Snapify: Capturing Snapshots of Offload Applications on Xeon Phi Manycore Processors

Arash Rezaei[1], Giuseppe Coviello[2], Cheng-Hong Li[2], Srimat Chakradhar[2], Frank Mueller[1]

[1] Department of Computer Science, North Carolina State University, Raleigh, NC.
[2] Computing Systems Architecture Department, NEC Laboratories America, Inc., Princeton, NJ.

ABSTRACT

Intel Xeon Phi coprocessors provide excellent performance acceleration for highly parallel applications and have been deployed in several top-ranking supercomputers. One popular approach of programming the Xeon Phi is the *offload* model, where parallel code is executed on the Xeon Phi, while the host system executes the sequential code. However, Xeon Phi's Many Integrated Core Platform Software Stack (MPSS) lacks fault-tolerance support for offload applications. This paper introduces Snapify, a set of extensions to MPSS that provides three novel features for Xeon Phi offload applications: checkpoint and restart, process swapping, and process migration. The core technique of Snapify is to take consistent process snapshots of the communicating offload processes and their host processes. To reduce the PCI latency of storing and retrieving process snapshots, Snapify uses a novel data transfer mechanism based on remote direct memory access (RDMA). Snapify can be used transparently by single-node and MPI applications, or be triggered directly by job schedulers through Snapify's API. Experimental results on OpenMP and MPI offload applications show that Snapify adds a runtime overhead of at most 5%, and this overhead is low enough for most use cases in practice.

Categories and Subject Descriptors

D.4.5 [**Reliability**]: Checkpoint/Restart—*Fault Tolerance*

Keywords

Checkpoint and restart, process swapping, process migration, snapshot, fault tolerance, Xeon Phi, coprocessor, system software

1. INTRODUCTION

A Xeon Phi coprocessor has up to 61 cores, connected by a high-speed on-chip interconnect. Each core supports four hardware threads, and has a 512-bit wide vector unit to execute SIMD instructions. The coprocessor has its own physical memory of 8/16GB, which can be accessed with an aggregate memory bandwidth of 352GB/s. Both the coprocessor and the host system run their own operating systems. The host system and the coprocessor do not share a common memory space, and the two are physically connected by the PCIe bus.

Xeon Phi's software stack (MPSS) supports two programming models. In the *offload* programming model, highly parallel, code segments are executed on Xeon Phi, while the host system executes the sequential code. On the other hand, the *native* programming model allows users to execute their applications entirely on the Xeon Phi coprocessor. MPSS provides high-level language support and a modified Linux OS on Xeon Phi to facilitate both programming models. As a result of its performance acceleration for highly parallel applications and ease of programming, Xeon Phi coprocessors have been deployed in HPC systems, including several top-ranking supercomputers [14].

Using coprocessors like Xeon Phi in HPC systems, however, compounds the problem of the increasing failure rate due to the system's growing size and complexity [6]. A recent study on a GPU-based supercomputer shows a failure rate of 13 hours on average [32]. It is projected that the mean time between failure of HPC systems will continue to shrink [6, 33]. Therefore programmers must adopt certain fault tolerance mechanism for their applications.

Checkpoint and restart is a fault-tolerance technique widely used in HPC systems. Such a method periodically takes a snapshot of the application state and saves it on persistent storage. In case of an error, the application can be restored to a former saved state. The popular checkpoint and restart tool BLCR provides application-transparent checkpoint and restart support [11]. It can take a snapshot of the entire process state in both the user and the kernel space, with no modification to the application code. It has been integrated with MPI to provide distributed checkpoint and restart for MPI applications running on a cluster [31].

Although Xeon Phi's software stack is designed to ease the programming effort, its support of fault tolerance is inadequate. MPSS uses BLCR to support checkpoint and restart of native applications. BLCR can either save the snapshot of a native process on Xeon Phi to the host's file system through Network File System (NFS), or to Xeon Phi's own local file system. However, both of these two storage choices have limitations. Saving directly to the host file system through NFS incurs high data transfer latency on PCIe bus. And because Xeon-Phi does not have any directly accessible storage and uses a RAM-based file system, a locally saved snapshot on Xeon Phi's own file system competes the physical memory space with active processes, including the native process whose snapshot is to be saved.

A more severe problem is that MPSS has no fault-tolerance support for offload applications. Given the fact that even a single-node offload application involves the participation of a number of host and coprocessor processes that use proprietary communication li-

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

http://dx.doi.org/10.1145/2600212.2600215.

braries in MPSS to exchange messages, it is not surprising that none of the existing distributed checkpoint and restart tools like BLCR or DMTCP [1] can be used. This is because these tools do not consider the communication between the coprocessor processes and the host processes.

In addition MPSS also lacks sufficient support for process migration among Xeon Phi coprocessors. When compute nodes are equipped with multiple coprocessors, process migration can benefit coprocessor load balancing and fault resiliency. For example, a job scheduler may decide to migrate an offload or native process from a heavily-loaded coprocessor to a lightly-loaded one to increase job turnaround time [3]. Moreover, by using fault prediction methods [30], it is possible to avoid imminent coprocessor failures by proactively migrating processes to other healthy coprocessors.

The limited physical memory on the Xeon Phi coprocessor also restricts the sharing of the coprocessor among multiple applications. Previous studies have shown that allowing multiple user applications to share coprocessor like GPU or Xeon Phi can significantly benefit system utilization and job turnaround time [3, 5, 25, 29]. However, Xeon Phi OS's own page swapping mechanism is a poor solution to overcome the capacity limit of Xeon Phi's physical memory for multiprocessing. First, as shown by the study in [5], Xeon Phi OS's swap uses the host's file system as its secondary storage. Swapping in and out memory pages between the host and the coprocessor incurs high data transfer latency. Second, many offload applications use pinned memory buffers to allow fast remote direct memory access (RDMA) between the host and the coprocessor's memory. Pinned memory pages cannot be swapped out by OS. As a result, the size of Xeon Phi's physical memory puts a hard limit on the number of processes that can concurrently run on the coprocessor.

Contributions. To address these shortcomings of MPSS, we created Snapify, a set of extensions to MPSS that captures snapshots of offload applications on Xeon Phi. We make the following specific contributions:

- We propose Snapify, an application-transparent, coordinated approach to take *consistent* snapshots of host and coprocessor processes of an offload application. We believe this is the first proposal that correctly and transparently captures a process-level snapshot of offload applications on many-core processors like the Xeon Phi.
- We use Snapify to implement three new capabilities for offload applications: application-transparent checkpoint and restart, process migration, and process swapping. Again, to the best of our knowledge, this is the first proposal that provides such capabilities for offload applications on Xeon Phi.
- We propose a fast, remote file access service based on RDMA that speeds up the storage and retrieval of snapshots (of both offload and native applications) from the host's file system.
- We evaluate Snapify on several OpenMP and MPI benchmarks. Our results show that Snapify imposes negligible overhead in normal execution of offload applications (less than 5%), and the overheads due to the capture of snapshots in checkpoint and restart, process swap, and process migration are small enough to make it practical to use Snapify for a variety of offload applications.

This paper is organized as follows. Section 2 gives a concise background of Xeon Phi's programming model. Section 3 discusses Snapify's design challenges. Section 4 describes Snapify's internal design and its API. Section 5 shows how Snapify's API can be used to implement checkpoint and restart, process swapping, and process migration. Section 6 introduces our RDMA-based remote file access service for snapshot storage. Section 7 presents the experimental results. Related work is discussed in Section 8.

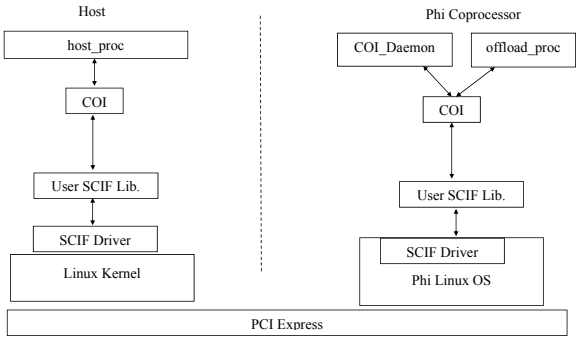


Figure 1: The software architecture of host & Xeon Phi.

2. BACKGROUND

In this section we briefly describe the offload programming model and its implementation in MPSS.

Xeon Phi's software stack MPSS consists of a hierarchy of runtime libraries that execute on the host and the coprocessor [17]. These libraries hide complex, low-level communication mechanisms away and present a simple, high-level offload programming semantics to the programmers. The software stack also has a modified Linux OS that is run on Xeon Phi. The Xeon Phi OS has its own file system (built on RAM disk using Xeon Phi's own physical memory), virtual memory management, and an optional page-swap mechanism that uses the host's file system as secondary storage.

To program an offload application a programmer uses special compiler directives to delineate regions of code to be executed on Xeon Phi coprocessors. In particular, the pragma "offlaod" is used to mark an offload region with language-specific scoping constructs (curly braces in C, and "begin" and "end" in Fortran). In particular, the offload pragma can have additional data transfer clauses "in" and "out", specifying the input and output buffers of the offload region. Input buffers specified in the "in" clauses are transferred from the host memory to the Xeon Phi coprocessor through the PCIe bus prior to the execution of the offload region, while output buffers specified in the "out" clauses are transferred back from the Xeon Phi coprocessor to the host memory following the completion of the offload region. More details of Xeon Phi programming can be found in [20].

The Xeon Phi compiler generates one binary for the host processor and one binary for the Xeon Phi coprocessor for an offload application. The host binary is an executable, while the Xeon Phi binary is a dynamically loadable library. The compiler translates each offload region as a function to be executed on the Xeon Phi coprocessor and saves the generated code in the Xeon Phi binary. For each offload region, in the host binary the compiler also generates a function call to the lower-layer runtime libraries, which coordinate the data transfer between the host and the Xeon Phi coprocessor and initiate the "remote" procedure call of the offload function.

The execution of an offload application to be accelerated by Xeon Phi coprocessors involves a minimum of three processes, as reported in Fig. 1. The three processes are a host process (`host_proc`) running on the host processor, an offload process (`offload_proc`) running on the Xeon Phi, and a service named Coprocessor Offload

Infrastructure (COI) daemon (`coi_daemon`), also running on the Xeon Phi. After a user launches the application (`host_proc`), the host process requests the COI daemon to launch a process on the Xeon Phi. Then the host process copies the Xeon Phi binary to the coprocessor. The binary contains offload functions to be executed on the coprocessor, and these functions are dynamically loaded into `offload_proc`'s memory space. The execution of an offload function is done by a server thread in the offload process. To execute a function, the host process sends a request to the server thread to run the specified function. Once the function completes on Xeon Phi, the server thread will send the function's returned value back to the host process.

Prior to the execution of an offload function the host process also transfers the input data needed by the offload region to the offload process's memory space. The host process also receives data, if any, that is generated by the offload process.

MPSS provides different levels of abstractions to facilitate the communications between the host process and the processes running on the Xeon Phi. The COI library is an upper level library offering APIs for a host process to perform process control and remote function calls on a Xeon Phi coprocessor [16, 17]. It also allows a host process to create buffers, called COI buffers. The host process can use COI's API to transfer data between the host process and the buffers allocated in the offload process. The COI library in turn uses the lower level Symmetric Communications Interface (SCIF) library to accomplish the real message exchanges and data transfers between the host process and the offload process [18].

The COI library on the Xeon Phi coprocessor manages the memory space used by COI buffers. A COI buffer is composed of one or more files that are memory mapped into a contiguous region. These files are called local store. COI buffers can be created and destroyed through COI functions, while the files created to be used by COI buffers are persistent until the offload process terminates.

SCIF provides two types of APIs that allow two processes in different memory space to communicate. The first type is message-based. The processes use `scif_send()` and `scif_receive()` to send and receive data. The second type offers remote direct memory access (RDMA) functions to speed up the data transfer. To use SCIF's RDMA functions to transfer a buffer, a process first registers the buffer's virtual memory address using `scif_register()` function. The function returns an offset address that can be used in SCIF's RDMA functions: `scif_vreadfrom()`, `scif_readfrom()`, `scif_vwriteto()`, and `scif_writeto()`. In fact, the COI library uses SCIF's RDMA functions to copy data in COI buffers between the host and coprocessors.

Each Xeon Phi device runs one COI daemon process to coordinate the execution of offload processes and the corresponding host processes. The COI daemon maintains SCIF connections with each active host process that uses COI library to offload part of its computation to Xeon Phi. The connections are used to communicate the process control messages between host processes and the COI daemon. For example, the COI daemon launches new offload processes upon requests from applications on the host. If the host process exits, the daemon will terminate the offload process and clean up the temporary files used by the offload process.

3. CHALLENGES

There are several challenges that must be overcome to capture a process-level snapshot of an offload application, and subsequently restart the application from the snapshot. These challenges arise from the distributed nature of an offload application on a Xeon Phi server, and Xeon Phi's own software stack. Below we summarize these new challenges.

Capturing consistent, distributed snapshots. Since the execution of an offload application on Xeon Phi involves multiple communicating processes that do not share memory or a global clock, it is necessary to ensure that the snapshots of the host and offload processes form a *consistent* global state [7, 22, 31]. In the simplest case, an offload application has three processes: a process on the host and two processes (an offload process, and `coi_daemon` process) on the Xeon Phi. A global state consists of states of the three processes, as well as the state of the communication among these processes. A consistent global state satisfies two properties. First, it is possible to reach this state during the normal operation of the application. Second, it is possible to correctly restart the processes and resume the execution of the application. As a counter example, if the host snapshot is taken before the host process sends a message to the coprocessor, and the snapshot of the offload process is taken after the receipt of the message, then this pair of snapshots does not form a consistent global state, and the global state cannot be applied to resume the application. Due to the distributed nature of Xeon Phi's offload model, snapshots obtained by just using an existing single-process checkpoint tool like BLCR [11] or MTCP [27] cannot form a consistent global state.

The states of communication channels are also part of the global state. Since we cannot take a snapshot of the physical state of a PCIe bus and the internal hardware state of its controllers, we must make sure all communication channels between the involved processes are drained before local snapshots are taken.

Xeon Phi-specific communication libraries. A Xeon Phi offload application uses its own proprietary communication libraries (i.e. COI and SCIF) for inter-process communication. Therefore the existing cluster checkpoint tools designed for applications based on communication libraries like MPI [31] or TCP/IP [1, 19, 28] cannot be applied to Xeon Phi offload applications.

Dealing with distributed states. The states that are distributed among the processes participating in the execution of an offload application may get disturbed by the action of taking a snapshot, swapping out, or restarting the offload process. For example, the COI daemon is responsible for monitoring the status of both the host and the offload process. If an offload process is terminated due to being swapped out or being migrated, the `coi_daemon` will assume that the offload process has crashed and (incorrectly) mark the process as terminated. On the other hand, when an offload process is restarted from a snapshot, the `coi_daemon` needs to be brought into the picture again to monitor the restarted host and offload process.

Storing and retrieving snapshots. The split of Xeon Phi's physical memory between the file system and system memory puts a serious restriction on how the snapshot of an offload process can be stored. A naive solution that saves a snapshot on Xeon Phi's local file system cannot be applied to any process, either native or offload, whose memory footprint exceeds 50% of the Xeon Phi's physical memory.[1] The same restriction also applies when we attempt to restart a process from a snapshot stored on Xeon Phi's local file system. Even if the snapshot and the process fit in the Xeon Phi's physical memory, the memory used to store the snapshot is unavailable to other offload or native applications on the Xeon Phi, resulting in a decrease in the number of applications that can run concurrently or even a crash of some applications due to lack of memory. Therefore, it is desirable to store and retrieve snapshots from the host's file system.

[1]The actual limit is less than half, since the system files and the OS use a small portion of the memory.

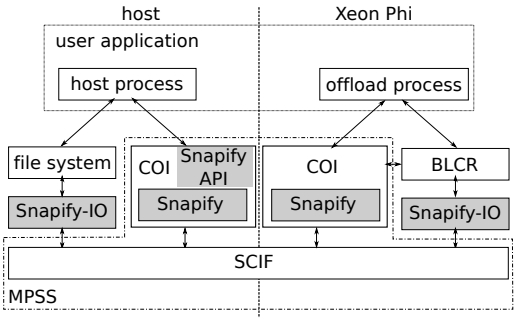


Figure 2: The architecture of Snapify.

```
typedef struct {   char* m_snapshot_path;
                   sem_t m_sem;
                   COIProcess* m_process;
                   }snapify_t;

void snapify_pause(snapify_t* snapshot);

void snapify_capture(snapify_t* snapshot,
                   bool terminate);

void snapify_wait(snapify_t* snapshot);

void snapify_resume(snapify_t* snapshot);

void snapify_restore(snapify_t* snapshot,
                   int device);
```

Table 1: Snapify API.

Saving data private to an offload process. A snapshot of an offload process contains the offload process's own private data. Unlike GPU-based coprocessor systems, an offload process on Xeon Phi is a full-blown Linux process. The offload process has host-allocated COI buffers, and its own private data that may not be visible to the host system (for example, stacks of threads or data regions that are either statically or dynamically allocated through standard system calls like `malloc()` by the functions in the offload process are not visible to the host). The offload-private data may persist across several offload regions. Therefore the strategy of only saving the host-controlled memory regions in a snapshot, as proposed for GPU-accelerated applications in CheCL [34] and CheCUDA [35], is not suitable for Xeon Phi's offload applications.

4. SNAPIFY

Fig. 2 shows the positioning of Snapify technologies in the software stack of the host and the Xeon Phi. Intel's MPSS includes COI (which contains the COI daemon) and SCIF. The file system is part of the Linux OS on the host, and BLCR is the open-source checkpoint and restart framework. Applications offload computations to the Xeon Phi using the COI library, which has a component that executes on the host and a component that executes on the Xeon Phi. Key technologies in Snapify are implemented as modifications to the COI library and the COI daemon, and as an independent user-level library called *Snapify-IO*. The implementation does not change the COI programming interface, and thus is transparent to user applications. Sections 4.1, 4.2, and 4.3 discuss our techniques to create process-level snapshots of the offload application, restore the execution of the application from a snapshot, and resume the application after a snapshot has been taken, respectively. Snapify-IO, to be described in Section 6, is a very efficient mechanism that we designed to store and retrieve snapshots from the host file system using SCIF's RDMA functions.

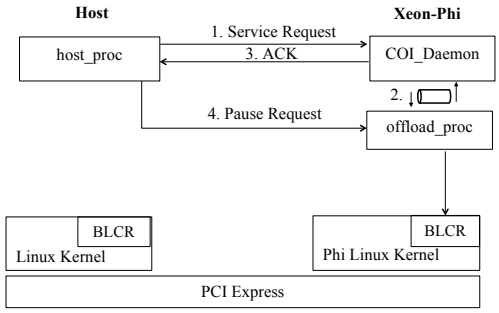


Figure 3: Overview of taking a snapshot.

Snapify provides a simple API that is used to capture snapshots of an offload process and restore the offload process from snapshots. The API is summarized in Table 1. Snapify's API defines a C structure and five functions. The functions are called by a host process to capture a snapshot of the offload process (`snapify_pause()` and `snapify_capture()`), to resume the communication and the partially-blocked execution of the host process and the offload process after a snapshot is taken (`snapify_resume()`), to restore an offload process from a snapshot (`snapify_restore()`), and to wait for a non-blocking API function to complete (`snapify_wait()`). All functions except `snapify_capture()` are blocking calls. The structure is used to pass parameters and to receive results of the functions. The API functions are detailed in the rest of this section.

4.1 Taking Snapshots

Taking a snapshot of an offload process involves the host process on the host and the COI daemon and the offload process on each of the coprocessors installed in a Xeon Phi server. Although our approach handles multiple Xeon Phi coprocessors in a server, for simplicity we assume there is only one Xeon Phi coprocessor in the following discussions. Therefore we consider the case of three involved processes: the host process, the COI daemon, and the offload process.

The snapshot process is accomplished in two separate steps. In step one, all of the communications between the host process and the offload process are stopped, and the channels are drained. In step two, a snapshot of the offload process is captured and saved in the file system of the host. These two steps are implemented by `snapify_pause()` and `snapify_capture()`, respectively.

Pause. To pause the communications between the host process and the offload process, the host process calls `snapify_pause()` and passes the handle of the offload process (`COIProcess` in the structure) to `snapify_pause()`. A directory structure in the host's file system for storing the files of a snapshot is also needed by `snapify_pause()` (and `snapify_capture()`). The path to the directory is passed to `snapify_pause()` through the member variable `m_snapshot_path`. In the first step of `snapify_pause()` it saves the copies of the runtime libraries from the host's file system needed by the offload process to the snapshot directory.[2]

Fig. 3 shows the interactions between the host process, the COI daemon, and the offload process that are triggered by

[2]MPSS maintains copies of the runtime libraries on the host file system. Therefore as an optimization we do not copy the libraries of the offload process from the coprocessor back to the host system.

`snapify_pause()`. Function `snapify_pause()` first sends a snapify-service request to the COI daemon (step 1 in Fig. 3). The daemon then creates a UNIX pipe to the offload process, and writes the pause request to the offload process. Next the daemon signals the offload process, triggering the signal handler in the offload process to the pipe and send an acknowledgement back to the daemon through the pipe (step 2). The daemon then relays the acknowledgement back to the host process (step 3). At this point all parties (the host process, the offload process, and the COI daemon) have agreed to pause the communications and drain the communication channels.

The COI daemon is chosen as the coordinator of Snapify's pause procedure. This is because there is one daemon per coprocessor, and each daemon listens to the same fixed SCIF port number. It services pause requests that may come from different host processes. It also maintains a list of active requests. Upon receiving a new pause request, the daemon adds an entry to the list. The entry is removed after the pause request is serviced.

To avoid any interference with its regular tasks, the daemon uses a dedicated Snapify monitor thread to oversee the progress of the pause procedure. Whenever a request is received and no monitor thread exists, the daemon creates a new monitor thread. The monitor thread keeps polling the pipes to the offload processes on the list of active pause requests for status updates. The monitor thread exits when there is no more active pause request in the list.

Following the initial handshake `snapify_pause()` sends a pause request to the offload process (step 4 in Fig. 3) to drain the communication channels. The draining needs the collaboration between the host process, the COI daemon, and the offload process, and will be discussed in more detail shortly. It is a necessary step to ensure that the snapshots form a consistent global state (see Section 3). During the draining process some of the threads in the host process and the offload process spawned by the COI library are blocked. The blocking of these threads keeps the SCIF channels from being used until `snapify_resume()` is called. These threads are responsible for sending and receiving COI commands, COI events, and the COI logs.

After the SCIF channels are quiesced, the offload process will save its local store (memory allocated in the offload process's memory space for storing data in COI buffers) to the host's snapshot directory. This operation does not use any existing SCIF channels between the host process and the offload process. Saving the local store and the snapshot will be discussed in detail in Section 6.

At the end of `snapify_pause()` all of the SCIF channels between the host process, the COI daemon, and the offload process become empty. To notify the host process that the pause has completed, the offload process sends a message through the pipe to the COI daemon, and the COI daemon informs the host process that the offload process has completed the pause operation. After this the offload process waits on the pipe to wait for the next request from the the host process. The next request is either a capture or a resume request, which will be discussed later.

We now give more details on how `snapify_pause()` drains the SCIF communication channels. We first classify all SCIF communication use instances in the COI runtime to four different cases.

1. The host process, the offload process, and the COI daemon exchange messages when an offload process is created and before it is destroyed. These messages carry information regarding process creation, confirmation, request for termination, and etc.

2. The host process and the offload process use one SCIF channel to perform RDMA transfers of the data in COI buffers. The RDMA transfers are carried out by `scif_writeto()` and `scif_readfrom()` functions.

3. The host process, the COI daemon, and the offload process have several pairs of client-server threads. Each server thread serves only one client thread. It handles the incoming commands in a sequential fashion. The commands are sent by the client thread through a dedicated SCIF channel.

4. The execution of an offload function is also implemented by a client-server model. In order to take a snapshot during the execution of an offload function, however, we treat this case separately. Our method handles both synchronous and asynchronous offload executions.

   Fig. 4 reports the client-server model that executes offload functions. When a thread *Pipe_Thread1* in the host process enters an offload region, it sends a run function request to the server thread *Pipe_Thread2* in the offload process. After sending this request, *Pipe_Thread1* performs a blocking receive to wait for the result, while *Pipe_Thread2* calls the function and sends the function's return value back to *Pipe_Thread1* of the host process.

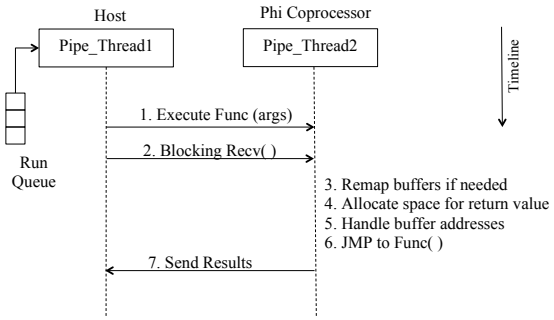


Figure 4: Executing an offload function.

For each of the four use cases of SCIF we develop a method to drain the SCIF communication channels. For case 1, we declare the initialization and cleanup code regions of creating and terminating offload processes as critical regions, protected by a mutex lock. When `snapify_pause()` is called, it will try to acquire the lock. If a thread is executing the code in a critical region, `snapify_pause()` will be blocked until the thread leaves the critical region. On the other hand, once `snapify_pause()` holds the lock, any other thread that attempts to enter these critical regions will be blocked.

For case 2 we delay any snapshot attempt when a RDMA transfer is active. Similar to the case above, we protect the call sites of SCIF's RDMA functions with mutex locks.

To handle a SCIF channel in the client-server model of case 3, we take advantage of the sequential nature of the client-server implementation in COI. We added a new "shutdown" request to the server's request handling routine. This request is only issued by `snapify_pause()`, and is used as a special marker that indicates no more commands will follow until `snapify_resume()` is called. To send the shutdown request, `snapify_pause()` first tries to acquire the lock that is used by the client thread to protect the communication channel. After `snapify_pause()` acquires the lock, the client thread will not be able to send any more requests. The lock that is used by a client thread will only be released in `snapify_resume()`. After acquiring the lock `snapify_pause()` sends the shutdown request to the server.

The pause function will not continue until all of the server threads in the host process, the COI daemon, and the offload process receives a shutdown command. This ensures that the SCIF channels used between the client and the server threads stay empty until `snapify_resume()`.

For case 4 to drain the SCIF channel used by *Pipe_Thread1* and *Pipe_Thread2* we made a number of changes to the implementation of the COI pipeline. First we transformed the two send functions in Step 1 and 7 to be blocking calls. We then placed these two send functions in two separate critical regions protected by mutex locks. The thread executing `snapify_pause()` in the host process and in the offload process will acquire these locks. The locks will be released in `snapify_resume()`.

Capture. To capture a snapshot of an offload process the host process calls `snapify_capture()`. Similar to `snapify_pause()`, the caller of `snapify_capture()` passes the handle to the offload process and the path to the directory on the host's file system where the snapshot files should be saved. It also gives a Boolean variable `terminate` to indicate whether the offload process should be terminated after its snapshot is captured. At the beginning `snapify_capture()` sends the capture request first to the COI daemon, which in turn forwards the request to the offload process through the pipe opened in `snapify_pause()`. The snapshot of the offload process can be captured by any application-transparent checkpoint tool.

Our current implementation uses BLCR to capture the snapshot of the offload process. When the offload process receives the capture request from the pipe, it calls BLCR's `cr_request_checkpoint()`. When the snapshot is captured, the offload process sends back the completion message using the pipe to the COI daemon, which in turn informs the host process.

The snapshot of an the offload process is saved on the host file system. The snapshot is written by the checkpoint and restart tool running on the coprocessor. Section 6 details several novel techniques of saving a snapshot "on the fly" from the coprocessor to the host file system.

Notice that `snapify_capture()` is a non-blocking function call. It returns immediately with a semaphore `m_sem` in `snapify_t* snapshot`. The caller can thereafter call `snapify_wait()` with the `snapify_t` structure to wait for the completion of the capturing operation. The semaphore will be signaled when the host process receives the complete message from the COI daemon.

4.2 Resume

To resume the execution of the blocked threads of both the host process and the offload process after a snapshot of the offload process is taken, the host process calls `snapify_resume()` with the handle of the offload process. To resume the host process first sends a resume request to the COI daemon. The daemon then forwards the request to the the offload process through the pipe that is created in `snapify_pause()`. In `snapify_resume()`, both the host process and the offload process release all the locks acquired in the pause operation. Once the locks are released in the offload process, the offload process sends an acknowledgement back to the host process through the COI daemon. After the host process receives the acknowledgement and releases the locks, it returns from `snapify_resume()`.

4.3 Restore

To restore an offload process from its snapshot the host process calls `snapify_restore()` with the path to the snapshot files. Snapify relies on the COI daemon and Xeon Phi's checkpoint and restart tool (BLCR) that is also used by `snapify_capture()` to restart an offload process. To restore, `snapify_restore()` first sends a restore request to the COI daemon. After receiving the restore request, the COI daemon first copies the local store and the runtime libraries needed by the offload process on the fly to the coprocessor. Then it calls BLCR to restart the offload process from its snapshot. We developed a novel I/O mechanism, called *Snapify-IO*, that allows BLCR to read the snapshot of the offload process "on-the-fly" from the host storage directly without first saving the entire snapshot in the memory file system on Xeon Phi. Snapify-IOwill be discussed in detail in Section 6.

After BLCR restores the process image of the offload process, the host process and the offload process will reconnect all of the disconnected SCIF communication channels between them. After SCIF channels are restored, the host process and the offload process re-registers the memory regions used by the COI buffers for RDMA. The re-registration of a buffer may return a new RDMA address different from the original one. Therefore we keep a lookup table of (old, new) address pairs for conversion.

Since the restored offload process is new, `snapify_restore()` returns a new COI process handle (`COIProcess*`) to the offload process. The new handle can be used by the host process in the subsequent COI function calls. Notice that the offload process, though restored, is not fully active after `snapify_restore()` returns. The caller needs to call `snapify_resume()` so that the blocked threads in the host process and the offload process can continue their executions.

5. API USE SCENARIOS

In this section we explain how Snapify's API can be used to implement checkpoint and restart, process swapping and migration.

Checkpoint and restart. To take a checkpoint of an offload application we need to capture both the snapshots of the host process and of the offload process. To capture a snapshot of the host process, we can use an application-transparent checkpoint and restart tool like BLCR on the host. As to the snapshot of the offload process, we use `snapify_pause()` and `snapify_capture()` in Snapify's API.

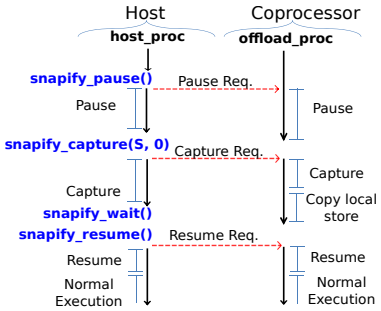
The sample code in Fig. 5(a) shows how Snapify's API can be combined with the host BLCR to implement checkpoint and restart for offload applications. Fig. 5(b) and 5(c) reports the timing diagrams of the checkpoint and restart. The function `snapify_blcr_callback()` is a callback function that is registered to BLCR on the host. When BLCR receives a checkpoint request, it will call `snapify_blcr_callback()`. Within `snapify_blcr_callback()`, we call BLCR's `cr_checkpoint()` to take a snapshot (a checkpoint) of the host process. Before `cr_checkpoint()`, we call `snapify_pause()` and `snapify_capture()` to take a snapshot of the offload process. Notice that `snapify_capture()` is a non-blocking call. Therefore we need to wait for its return in the "continue" section of the `if` statement after `cr_checkpoint()` returns.

In restarting BLCR first restores the host process. The execution of the restored host process will begin after `cr_checkpoint()` returns with `ret` > 0. The control flow of the execution will go through the "restart" section of the "if" statement. There we call `snapify_restore()` to recreate the offload process. In the sample code the offload process will be restored on a Xeon Phi coprocessor whose device ID is extracted from `COIProcess*` by function `GetDeviceID()`.

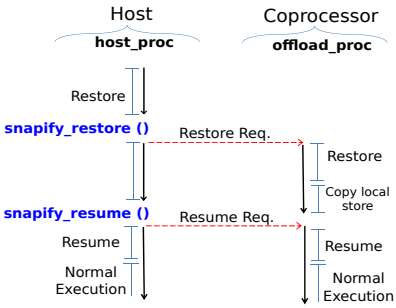
Process swapping. Fig. 6 shows sample process swapping-out and swapping-in functions and their timing diagrams. Process-

```
int snapify_blcr_callback(void* args){
    int ret = 0;
    snapify_t* snapshot = (snapify_t*)args;
    snapify_pause(snapshot);
    snapify_capture(snapshot, false);
    ret = cr_checkpoint(0);
    if ( ret > 0 ) { // Restarting.
       snapify_restore(snapshot,
         GetDeviceId(snapshot->m_process));
       snapify_resume(snapshot);
       // Save snapshot.m_process.
    }
    else { // Continue.
       snapify_wait(snapshot);
       snapify_resume(snapshot);
    }
}
```

(a) Sample code.



(b) Checkpoint timing diagram.



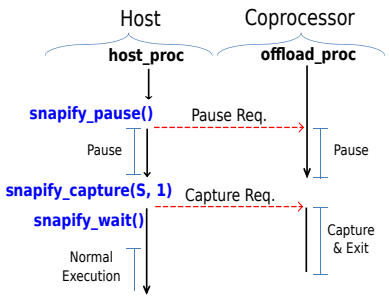
(c) Restart timing diagram.

Figure 5: Using Snapify's API to implement checkpoint and restart.

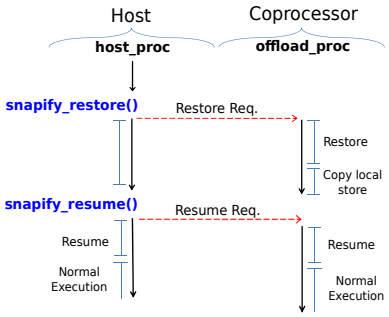
```
snapify_t*  snapify_swapout(const char* path,
      COIProcess* proc){
    snapify_t* snapshot = (snapify_t*)malloc(
        sizeof(snapify_t));
    snapshot->m_snapshot_path = path;
    snapshot->m_process = proc;
    snapify_pause(snapshot);
    snapify_capture(snapshot, true);
    snapify_wait(snapshot);
    return snapshot;
}

COIProcess* snapify_swapin(snapify_t*
     snapshot, int device){
    COIProcess* ret = 0;
    snapify_restore(snapshot, device);
    snapify_resume(snapshot);
    ret = snapshot->m_process;
    free(snapshot);
    return ret;
}
```

(a) Sample code.



(b) Swapping-out timing diagram.



(c) Swapping-in timing diagram

Figure 6: Using Snapify's API to implement process swapping.

swapping can be used, for example, by a job scheduler to swap out one offload process and swap in another based on the scheduler's scheduling and resource management policies. Both of the swapping functions are called in the context of the host process. The caller of `snapify_swapout()` needs to prepare a directory where the snapshot files of the offload process can be stored, and passes the path in parameter `snapshot` to `snapify_swapout()`. The implementation of `snapify_swapout()` is fairly straightforward: we call `snapify_pause()`, `snapify_capture()`, and `snapify_wait()` one by one. Since the offload process is to be swapped out, we set the second parameter of `snapify_capture()` to be true, terminating the offload process after its snapshot is captured and saved. The returned pointer of `snapify_t` structure from `snapify_swapout()` represents a snapshot of the offload process. It can be used to restore the offload process.

The swapping-in of an offload process reverses the effect of swapping-out. In `snapify_swapin()`, we use `snapify_restore()` to restore the offload process. The returned Snapify data structure `snapshot` from `snapify_swapout()` is passed to `snapify_swapin()`, which uses the path of the snapshot files in `snapshot` to restart the the offload process on the specified Xeon Phi coprocessor (identified by `device_to` parameter). The new handle to the restored offload process is returned at the end of `snapify_swapin()`.

Process migration. Fig. 7 shows the implementation of a process-migration function. Process migration moves an offload process from one coprocessor to another on the same machine. It can be viewed as swapping out the offload process from coprocessor 1 and swapping it in on coprocessor 2. Its implementation simply reuses `snapify_swapout()` and `snapify_swapin()`.

Command-line tools. The offload applications can directly benefit from Snapify without any modifications. Checkpoint and restart can be applied transparently by using BLCR's `cr_checkpoint` command-line tool on the host system. This utility will send a signal to trigger the checkpoint procedure, which calls Snapify's own BLCR callback function as shown in Fig. 5(a). If the MPI runtime supports BLCR, MPI applications using Xeon Phi for offload computation will automatically benefit from Snapify.

In order to provide swapping and migration transparently, we provide a command-line utility named `snapify`. Its arguments are the PID of the host process and a command. The commands include swapping-out, swapping-in, and migration. In case of

```
COIProcess* snapify_migration(COIProcess* proc, int
     device_to){
    const char* path = "/tmp";
    snapify_t* snapshot = snapify_swapout(path, proc);
    return snapify_swapin(snapshot, device_to);
}
```

Figure 7: An implementation of process migration.

swapping-in and migration, `snapify` also needs an additional parameter indicates the coprocessor number on which the offload process will be launched. This utility signals the host process and submits the command through a pipe. The signal handler provided by Snapify in the host process then calls one of the three functions in Fig. 6(a) and 7 according to the user-given command.

Remark. Process swapping and migration may lead to resource contentions. E.g., two processes might be swapped into the same Xeon Phi. Such problems are best addressed by a job scheduler like COSMIC in [5], and are beyond the scope of this paper.

6. Snapify-IO

All of the snapshots taken on the host and on a Xeon Phi coprocessor are saved to a file system mounted in the host OS. Snapify provides three novel "on-the-fly" approaches to store and retrieve snapshots between the host and coprocessors. All of these methods use very little Xeon Phi memory for buffering. In the following we will first describe the most efficient approach based on SCIF's RDMA API, called Snapify-IO. Then we will discuss our NFS-based methods.

Snapify-IO. Snapify-IO is a remote file access service that transfers data using RDMA between the host and the Xeon Phi coprocessors on a Xeon Phi server. It provides a simple interface that uses UNIX file descriptors as data access handles. Snapify-IO allows a local process running on a Xeon Phi coprocessor to read from or write to a remote file on the host through standard file I/O functions, as if the file is local. For example, the file descriptor created by Snapify-IO can be directly passed to BLCR for saving and retrieving snapshots. Internally, Snapify-IO transfers the data over the PCIe bus using SCIF's RDMA data transfer functions.

Fig. 8 shows Snapify-IO's architecture. Snapify-IO consists of a user-level library providing a simple I/O interface (Snapify-IO library) and a standalone binary called *Snapify-IO daemon*. The Snapify-IO library is linked to the user code that wants to use Snapify-IO for remote file I/O, while each SCIF node (the host and any of the Xeon Phi coprocessors on a Xeon Phi server) runs a Snapify-IO daemon as a long-running process. The Snapify-IO daemon serves I/O requests from both the local user processes using Snapify-IO library and remote Snapify-IO daemons. It can either receive data from a local process, transfer the data to a remote Snapify-IO daemon, which in turn saves the data into a remote file system. Or it can retrieve data from a local file system, transfer the data to a remote Snapify-IO daemon, which feeds the data into a remote user process.

Snapify-IO library is designed for transparent integration with the standard `read()` and `write()` system calls. Its only API function `snapifyio_open()` returns a standard UNIX file descriptor. It accepts three arguments: a SCIF node ID, a path to a file that is valid on the SCIF node, and a file access mode flag indicating either a read or write mode (but not both). The returned file descriptor represents a file on a (remote) SCIF node as specified by the arguments.

Snapify-IO uses a UNIX socket as the local communication channel between the Snapify-IO library and the Snapify-IO daemon. When `snapifyio_open()` is called, the Snapify-IO library creates a UNIX socket that connects to the local Snapify-IO daemon. Once the socket is established, `snapifyio_open()` sends the SCIF node ID, the file path, and the access mode to the Snapify-IO daemon. It then returns the file descriptor of the socket to the caller. To serve the local socket connections, the Snapify-IO daemon has a local server thread listening on a designated port. Once the server thread accepts the socket connection from the Snapify-IO library, it spawns a local handler thread to handle further I/O activities coming from the user process, which may either write to or read from the socket, depending on the access mode. Notice that the file descriptor returned by `snapifyio_open()` is a UNIX file descriptor, so the user code can call `close()` to release the resources associated with the file descriptor.

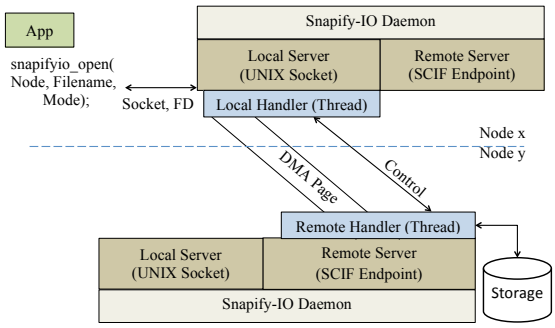


Figure 8: The architecture of Snapify-IO.

The communication channel between two Snapify-IO daemons is a SCIF connection. After receiving the SCIF node ID, the file path, and the access mode from a local user process, the Snapify-IO daemon's local handler thread will create a new SCIF connection to the Snapify-IO daemon on the specified SCIF node. Once the SCIF connection is established, the local handler thread will forward the path and the access mode of the file to the remote Snapify-IO daemon, and register an internal buffer to the SCIF library for RDMA transfer. The buffer size is configurable. To balance between the requirement of minimizing memory footprint and the need of shorter transfer latency, the buffer size is set at 4MB. To handle incoming SCIF connections, the Snapify-IO daemon employs a remote server thread, which listens to a predetermined SCIF port. Once the remote server thread accepts a SCIF connection from a remote Snapify-IO daemon, it spawns a handler thread to handle communications over the newly established SCIF channel.

Once the communication channels are established, the local handler thread will start to direct the data flow between the user application and the remote file. In write access mode, the local handler will copy the data written to the socket by the user application to the registered RDMA buffer. After the buffer is filled, the local handler will send a SCIF message to notify the remote Snapify-IO daemon (i.e. the daemon's remote handler thread) using `scif_send()`. Subsequently the remote handler thread will use SCIF's RDMA function `scif_vreadfrom()` to read the data from the registered RDMA buffer, and saves the data to the file system at the specified location. After the RDMA completes, the local handler thread will reuse the RDMA buffer and repeat the above process until all data from the user process are saved to the remote file. In the read access mode, the data flow in the reverse direction. The remote handler thread in the remote Snapify-IO daemon will read data from the specified file, and copy the data to the registered RDMA buffer using `scif_vwriteto()`. Once the buffer

Table 2: Characteristics of our Xeon Phi server.

	Host Processor	Coprocessor
CPU	Intel E5-2630 @ 2.30GHz	Intel Xeon Phi 5110P
Cores	6 physical cores (12 threads) per socket	60 physical cores (240 threads) per coprocessor
Memory	32GB	16GB per coprocessor
OS	Linux RHEL 6.2, kernel 2.6.32-220	Linux kernel 2.6.38.8 MPSS 2.1.6720-130
Number	2 CPU sockets	2 coprocessors

is filled, it will notify the local handler thread in the local Snapify-IO daemon, which in turn will copy the data from the RDMA buffer to the socket.

Snapify uses Snapify-IO to save and retrieve snapshots. The COI library with Snapify implementations is linked with the Snapify-IO library. To take a snapshot of an offload process, Snapify calls `snapifyio_open()` in write mode in the pre-snapshot phase. The returned file descriptor is then passed to BLCR, which uses it to write the context file. Similarly, to restart an offload process Snapify calls `snapifyio_open()` to open a remote context file in read mode. The returned file descriptor is then used by BLCR to load the process context from the remote file to the local Xeon Phi's memory. Thanks to Snapify-IO, the data transfer between a Xeon Phi coprocessor and the host's file system is completely transparent to BLCR. In addition, the Snapify-IO library does not introduce extra SCIF connections. Therefore it does not complicate the process of taking a snapshot.

NFS. We also implemented two optimizations to speedup NFS-based snapshot storage. In these approaches Snapify uses NFS to mount the host file system on a Xeon Phi coprocessor. A snapshot is stored or retrieved through the NFS. To overcome the problem of high latency of small writes in NFS, we developed two new approaches based on buffering. In the first approach we modified BLCR's kernel module such that it accumulates write data to a larger chunk before the data is written to the file system. Since using a modified BLCR's kernel module may not always be feasible, our second approach uses the same concept but applies it in the user level. In this approach the BLCR writes are redirected to our user-space utility through the standard output and input. This utility buffers data from its standard input and writes out the data in the buffer to NFS at larger granularity.

7. EXPERIMENTAL RESULTS

We used micro benchmarks and a suite of MPI and OpenMP applications to evaluate Snapify-IO and Snapify on a Xeon-Phi cluster. This section reports our experimental results.

Setup. Table 2 shows the hardware and software configuration of our computing system. For Snapify-IO evaluation, we used a single node that has one Xeon Phi (8GB of physical memory). For MPI applications, we used a 4-node cluster, where each node in the cluster has one Xeon Phi many-core processor with 8GB of memory.

Snapify-IO performance. To evaluate the performance of Snapify-IO, we used a micro-benchmark that copies files of various sizes between the host and the Xeon Phi. The micro-benchmark runs natively on the Xeon Phi. We compared the time taken by our Snapify-IO to move files between the host and the Xeon Phi with the time taken by two methods natively supported by Xeon Phi's OS, i.e. `scp` and read/write from NFS mounted directories.

Table 3 shows the time taken to copy files of different sizes (file-size ranged from 1MB to 1GB). We observed that Snapify-IO consistently performs better than NFS and `scp` (except for the 1 MB file-size case, where NFS outperforms others by buffering data). As the file size increases, Snapify-IO's advantage is more pronounced.

Table 3: File copy performance (seconds).

Size	Device to Host			Host to Device		
(MB)	scp	NFS	Snapify-IO	scp	NFS	Snapify-IO
1	0.97	0.01	0.03	0.48	0.01	0.07
64	15.07	3.23	0.43	14.08	1.94	0.53
128	29.88	6.36	0.85	28.30	3.96	1.82
256	57.02	11.81	1.67	55.50	7.95	3.68
512	113.71	20.91	3.34	111.07	16.07	6.17
1024	224.68	40.10	6.76	221.59	31.06	9.57

For a 1GB file, Snapify-IO has about 6x better write performance and 3x better read performance when compared with NFS. For the same file, Snapify-IO has 30x faster write performance and 22x faster read performance when compared with `scp`. We also observed that transfer of a file from the Xeon Phi to the host by using Snapify-IO is generally faster than moving the same file from the host to Xeon Phi. This is because Snapify-IO daemon on the host flushes the file to the secondary storage asynchronously. Thus the write operation on the host runs parallel to the data transfer.

We also evaluated the impact of using Snapify-IO for storing and restoring BLCR's checkpoints of processes on the Xeon Phi. We ran a second micro-benchmark as a native application on the coprocessor and captured the snapshots using BLCR. Our micro-benchmark performed a `malloc()` call and it had a long loop in an OpenMP region (240 threads). We used different malloc sizes (ranging from 1MB to 4GB) to control the file-size of snapshots. Table 4 compares the performance of Snapify-IO with three variants of read/write from NFS mounted directories, as well as a method (labeled as *Local* in Table 4) that saves the application snapshot in the physical memory of the Xeon Phi. The BLCR checkpoint time in Table 4 is the end-to-end latency of capturing and saving the process snapshot. BLCR restart time is the end-to-end time to read and restore the snapshot. Note that in most cases the snapshot is written to and read from the host file system (except the *Local* case). As expected, storing and restoring snapshots from the physical memory of the Xeon Phi (the *Local* case) takes the least time. However, when the checkpoint file-size increases to 4GB, it is impossible to store the checkpoint file in the physical memory of the Xeon Phi (memory limit on the Xeon Phi card is 8GB, and 4GB is already used by the micro-benchmark). In practice, it is not feasible to save checkpoint files locally because, more often than not, several other processes on the Xeon Phi are already using the limited physical memory on the Xeon Phi.

The performance of all three variants of NFS was poor (when compared with Snapify-IO) for storing checkpoints. BLCR performs multiple small writes before reaching the loop where it actually takes snapshots of the application's memory pages, and these small writes lead to poor performance for the NFS variants. Our method of "NFS-Buffered in kernel" boosts the performance of NFS to a large degree while our buffering in user-space does so to a lesser degree (but it still provides significant improvements). Finally, Snapify-IO performance is a large improvement compared to NFS and NFS-Buffered (both modes).

Note that the buffering solutions do not apply to the cases of restarting or restoring. Again, restarting from checkpoint files in the physical memory of the Xeon Phi is very fast, but this is generally not possible due to the limited physical memory on the Xeon Phi. We observed that Snapify-IO performs 1.4x, 2.6x and 5.9x faster than NFS for 1MB, 256MB and 4GB snapshots, respectively.

Snapify overhead. We evaluated Snapify on 8 OpenMP and 3 NAS MPI benchmarks (LU-MZ, SP-MZ, BT-MZ in [9]). The benchmarks are described in Table 5. All benchmarks were modified to offload computations to the Xeon Phi. All time measurements were made on the host, unless mentioned otherwise.

Table 4: Comparing Snapify-IO with NFS-based I/O in BLCR (seconds).

	Checkpoint time (Snapshot + Write)					Restart Time (Read + Restore)		
Size (MB)	Local (RAM)	NFS	NFS-Buffered (kernel)	NFS-Buffered (user)	Snapify-IO	Local (RAM)	NFS	Snapify-IO
1	2.16	67.64	1.70	2.71	**2.76**	0.577	1.392	**0.979**
64	2.38	71.21	3.25	4.81	**2.28**	0.829	3.547	**1.377**
128	2.62	70.97	4.30	7.64	**3.71**	0.832	5.834	**2.066**
256	3.15	75.67	7.30	12.48	**4.59**	1.255	8.975	**3.397**
512	4.12	80.98	12.60	20.98	**6.44**	2.034	17.613	**4.934**
1024	6.34	87.27	21.44	30.47	**9.94**	3.732	32.550	**7.373**
2048	10.14	118.65	38.48	53.82	**17.41**	6.736	62.217	**13.506**
4096	NA	155.92	93.05	109.06	**32.84**	NA	121.452	**22.935**

Table 5: Description of benchmarks.

Name	Description	Problem Size
MD	Molecular dynamics simulation	25000 particles, 10 time steps
MC	Monte Carlo simulation of N paths and T time steps	N = 32M, T = 2000
SG	A series of matrix-matrix multiplications (SGEMM)	8Kx8K matrices, 10 iterations
SS	Supervised semantic search indexing computing top K for each of the Q queries	256K documents, K=32, Q=512
KM	Computing K-means using Lloyd clustering algorithm	4M points, 3 dimensions, 32 means
LU-MZ	A CFD application using lower-upper Gauss-Seidel solver [2]	Grid: 162x162x162, 250 iterations
BT-MZ	Computation fluid dynamics (CFD) using block tri-diagonal solver [2]	Grid: 162x162x162, 200 iterations
SP-MZ	A CFD application using scalar penta-diagonal solver [2]	Grid: 162x162x162, 400 iterations

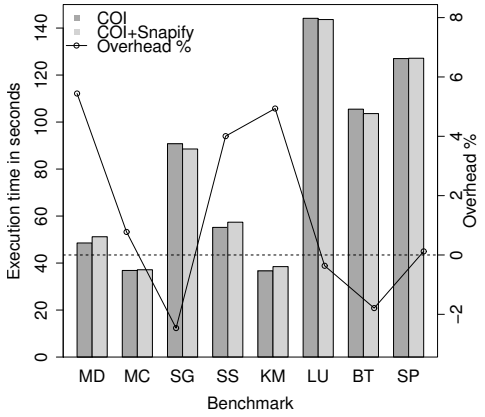


Figure 9: Runtime overhead of Snapify.

Fig. 9 compares the runtime of the normal executions (no snapshot) of the OpenMP benchmarks with and without Snapify support. Each experimental run was repeated 20 times. The average runtime is reported as bars, and the runtime overhead (in percentage) added by Snapify is shown in line graph on the right y-axis. In average Snapify adds a 1.5% overhead to the application runtime, and in the worst case the overhead is less than 5% (MD). We used the Linux command-line tool `time` to measure the end-to-end execution time of an offload application.

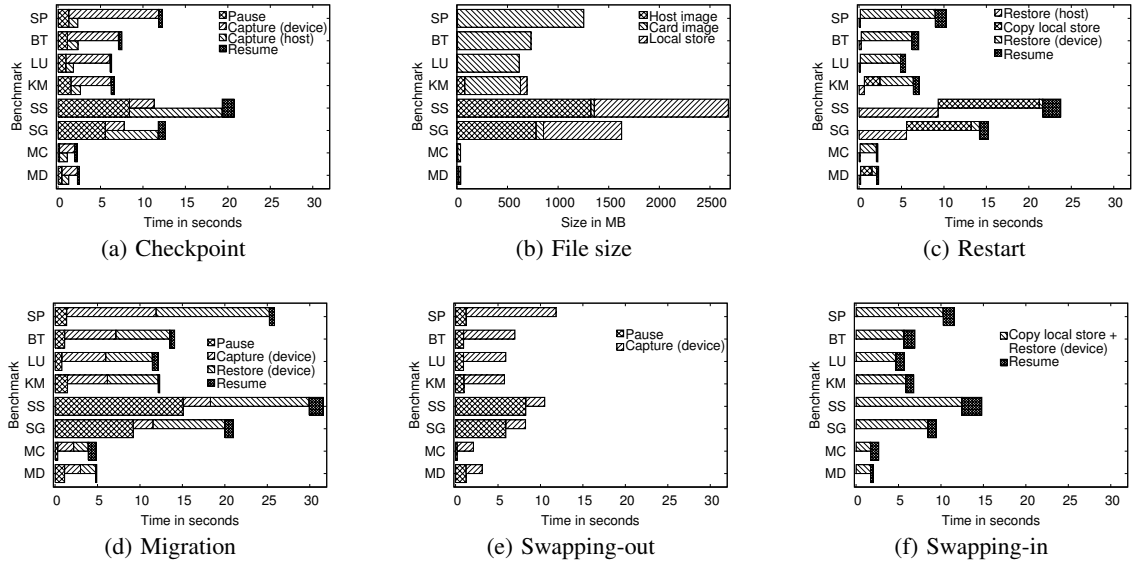
Checkpoint and restart. Fig. 10(a) shows the checkpoint time and 10(b) reports the size of the files generated by the checkpoint procedure. As detailed in Section 4.1, during pause the local store (files on Xeon Phi for COI buffers) of the offload process was stored in the snapshot directory on the host system as a file. Thus for benchmarks with a large local store (SS and SG in Fig. 10(b)), the pause is longer. The time that BLCR on the host and BLCR on the coprocessor take to capture and save snapshots are the bars labeled as "Snapshot + Write (host)" and "Snapshot + Write (device)", respectively. The host BLCR finishes early in all cases except for SS and SG. In these two cases, Fig. 10(b) indicates a large host-process snapshot while the offload-process snapshot is fairly small. Thus the offload process finishes early. The checkpoint time ranges from 3 to 21 seconds in time, shorter for small files (8.4 MB) and longer for large ones (1.3 GB).

Fig. 10(c) reports the restart time of the OpenMP benchmarks. The total restart time ranges from 3 to 24 seconds across the benchmarks. Fig. 10(c) also reports the breakdown of the time spent in each stage of the restart. The host-restart time varies based on the size of the host-process snapshot. Benchmarks SS and SG have larger host snapshots, and thus longer host-restart time. The time of restoring an offload process strongly depends on the size of local store, which is copied from the host to the coprocessor when the offload process is restored.

Process migration. Fig. 10(d) shows the runtime overhead of process migration. The migration time varies from 4.9 seconds (MC) to 31.6 seconds (SS). As expected, it is strongly correlated with the size of the local store and the snapshot of an offload process. In process migration, the offload process copies its local store directly from its current coprocessor to another coprocessor using Snapify-IO. Thus the pause time in process migration is different from the one in the checkpoint procedure. In all but one benchmarks the time of capturing and saving the snapshot of an offload process is shorter than the time of reading the snapshot and restoring the offload process. This is because Snapify-IO is faster when writing to the host from a coprocessor, as explained earlier.

Process swapping. Fig. 10(e) and 10(f) show the runtime of swapping-out and swapping-in, respectively. The time of swapping out the offload processes ranges from 2.1 seconds to 11.8 seconds, and the time of swapping-in takes between 2 seconds and 14.8 seconds. Except in the case of SS and SG, the pause of swapping-out is much shorter than the time of the capturing phase. Again, this is because the local stores of SS and SG are larger than their snapshots.

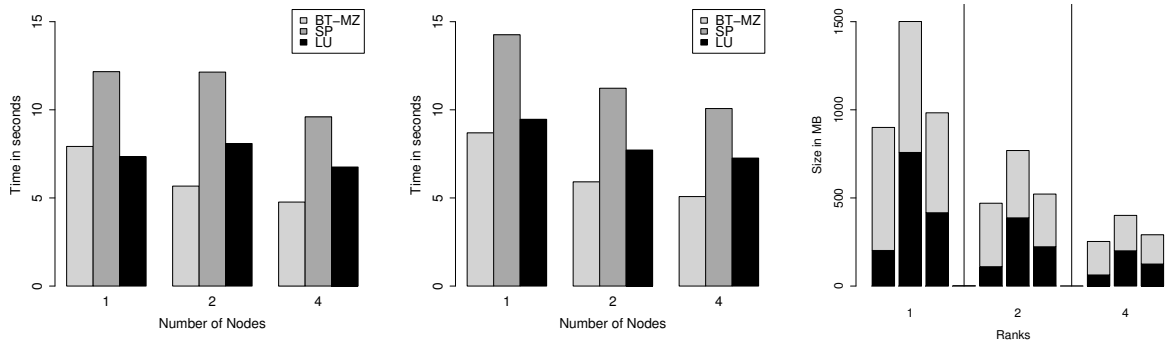
Checkpoint and restart for MPI. We use three MPI benchmarks LU-MZ, SP-MZ, and BT-MZ, to evaluate checkpoint and restart of MPI applications. For all three benchmarks, we choose the class C input size, and run the benchmarks with 1, 2 and 4 MPI tasks (ranks). Each rank is executed on one node. Fig. 11(a) and 11(b) report the time of taking a checkpoint and restarting from a checkpoint, respectively. Fig. 11(c) shows the checkpoint size of a single rank. We observe that as the number of nodes increases, CR time decreases at various degrees. This is because the checkpoint size of each MPI rank decreases as the total number of MPI ranks increases. CR time ranges between 4 and 14 seconds for a single checkpoint, depending on the respective benchmarks and the number of the MPI ranks. When no checkpoint is performed, the runtime of the benchmarks ranges from 2-3 minutes for the selected



(a) Checkpoint (b) File size (c) Restart

(d) Migration (e) Swapping-out (f) Swapping-in

Figure 10: Performance evaluation on OpenMP benchmarks.



(a) Checkpoint (b) Restart (c) File size

Figure 11: Performance evaluation of checkpoint and restart on MPI benchmarks.

input size. This indicates the feasibility to taking frequent checkpoints, particularly for larger inputs and longer runtime.

8. RELATED WORK

Checkpoint and restart (CR) has a long history in computing systems. Libckpt [26] was one of the first UNIX implementations. Condor [36] provides CR and process migration for load balancing. These libraries provide process-level checkpointing. There are also user-level checkpointing libraries [10, 13, 19], and recent studies suggest using nonvolatile memory to improve CR performance [21]. Rollback-recovery protocols in message-passing systems is a classical research field [12]. Multiple MPI libraries, including Open MPI [15] and MPICH-V [4], provide distributed CR for MPI applications.

Several previous studies proposed CR and process migration for GPUs. CheCUDA supports a part of basic CUDA APIs [35]. It copies all the user data from a GPU to the host system during checkpointing. Then it destroys the CUDA context before taking a checkpoint. The data and context are copied back from the host to the GPU at post-checkpoint (restart) time. NVCR keeps a database of the memory allocations in GPUs [23]. Before checkpointing it releases all the memory contents and replays the log at the restart time. The replay is necessary to avoid invalid memory addresses at restart time. This imposes overhead during restart time and specially normal execution of application. CheCL provides CR and migration for OpenCL [34]. CheCL synchronizes the host and command queues by waiting for all commands to complete. A command queue is used to schedule the execution of kernels and perform memory operations in OpenCL context. CheCL benefits from a proxy mechanism to decouple the process from OpenCL implementation. However, each kernel execution involves an additional step of inter-process communication, incurring extra communication latency.

Process migration has been extensively studied in the past. Zap is a system that performs process-group migration [24], while Wang et al. studied live migration of processes in HPC environment [37]. In addition, live migration of entire virtual machines is a very useful tool for data center and cluster administrations [8].

9. CONCLUSION

To conclude, in this paper we presented Snapify, a set of extensions to Xeon Phi's software stack that captures process snapshots of offload applications. Using Snapify we implemented application-transparent checkpoint and restart, process migration, and process swapping for offload applications. Experimental results on OpenMP and MPI offload applications show that Snapify

added negligible runtime overhead (1.5% in average) and is very efficient in taking snapshots and restoring processes.

We also created Snapify-IO, a remote file access service based on RDMA to transfer process snapshots between the host system and coprocessors. Snapify-IO benefits both Snapify *and* the default checkpoint and restart tool for native applications. For native applications our experimental results show that Snapify-IO achieves 4.7x to 8.8x speedup in checkpoint, and 4.4x to 5.3x speedup in restart over NFS, for snapshot size between 1GB to 4GB.

10. REFERENCES

[1] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proc. of Intl. Parallel and Distributed Processing Symposium*, 2009.

[2] D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Advanced Supercomputing Division, Mar. 1994.

[3] M. Becchi, K. Sajjapongse, I. Graves, A. Procter, V. Ravi, and S. Chakradhar. A virtual memory based runtime to support multi-tenancy in clusters with GPUs. In *Proc. of the Intl. Symposium on High-Perf. Parallel and Distributed Computing*, pages 97–108, June 2012.

[4] G. Bosilca et al. MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In *Proc. of the ACM/IEEE Intl. Conf. for High Perf. Computing, Networking, Storage and Analysis*, 2002.

[5] S. Cadambi, G. Coviello, C.-H. Li, R. Phull, K. Rao, M. Sankaradass, and S. Chakradhar. COSMIC: Middleware for high performance and reliable multiprocessing on Xeon Phi coprocessors. In *Proc. of the Intl. Symposium on High-Perf. Parallel and Distributed Computing*, pages 215–226, June 2013.

[6] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388, Nov. 2009.

[7] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Tran. on Computer Syst.*, 3(1):63–75, Feb. 1985.

[8] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. of the USENIX Conf. on Networked Syst. Design and Implementation*, pages 273–286, 2005.

[9] R. F. V. der Wijngaart and H. Jin. NAS parallel benchmarks, multi-zone versions. Technical Report NAS-03-010, NASA Advanced Supercomputing Division, July 2003.

[10] W. R. Dieter. User-level checkpointing for linuxthreads programs. In *USENIX Annual Technical Conf. (FREENIX Track)*, pages 81–92, 2001.

[11] J. Duell. The design and implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical report, Lawrence Berkeley National Laboratory, 2003.

[12] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.

[13] http://www.criu.org/.

[14] http://www.top500.org/.

[15] 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 *Proc. of Intl. Parallel and Distributed Processing Symposium*, Mar. 2007.

[16] Intel Corporation. *MIC COI API Reference Manual*, 0.65 edition.

[17] Intel Corporation. *Intel Manycore Platform Software Stack (Intel MPSS): User's Guide*, 2013.

[18] Intel Corporation. *Symmetric Communications Interface (SCIF) for Intel Xeon Phi Product Family Users Guide*, 2013.

[19] G. J. Janakiraman, J. Renato, S. D. Subhraveti, and Y. Turner. Cruz: Application-transparent distributed checkpoint-restart on standard operating systems. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 260–269, June 2005.

[20] J. Jeffers and J. Reindeer. *Intel Xeon Phi Coprocessor High-Performance Programming*. Morgan Kaufmann Publishers, 2013.

[21] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. Optimizing checkpoints using NVM as virtual memory. *Proc. of Intl. Parallel and Distributed Processing Symposium*, pages 29–40, 2013.

[22] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Tran. on Software Engineering*, SE-13(1):23–31, Jan. 1987.

[23] A. Nukada, H. Takizawa, and S. Matsuoka. NVCR: A transparent checkpoint-restart library for NVIDIA CUDA. In *IEEE Intl. Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 104–113, 2011.

[24] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proc. of the USENIX Conf. on Oper. Syst. Design and Implementation*, pages 361–376, Dec. 2002.

[25] R. Phull, C.-H. Li, K. Rao, H. Cadambi, and S. Chakradhar. Interference-driven resource management for GPU-based heterogeneous clusters. In *Proc. of the Intl. Symposium on High-Perf. Parallel and Distributed Computing*, pages 109–120, June 2012.

[26] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *USENIX Technical Conf. Proc.*, 1995.

[27] M. Rieker, J. Ansel, and G. Cooperman. Transparent user-level checkpointing for the native posix thread library for linux. In *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 492–498, July 2006.

[28] J. F. Ruscio, M. A. Heffner, and S. Varadarajan. DejaVu: Transparent user-level checkpointing, migration, and recovery for distributed systems. In *Proc. of Intl. Parallel and Distributed Processing Symposium*, Mar. 2007.

[29] K. Sajjapongse, X. Wang, and M. Becchi. A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with GPUs. In *Proc. of the Intl. Symposium on High-Perf. Parallel and Distributed Computing*, pages 179–190, June 2013.

[30] F. Salfner, M. Lenk, and M. Malek. A survey of online failure prediction methods. *ACM Comput. Surv.*, 42(3), Mar. 2010.

[31] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proc. of the Symposium of Los Alamos Computer Science Institute*, pages 479–493, 2003.

[32] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka. Design and modeling of a non-blocking checkpointing system. In *Proc. of the ACM/IEEE Intl. Conf. for High Perf. Computing, Networking, Storage and Analysis*, Nov. 2012.

[33] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *J. of Physics: Conf. Series*, 78(1), 2007.

[34] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi. CheCL: Transparent checkpointing and process migration of OpenCL applications. In *Proc. of Intl. Parallel and Distributed Processing Symposium*, pages 864–876, May 2011.

[35] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi. CheCUDA: A checkpoint/restart tool for CUDA applications. In *Proc. of the Intl. Conf. on Parallel and Distributed Computing, Applications and Technologies*, pages 408–413, Dec. 2009.

[36] T. Tannenbaum and M. Litzkow. Checkpointing and migration of unix processes in the Condor distributed processing system. *Dr Dobbs Journal*, Feb. 1995.

[37] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in HPC environments. In *Proc. of the ACM/IEEE Intl. Conf. for High Perf. Computing, Networking, Storage and Analysis*, 2008.