkpoint and restart.

4.2.1 High-Level overview of DMTCP vs. BLCR

Runtime overhead.

The runtime overhead is the overhead for running a program under a checkpoint-restart package as compared to running natively. No checkpoints are taken when measuring runtime overhead. Figure 4 shows that the overhead of running DMTCP or BLCR is typically only a few per cent, and the runtime performance of the two systems is comparable. (Curves are based on strong scalability: work is held constant as the number of MPI processes varies.) For longer program runs, the total runtime overhead is in the range of 1% to 2%, or 1 to 5 seconds, with 3 seconds being common. Since these overhead times do not correlate with the length of time for which the program was run, we posit that they reflect the constant overhead incurred primarily at the time of program startup (see, for example, Table 2).

Checkpoint/Restart times.

Figure 5 shows checkpoint times. These times are as reported by a central DMTCP coordinator, or by an Open MPI coordinator in the case of BLCR. Open MPI/BLCR does not report restart times. For DMTCP, restart times are approximately between 2 and 3 seconds for LU.C and EP.D, and between 2 and 4 seconds for BT.C and SP.C. Consistent with strong scalability, both checkpoint and restart times were longer when there were fewer MPI processes.

Checkpoint times are particularly important for issues of fault tolerance, since checkpoints are by far the more common operation. In the case of DMTCP, For the BT, SP, and LU benchmarks, the memory footprint decreases with an increasing number of MPI processes, thus accounting for decreasing checkpoint times. The EP *Embarrassingly Parallel* benchmark is an exception, in which the memory footprint remains approximately 150 MB, independently of the number of processes.

In contrast, the times for checkpointing with Open MPI/BLCR are often roughly constant. We estimate this time is dominated by the last phase, in which Open MPI copies the local checkpoint images to a single, central node.

Note that the restart time for DMTCP was typically under 4 seconds, even for larger computations using 64 computer nodes. The Open MPI/BLCR package did not report restart times.

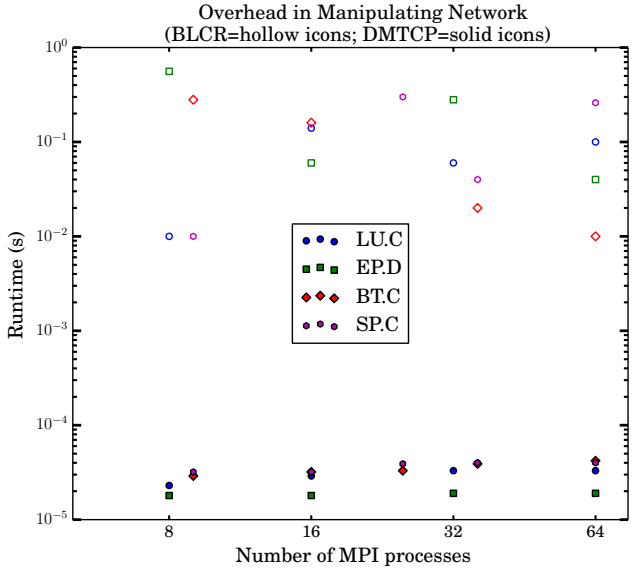

Figure 6: Overhead due to network for DMTCP and for BLCR. For BLCR, three times were measured for each case, and the middle time is reported.

4.2.2 Overhead of Network Management: DMTCP vs. BLCR

Figure 6 has the goal of analyzing the network-only portion of the checkpoint/restart time for both DMTCP and BLCR. For DMTCP, this includes the time of draining all completion queues, as well as the time of saving all Infini-

Band states. For BLCR, this includes all times for BLCR to tear down and rebuild the InfiniBand network. These times are not a bottleneck for the current experiments, when checkpointing to disk. But future technologies promise faster stable storage, which can expose the network-related time as a major cost.

Since Open MPI/BLCR must tear down the InfiniBand network, and re-build it at the time of checkpoint, it incurs a significantly higher network cost, as expected. Two advantages of the current approach are exposed in Figure 6:

1. The DMTCP plugin approach is shown to be over 100 times faster (two orders of magnitude), as compared to BLCR.

2. For any particular NAS benchmark, the DMTCP approach presents a clear and predictable trend, while the times for BLCR vary by an order of magnitude even within the same NAS benchmark and for the same number of MPI processes, under repeated runs.

In the case of BLCR, we speculate that the lack of reproducibility of times is due to some additional component of the Open MPI checkpoint-restart service, which cannot be interrupted by a network tear-down. The seemingly random random delays would be explained by whether a checkpoint is requested near the beginning or end of such a pending operation.

Note also that the times for DMTCP represent the total time spent by DMTCP in the InfiniBand plugin. No other part of the DMTCP code deals with any aspect of InfiniBand.

Number of processes	Runtime natively (s)	Runtime w/ DMTCP (s)	Ckpt (s)	Restart (s)
4	123.5	124.2	27.6	9.7
8	64.2	65.1	21.9	8.9
16	34.2	35.5	16.3	7.0

Table 6: Runtime overhead and Checkpoint-restart times for UPC FT B running under DMTCP

4.3 Checkpointing under UPC: A non-MPI Case Study

The study of checkpointing of Berkeley UPC is based on a port of the NAS parallel benchmarks at George Washington University [13]. Since that did not include a port of the LU benchmark, we switch to considering FT in this section.

The Berkeley UPC package was compiled to run natively over InfiniBand for this experiment, and it did not use MPI at all. The FT B NAS benchmark, as ported to run on UPC, was chosen because the port of NAS to UPC does not support the more communication-intensive LU benchmark. Table 6 shows that the native runtimes for FT B under UPC are comparable to the time for MPI in Figure 4. DMTCP total run-time overhead ranges from 4% down to less than 1%. We posit that the higher overhead of 4% is due to the extremely short running time in the case of 16 processes, and is explained by significant startup overhead, consistent with Table 2.

Note that BLCR could not be tested in this regime, since BLCR depends on the Open MPI checkpoint-restart service for its use in distributed computations.

4.4 Migrating InfiniBand to TCP sockets

Some traditional checkpoint-restart services, such as that that of Open MPI [15], offer the ability to checkpoint over one network, and restart on a second network. This section represents an early proof of principle that a similar technology could be implemented as a DMTCP plugin. This capability has the potential to support interactive debugging in a production environment, by copying checkpoint images from an InfiniBand-based production cluster to an Ethernet/TCP-based debug cluster. Note that since DMTCP is a user-space package, it does not require that the Linux kernels on the two clusters be the same.

4.4.1 Ping-pong

The IB2TCP plugin was tested with a communication-intensive ping-pong example InfiniBand program from the OFED distribution. In this case, a smaller development cluster was used, with 6-core Xeon X5650 CPUs and a Mellanox HCA for InfiniBand. Gigabit Ethernet was used for the Ethernet portion. Parameters were set to run over 100,000 iterations.

Environment	Transfer time (s)	Transfer rate (Gigabits/s)
IB (w/o DMTCP)	0.9	7.2
DMTCP/IB (w/o IB2TCP)	1.2	5.7
DMTCP/IB2TCP/IB	1.4	4.6
DMTCP/IB2TCP/Ethernet	65.7	0.1

Table 7: Transfer time variations using two nodes on InfiniBand versus Gigabit Ethernet hardware, with the DMTCP InfiniBand and IB2TCP plugin; 100,000 iterations of ping-pong, for a total transfer size of 819 MB

Table 7 presents the results. This is a worst case, since a typical MPI program is not as communication-intensive as the ping-ping test program. We hypothesize that the full transfer rate for Gigabit Ethernet was not achieved by this hardware/Linux combination.

Environment	Runtime (s)
IB (w/o DMTCP)	26.61
DMTCP/IB (w/o IB2TCP)	27.81
DMTCP/IB2TCP/IB	27.38
DMTCP/IB2TCP/Ethernet (restart on two nodes)	45.75
DMTCP/IB2TCP/Ethernet (restart on a single node)	66.34

Table 8: Runtime variations (no checkpoint-restart) for LU.A.2 using two nodes on InfiniBand versus Gigabit Ethernet hardware, with the InfiniBand and IB2TCP plugin.

4.4.2 NAS LU.A.2 Benchmark

The NAS LU.A benchmark was conducted on the the same MGHPCC cluster that was described in Section 4.1. Table 8 shows times for NAS LU.A.2 (2 MPI nodes) for migrating from InfiniBand to TCP using the IB2TCP plugin. The test is limited to two nodes due to a missing feature in DMTCP, the support for a function wrapper around the "poll" system call, used by Open MPI.

The combined InfiniBand and IB2TCP plugins do not add considerable overhead to the run time of the application at runtime. However, when the process is migrated from InfiniBand to TCP, the runtime increases drastically. A runtime overhead of 67% is seen for the restarted computation on on two TCP nodes. The runtime overhead further increases to 142% when the entire computation is restarted on a single node.

5. LIMITATIONS AND FUTURE WORK

In this section, we discuss some other limitations in the current implementation and our plans to overcome them in future.

5.1 Heterogeneous InfiniBand Architectures

Recall that DMTCP copies and restores all of user-space memory. In reviewing Figure 2, one notes that the user-space memory includes a low-level device-dependent driver in user space. If, for example, one checkpoints on a cluster partition using Intel/QLogic, and if one restarts on a Mellanox partition, then the Mellanox low-level driver will be missing. This presents a restriction for heterogeneous computing centers in the current implementation.

There are two possible alternative implementations as described next. First, it is possible to implement a generic "stub" driver library, which can then dispatch to the appropriate device-dependent library. Second, it is possible to force the InfiniBand library to re-initialize itself by restoring the pre-initialization version of the InfiniBand library data segment, instead of the data segment as it existed just prior to checkpoint. This will cause the InfiniBand library to appear to be uninitialized, and it will re-load the appropriate device-dependent library.

5.2 Out-of-Sync Send/Recv Completions

The InfiniBand hardware may post completions to the sender and receiver at slightly different times. Thus, after draining the completion queue, the plugin waits for a fraction of a second, and then drains the completion queue one more time. This is repeated until no completions are seen in the latest period. Thus, correctness is affected only if the InfiniBand hardware posts corresponding completions relatively far apart in time, which is highly unlikely. (Note that this situation occurs in two cases: InfiniBand send-receive mode; and InfiniBand RDMA mode for the special case of ibv_post_send while setting the immediate data flag.)

In a related issue, when using the immediate data flag or the inline flag in the RDMA model, a completion is posted only on the receiving node. These flags are intended primarily for applications that send small messages. Hence, the current implementation sleeps for a small amount of time to ensure that such messages complete. A future implementation will use the DMTCP coordinator to complete the bookkeeping concerning messages sent and received, and will continue to wait if needed.

5.3 Unreliable Connections

The current implementation does not support unreliable connections (the analog of UDP for TCP/IP). Most target applications do not use this mode, and so this is not considered a priority. For a potential solution, one could add wrappers for InfiniBand functions that provide unreliable connections and adjust the draining logic.

5.4 Virtual Id Conflicts After Restart

In typical MPI implementations, memory region keys (rkey), queue pair numbers (qp_num), and local ids (lid) are all exchanged out-of-band. Since virtualized ids are passed to the target application, it is the virtualized ids that are passed out-of-band. The remote plugin is then responsible for translating the virtual ids to the real ids known to the InfiniBand hardware, on later InfiniBand calls.

The current implementation ensures that this is possible, and that there are no conflicts prior to the first checkpoint, as described in Section 3.2. In typical InfiniBand applications, queue pairs are created only during startup, and so all rkeys, qp_nums and lids will be assigned prior to the first checkpoint. However, it is theoretically possible for an application to create a new queue pair, memory region, or to query its local id after the first restart.

The current implementation assigns the virtual id to be the same as the real id at the time of the initial creation of the InfiniBand object. (After restart, the InfiniBand hardware may assign a different real id, but the virtual id for that object will remain the same.) If an object is created after restart, the real id assigned by InfiniBand may be the same as for an object created prior to checkpoint. This would create a conflict of the corresponding virtual ids.

Two solutions to this problem are possible. The simplest is to use DMTCP's publish-subscribe feature to generate globally unique virtual rkeys, and update a global table of virtual-to-real rkeys. In particular, one could use the existing implementation before the first checkpoint, and then switch to a publish-subscribe implementation after restart. A second solution is to choose the virtual rkeys in a globally unique manner, similarly to the globally unique protection domain ids of the current plugin.

6. RELATED WORK

In the case of distributed computations over TCP (e.g., over Ethernet), several distributed checkpointing approaches have been proposed [7, 17, 22, 21, 27]. Unfortunately, those solutions do not extend to supporting the InfiniBand network. Other solutions for distributed checkpointing are specific to a particular MPI implementation [4, 11, 15, 16, 20, 23, 25, 26]. These MPI-based checkpoint-restart services "tear down" the InfiniBand connection, after which a single-process checkpoint-restart package can be applied.

The implementation described here can be viewed as interposing a shadow device driver between the end user's code and the true device driver. This provides an opportunity to virtualize the fields of the queue pair struct seen by the end user code. Thus, the InfiniBand driver is modelled without the need to understand its internals. This is analogous to the idea of using a shadow kernel device by Swift et al. [28, 29]. In that work, after a catastrophic failure by the kernel device driver, the shadow device driver was able to take over and place the true device driver back in a sane state. In a similar manner, restarting on a new host with a new HCA Adapter can be viewed as a catastrophic failure of the InfiniBand user-space library. Our virtual queue pair along with the log of pending posts and modifications to the queue pair serves as a type of shadow device driver. This allows us to place back into a sane state the HCA hardware, the kernel driver and the device-dependent user-space driver.

This work is based on DMTCP (Distributed MultiThreaded CheckPointing) [1]. The DMTCP project began in 2004 [6, 7]. With the development of DMTCP versions 2.x, it has emphasized the use of plugins [8] for more modular maintainable code.

Currently, BLCR [14] is widely used as one component of an MPI dialect-specific checkpoint-restart service. This design is fundamentally different, since an MPI-specific checkpoint-restart service calls BLCR, whereas DMTCP transparently invokes an arbitrary MPI implementation. Since BLCR is kernel-based, it provides direct support only on one computer node. Most MPI dialects overcome this in their checkpoint-restart service by disconnecting any network connections, delegating to BLCR the task of a single-node checkpoint, and then reconnect the network connection. Among the MPI implementations using BLCR are Open MPI [16] (CRCP coordination protocol), LAM/MPI [25], MPICH-V [4], and MVAPICH2 [11]. Other MPI implementations provide their own analogs [11, 23, 25, 26]. In some cases, an MPI implementation may support an *application-initiated* protocol in combination with BLCR (such as SELF [16, 25]). For application-initiated checkpointing, the application writer guarantees that there are no active messages at the time of calling for a checkpoint.

Some recommended technical reports for further reading on the design of InfiniBand are [2, 19], along with the earlier introduction to the C API [30]. The report [19] was a direct result of the original search for a clean design in checkpointing over InfiniBand, and [20] represents a talk on interim progress.

In addition to DMTCP, there have been several packages for transparent, distributed checkpoint-restart of applications running over TCP sockets [17, 21, 22, 27]. The first two packages ([17] and [22, 21]) are based on the Zap package [24].

The Berkeley language Unified Parallel C (UPC) [10] is an example of a PGAS language (Partitioned Global Address Space). It runs over GASNet [3] and evolved from experience with earlier languages for DSM (Distributed Shared Memory).

7. CONCLUSION

A new approach to distributed transparent checkpoint-restart over InfiniBand has been demonstrated. This direct approach accommodates computations both for MPI and for UPC (Unified Parallel C). The approach uses a mechanism similar to that of a shadow device driver [28, 29]. In tests on the NAS LU parallel benchmark, a run-time overhead of between 0.8% and 1.7% is found on a computation with up to 2,048 MPI processes. Startup overhead is up to 13 seconds, and grows as the cube root of the number of MPI processes. Checkpoint times are roughly proportional to the total size of all checkpoint images on a single computer node. In one example with 512 MPI processes, checkpoint times varied by a factor of 6.5 (from 232 seconds to 36 seconds), depending on whether checkpoint images were written to a local disk or to a faster, Lustre-based back-end. In both cases, there were 16 MPI processes per node, and a total of approximately 5.8 GB per node was written.

The new approach also provides a viable checkpoint-restart mechanism for running UPC natively over InfiniBand — something that previously did not exist. Finally, an IB2TCP plugin was shown for migrating from InfiniBand to TCP, demonstrating that the plugin design is compatible with a an interconnection-agnostic feature, similar to that of the Open MPI checkpoint restart service [15]. Since the DMTCP plugin approach is purely user-space, it has the added benefit of supporting a destination cluster with a different Linux kernel image.

Acknowledgment

We are grateful to facilities provided at several institutions with which to test over a variety of configurations. We would like to thank: L. Shawn Matott (U. of Buffalo, development and benchmarking facilities); Henry Neeman (Oklahoma University, development facilities); Larry Owen and Anthony Skjellum (the University of Alabama at Birmingham, facilities based on NSF grant CNS-1337747); and the Massachusetts Green High Performance Computing Center (facilities for scalability testing). Dotan Barak provided helpful advice on the implementation of OpenFabrics InfiniBand. Jeffrey M. Squyres and Joshua Hursey provided helpful advice on the interaction of Open MPI and InfiniBand. Artem Polyakov provided advice on using the DMTCP batch-queue (resource manager) plugin. We also benefited from valuable comments and feedback from the reviewers. We are also grateful for help from Bogdan Nicolae in shepherding the paper.

8. REFERENCES

[1] J. Ansel, G. Cooperman, and K. Arya. DMTCP: Scalable user-level transparent checkpointing for cluster computations and the desktop. In *Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS-09, systems track)*. IEEE Press, 2009. published on CD; version also available at `http://arxiv.org/abs/cs.DC/0701037`; software available at `http://dmtcp.sourceforge.net`.

[2] T. Bedeir. Building an RDMA-capable application with IB Verbs. Technical report, `http://www.hpcadvisorycouncil.com/`, August 2010. `http://www.hpcadvisorycouncil.com/pdf/building-an-rdma-capable-application-with-ib-verbs.pdf`.

[3] D. Bonachea. GASNet specification, v1.1. Technical report UCB/CSD-02-1207, U. of California, Berkeley, October 2002. `http://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-02-1207.pdf`.

[4] A. Bouteiler, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. MPICH-V project: a multiprotocol automatic fault tolerant MPI. *International Journal of High Performance Computing Applications*, 20:319–333, 2006.

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

[6] G. Cooperman, J. Ansel, and X. Ma. Adaptive checkpointing for master-worker style parallelism (extended abstract). In *Proc. of 2005 IEEE Computer Society International Conference on Cluster Computing*. IEEE Press, 2005. conf. proc. on CD.

[7] G. Cooperman, J. Ansel, and X. Ma. Transparent adaptive library-based checkpointing for master-worker style parallelism. In *Proceedings of the 6^{th} IEEE International Symposium on Cluster Computing and the Grid (CCGrid06)*, pages 283–291, Singapore, 2006. IEEE Press.

[8] DMTCP team. Tutorial for DMTCP plugins, accessed Apr., 2014. `http://dmtcp.sourceforge.net/api.html`.

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

[10] T. El-Ghazawi and F. Cantonnet. UPC performance and potential: A NPB experimental study. In *Proc. of the 2002 ACM/IEEE Conference on Supercomputing*, Supercomputing '02, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[11] Q. Gao, W. Yu, W. Huang, and D. K. Panda. Application-transparent checkpoint/restart for MPI programs over InfiniBand. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 471–478, Washington, DC, USA, 2006. IEEE Computer Society.

[12] R. Garg, K. Sodha, Z. Jin, and G. Cooperman. Checkpoint-restart for a network of virtual machines. In *Proc. of 2013 IEEE Computer Society International Conference on Cluster Computing*. IEEE Press, 2013.

[13] GWU High-Performance Computing Laboratory. UPC NAS parallel benchmarks. `http://threads.hpcl.gwu.edu/sites/npb-upc`, accessed Jan., 2014, 2014.

[14] P. Hargrove and J. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters. *Journal of Physics Conference Series*, 46:494–499, Sept. 2006.

[15] J. Hursey, T. I. Mattox, and A. Lumsdaine. Interconnect agnostic checkpoint/restart in Open MPI. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 49–58, New York, NY, USA, 2009. ACM.

[16] 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 21^{st} IEEE International Parallel and Distributed Processing Symposium (IPDPS) / 12^{th} IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems*. IEEE Computer Society, March 2007.

[17] G. Janakiraman, J. Santos, D. Subhraveti, and Y. Turner. Cruz: Application-transparent distributed checkpoint-restart on standard operating systems. In *Dependable Systems and Networks (DSN-05)*, pages 260–269, 2005.

[18] W. Jiang, J. Liu, H.-W. Jin, D. K. Panda, W. Gropp, and R. Thakur. High performance MPI-2 one-sided communication over InfiniBand. In *CCGRID*, pages 531–538, 2004.

[19] G. Kerr. Dissecting a small InfiniBand application using the Verbs API. arxiv:1105.1827v2 [cs.dc] technical report, arXiv.org, May 2011.

[20] G. Kerr, A. Brick, G. Cooperman, and S. Bratus. Checkpoint-restart: Proprietary hardware and the 'spiderweb API', July 8–10 2011. talk: abstract at `http://recon.cx/2011/schedule/events/112.en.html`; video at `https://archive.org/details/Recon_2011_Checkpoint_Restart`.

[21] O. Laadan and J. Nieh. Transparent checkpoint-restart of multiple processes for commodity clusters. In *2007 USENIX Annual Technical Conference*, pages 323–336, 2007.

[22] O. Laadan, D. Phung, and J. Nieh. Transparent networked checkpoint-restart for commodity clusters. In *2005 IEEE International Conference on Cluster Computing*. IEEE Press, 2005.

[23] P. Lemarinier, A. Bouteillerand, T. Herault, G. Krawezik, and F. Cappello. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 115–124, Washington, DC, USA, 2004. IEEE Computer Society.

[24] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Prof. of 5^{th} Symposium on Operating Systems Design and Implementation (OSDI-2002)*, 2002.

[25] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, 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*, 19(4):479–493, 2005.

[26] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, 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*, 19(4):479–493, 2005.

[27] O. O. Sudakov, I. S. Meshcheriakov, and Y. V. Boyko. CHPOX: Transparent checkpointing system for Linux clusters. In *IEEE Int. Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, pages 159–164, 2007.

[28] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[29] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Trans. Comput. Syst.*, 24(4):333–360, Nov. 2006.

[30] B. Woodruff, S. Hefty, R. Dreier, and H. Rosenstock. Introduction to the InfiniBand core software. In *Proceedings of the Linux Symposium (Volume Two)*, pages 271–282, July 2005.