A 1 PB/s File System to Checkpoint Three Million MPI Tasks[*]

Raghunath Rajachandrasekar
The Ohio State University
rajachan@cse.ohio-state.edu

Adam Moody
Lawrence Livermore National Laboratory
moody20@llnl.gov

Kathryn Mohror
Lawrence Livermore National Laboratory
kathryn@llnl.gov

Dhabaleswar K. (DK) Panda
The Ohio State University
panda@cse.ohio-state.edu

ABSTRACT

With the massive scale of high-performance computing systems, long-running scientific parallel applications periodically save the state of their execution to files called checkpoints to recover from system failures. Checkpoints are stored on external parallel file systems, but limited bandwidth makes this a time-consuming operation. Multilevel checkpointing systems, like the Scalable Checkpoint/Restart (SCR) library, alleviate this bottleneck by caching checkpoints in storage located close to the compute nodes. However, most large scale systems do not provide file storage on compute nodes, preventing the use of SCR.

We have implemented a novel user-space file system that stores data in main memory and transparently spills over to other storage, like local flash memory or the parallel file system, as needed. This technique extends the reach of libraries like SCR to systems where they otherwise could not be used. Furthermore, we expose file contents for Remote Direct Memory Access, allowing external tools to copy checkpoints to the parallel file system in the background with reduced CPU interruption. Our file system scales linearly with node count and delivers a 1 PB/s throughput at three million MPI processes, which is 20x faster than the system RAM disk and 1000x faster than the parallel file system.

Categories and Subject Descriptors

C.4 [**PERFORMANCE OF SYSTEMS**]: Fault tolerance; D.4.3 [**File Systems Management**]: Distributed file systems; D.4.5 [**Reliability**]: Checkpoint/restart

Keywords

HPC, Multilevel Checkpointing, File Systems, Persistent-Memory, SSD, RDMA, Fault-Tolerance

[*]This article has been authored by Lawrence Livermore National Security, LLC under Contract #DE-AC52-07NA27344 with the U.S. Department of Energy.

1. INTRODUCTION

In high-performance computing (HPC), tightly-coupled, parallel applications run in lock-step over thousands to millions of processor cores. These applications simulate physical phenomena such as hurricanes or the effect of aging on the nuclear weapons stockpile. The results of these simulations are important and time-critical, e.g., we want to know the path of the hurricane before it makes landfall. Thus, these applications are run on the fastest supercomputers in the world at the largest scales possible. However, due to the increased component count, large-scale executions are more prone to experience faults, with Mean Times Between Failures (MTBF) on the order of hours or days due to hardware breakdowns and soft errors [12, 17, 24, 25, 28].

HPC applications survive failures by saving their state in files called checkpoints on stable storage, usually a globally-accessible parallel file system. When a fault occurs, the application rolls back to a previously saved checkpoint and restarts its execution. Although parallel file systems are optimized for concurrent access by large scale applications, checkpointing overhead can still dominate application run times, where a single checkpoint can take on the order of tens of minutes [14, 22]. A 2005 study by Los Alamos National Laboratory shows that about 60% of an HPC application's wall-clock time was spent in checkpoint/restart alone [21]. Similarly, a study from Sandia National Laboratories predicts that a 168-hour job on 100,000 nodes with a node MTBF of 5 years will spend only 35% of its time in compute work and the rest of its time in checkpointing activities [11]. On current HPC systems, checkpointing utilizes 75-80% of the I/O traffic [1, 20]. On future systems, checkpointing activities are predicted to dominate compute time and overwhelm file system resources [10, 18].

Multilevel checkpointing systems are a recent optimization to this problem and reduce I/O times significantly [7, 18]. They utilize node-local storage for low-overhead, frequent checkpointing, and only write a select few checkpoints to the parallel file system. Node-local storage is appealing because it scales with the size of the application; as more compute nodes are used, more storage is available. Unfortunately, node-local storage is a scarce resource. While a handful of HPC systems have storage devices such as SSDs on all compute nodes, most systems only have main memory, and some of those do not provide any file system interface to this memory, e.g., RAM disk. Additionally, to use an in-memory file system, an application must dedicate sufficient memory to store checkpoints, which may not always be feasible or desirable.

We address these problems with a new in-memory file system called `CRUISE`: Checkpoint Restart in User SpacE. `CRUISE` is optimized for use with multilevel checkpointing libraries to provide low-overhead, scalable file storage on systems that provide some form of memory that persists beyond the life of a process, such as System V IPC shared memory. `CRUISE` supports a minimal set of POSIX semantics such that its use is transparent when checkpointing HPC applications. An application specifies a bound on memory usage, and if its checkpoint files are too large to fit within this limit, `CRUISE` stores what it can in memory and then *spills-over* the remaining bytes in slower but larger storage, such as an SSD or the parallel file system. Finally, `CRUISE` supports Remote Direct Memory Access (RDMA) semantics that allow a remote server process to directly read files from a compute node's memory.

In this paper, we make the following contributions:

- Thorough discussion and evaluation of design alternatives for our in-memory file system
- Detailed description of the design and architecture of `CRUISE`
- A new mechanism for honoring memory usage bounds, namely, spill-over
- Interfaces to allow external asynchronous data-transfer libraries to interact with `CRUISE`
- Large-scale performance and scalability evaluation of `CRUISE`

2. DESIGN GOALS AND BACKGROUND

In this section, we list our design goals and present background on the checkpointing library and the I/O workload characteristics for which we designed `CRUISE`.

2.1 Design Goals for CRUISE

We were guided by several goals when architecting `CRUISE`. First, we wanted to provide a file system on machines that have no local storage other than memory. Second, we wanted a framework to support spill-over for checkpoints that are too large to fit in the available local storage. Third, we wanted to enable remote access to checkpoint data using RDMA. Remote access to data stored in compute node memory enables it to be copied to slower, more resilient storage in the background. Fourth, we wanted to develop a file system that could perform near memory speeds to allow for low checkpointing overhead. Finally, we wanted to implement these capabilities with methods that are portable across a range of HPC platforms. In particular our initial target systems are Linux clusters and IBM Blue Gene/Q systems.

2.2 The SCR Library

We developed `CRUISE` to extend the capabilities of the Scalable Checkpoint/Restart (SCR) library [26].[1] SCR is a multilevel checkpointing system that enables MPI applications to attain high bandwidth for checkpoint and restart I/O [18]. SCR achieves this by saving checkpoints to node-local storage instead of the parallel file system. It can use any available file storage, e.g., RAM disks, magnetic hard-drives, or SSDs. SCR caches recent checkpoints, and discards an older checkpoint with each newly saved one. SCR applies redundancy schemes to the cache, so it can recover checkpoint files even if a failure disables a small portion of the system. It periodically flushes a cached checkpoint to the parallel file system in order to withstand catastrophic failures.

[1]Although we developed `CRUISE` to support SCR, it is generally applicable to any multilevel checkpointing library.

SCR's design is based on two key properties. First, a job only needs its most recent checkpoint—as soon as the job writes the next checkpoint, a previous checkpoint can be deleted. Second, the majority of failures only disable a small portion of the system, leaving most of the system intact. For example, the results obtained in [18] showed that 85% of failures disabled less than 1% of the compute nodes on the clusters in question.

2.3 Checkpoint/Restart I/O Characteristics

Checkpoint/restart I/O workloads have certain characteristics that allow us to optimize our design and implementation of `CRUISE`. In this work, we only consider *application-level checkpointing*, where the application explicitly writes its data to files. This differs from *system-level checkpointing* in which the entirety of the application's memory is saved by an external agent. Application-level checkpointing is typically more efficient, because only the data that is needed for restart is saved, instead of the entire memory. Here, we detail the characteristics of typical application-level checkpoint I/O workloads.

A single file per process. Many applications save state in a unique file per process. This checkpointing style is a natural fit for multilevel checkpointing libraries. In fact, SCR imposes the additional constraint that a process may not read files written by another process. As such, there is no need to share files between processes, so storage can be private to each process, which eliminates inter-process consistency and reduces the need for locking.

Dense files. In general, POSIX allows for sparse files in which small amounts of data are scattered at distant offsets within the file. For example, a process could create a file, write a byte, and then seek to an offset later in the file to write more data, leaving a hole. File systems may then optimize for this case by tracking the locations and sizes of holes to avoid consuming space on the storage device. However, checkpoints typically consist of a large volume of data that is written sequentially to a file. Thus, it will suffice to support non-sequential writes in `CRUISE` without incurring the overhead of tracking these holes to optimize data placement.

Write-once-read-rarely files. A checkpoint file is not modified once written, and it is only read during a restart after a failure, which is assumed to be a rare event relative to the number of checkpoints taken. This property makes it feasible to access file data by external methods such as RDMA without concern for file consistency. Once written, the file contents do not change.

Temporal nature of checkpoint data. Since an application restarts from its most recent checkpoint, older checkpoints can be discarded as newer checkpoints are written. SCR records its own metadata to track checkpoint times, so we need not track POSIX file timestamps in `CRUISE`. Also, SCR only stores a few checkpoints at a time, so `CRUISE`uses small fixed-sized arrays to record file metadata.

Globally coordinated operation. Typically, parallel application processes coordinate with each other to ensure that all message passing activity has completed before saving a checkpoint. This coordination means that all processes block until the checkpointing operation is complete, and when a failure occurs, all processes are restarted at the same time. This means that CRUISE can clear all locks when the file system is remounted.

Although, we designed CRUISE to take advantage of the above characteristics, it can be extended to handle other variants. In particular, with slight modification, it is also applicable to uncoordinated checkpointing.

3. DESIGN ALTERNATIVES

Logically, CRUISE requires two layers of software: the first layer intercepts POSIX calls made by the application or checkpoint library, and the second layer interacts with the storage media to manage file data. We considered several design alternatives for each layer that differ in imposed overheads, performance, portability, and capability to support our design goals.

3.1 Intercepting Application I/O

With CRUISE, our objective is to transparently intercept existing application I/O routines such as `read()`, `write()`, `fread()`, and `fwrite()`, and metadata operations such as `open()`, `close()`, and `lseek()`. We considered two options for implementing the interception layer: FUSE and I/O wrappers.

3.1.1 FUSE-based File System

A natural choice for intercepting application I/O in user-space is to use the Filesystem in User Space (FUSE) module [3]. A file system implementation that uses FUSE can act as an intermediary between the application and the actual underlying file system, e.g., a parallel file system.

The FUSE module is available with all mainstream Linux kernels starting from version 2.4.x. The kernel module works with a user-space library to provide an intuitive interface for implementing a file system with minimal effort and coding. Given that a FUSE file system can be mounted just as any other, it is straight-forward to intercept application I/O operations transparently. However, a significant drawback is that FUSE is not available on all HPC systems. Some HPC systems do not run Linux, and some do not load the necessary kernel module.

Another problem is relatively poor performance for checkpointing workloads. First, because I/O data traverses between user-space and kernel-space multiple times, FUSE can introduce a significant amount of overhead on top of any overhead added by the file system implementation. Second, the use of FUSE implies a large number of small I/O requests for writing checkpoints. By default, FUSE limits writes to 4 KB units. Although the unit size can be optionally increased to 128 KB, that is relatively small for checkpoint workloads that can have file sizes on the order of hundreds of megabytes per process. When FUSE is used in such workloads, many I/O requests are generated at the Virtual File System (VFS) layer leading to several context switches between the application and the kernel.

We quantified the overhead incurred by FUSE using a dummy file system that simply intercepts I/O operations from an application and passes the data to the underlying file system, a kernel-provided RAM disk in this experiment. Direct I/O was used to isolate the effects of the VFS cache. For these runs, we measured the `write()` throughput of a single process that wrote a 50 MB file to both native RAM disk, and to the dummy FUSE mounted atop the RAM disk. We found that the bandwidth achieved by FUSE was 80 MB/s, while the bandwidth of RAM disk was 1,610 MB/s. Due to the large overheads of using FUSE, the FUSE file system only gets approximately 5% of the performance of writing to RAM disk directly.

Location	*Throughput (MB/s)*
NFS	84.50
HDD	97.43
Parallel FS	764.18
SSD	1026.39
RAM disk	8555.26
Memory	15097.85

Table 1: I/O throughput for the storage hierarchy on the OSU-RI system described in Section 7.1

3.1.2 Linker-Assisted I/O Call Wrappers

The other alternative we considered for intercepting application I/O was to use a set of wrapper functions around the native POSIX I/O operations. The GNU Linker (`ld`) supports intercepting standard I/O library calls with user-space wrappers. This can be done statically during link-time, or dynamically at run time using `LD_PRELOAD`. This method works without significant overhead because all control remains completely in user-space without data movement to and from the kernel. The difficulty is that a significant amount of work is involved to write wrappers for all of the POSIX I/O routines that an application might use.

Two goals for CRUISE are portability and low overhead for checkpoint workloads, so in spite of the additional work required to write linker-assisted wrapper functions, we opted for this method due to its better performance and portability.

3.2 In-Memory File Storage

Table 1 illustrates the I/O throughput of different levels in the storage hierarchy. We show the performance for several stable storage options: the Network File System (NFS), spinning magnetic hard-disk (HDD), parallel file system, and solid-state disk (SSD). We also show the performance of two memory storage options, RAM disk and shared memory via a memory-to-memory copy operation (Memory). Of course, the memory-based storage options far out-perform stable storage. A key design goal of CRUISE is to store application checkpoint files in memory to improve performance and, more importantly, to serve as a local file system on HPC systems that provide no other form of local storage. Here, we discuss three options that we considered for in-memory storage, RAM disk, a RAM disk-backed memory map, and a persistent memory segment.

3.2.1 Kernel-Provided RAM disk

RAM disk is a kernel-provided virtual file system backed by the volatile physical memory on a node. RAM disk can be mounted like any other file system, and the data stored in it persists for the lifetime of the mount. The kernel manages the memory allocated to RAM disk, enabling persistence beyond the lifetime of user-space processes but not across node reboots or crashes. RAM disk also provides standard file system interfaces and is fully POSIX-compliant, making it a natural choice for in-memory data storage.

However, by comparing the RAM disk to the memory copy performance in Table 1, it is evident that RAM disk does not fully utilize the throughput offered by the physical memory subsystem. Another drawback with RAM disk is that one can not directly access file contents with RDMA.

3.2.2 *A RAM disk-Backed Memory-Map*

The drawbacks regarding performance and RDMA capability could be addressed by memory mapping a file residing in RAM disk. This approach could fully utilize the bandwidth offered by the physical memory subsystem simply by copying checkpoint data from application buffers to the memory-mapped region using `memcpy()`. Once the checkpoint is written to the memory-map, it can be synchronized with the backing RAM disk file using `msync()`. Then one can simply read the normal RAM disk file during recovery.

However, given that the file backing the memory-map resides in the memory reserved for RAM disk, the checkpoint data occupies twice the amount of space. Moreover, there are difficulties involved with tracking consistency between the memory-mapped region and the backing RAM disk file.

3.2.3 *Byte-Addressable Persistent Memory Segment*

The third approach we considered was to directly store the checkpoint data in physical memory. Our target systems all provide a mechanism to acquire a fixed-size segment of byte-addressable memory which can persist beyond the lifetime of the process that creates it. This includes systems such as the recent IBM Blue Gene/Q that provides so-called *persistent memory*, and all Linux clusters that provide System V IPC shared memory segments.

The downside of this method is that it requires implementation of memory allocation and management, data placement, garbage collection, and other such file system activities. In short, the difficulty lies in implementing the numerous functions and semantics of a POSIX-like file system.

The advantages are the fine-grained management of the data and access to the entire bandwidth of the memory device. Additionally, we expect this approach to work with future byte-addressable Non-Volatile Memory (NVM) or Storage Class Memory (SCM) architectures.

Although the use of a byte-addressable memory segment requires significant implementation effort to perform the activities of a file system, we chose this method for `CRUISE` for its portability and performance.

3.3 Limitations of the Kernel Buffer Cache

One could argue that the buffer cache maintained in the kernel is a viable alternative that satisfies most of the design goals for `CRUISE`. The benefits of using the buffer cache include fast writes, asynchronous flush of data to a local or remote file system, and dynamic management of application and file system memory.

However, the potential pitfalls of using the buffer cache in a multilevel checkpointing system outweigh these benefits. One, with multilevel checkpointing, there are situations wherein a cached checkpoint need not be persisted to stable storage. The kernel, however, cannot make this distinction and may unnecessarily flush all data in the buffer cache to the underlying storage system. Two, using the buffer cache involves copies between user and kernel space, reducing write throughput. Three, using the buffer cache does not permit direct access to data for the RDMA capability, which is desirable for asynchronous checkpoint staging. And four, we lose control over when data is moved from the compute node

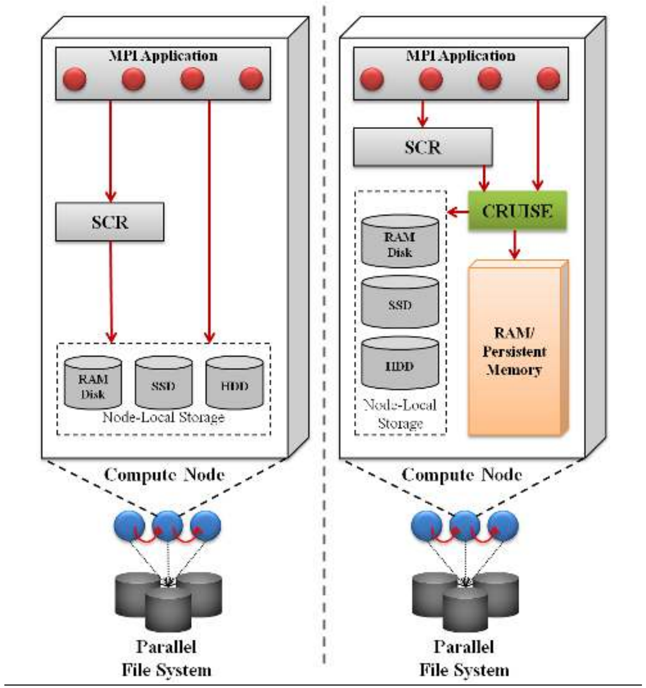


Figure 1: Architecture of `CRUISE`

to the remote file system. With an in-memory file system like `CRUISE`, we can orchestrate data movement such that it does not impact the performance of large-scale HPC applications with file system noise. `CRUISE` is an initial proof-of-concept system intended to work with byte-addressable NVM architectures that cannot be serviced by the buffer cache.

4. ARCHITECTURE AND DESIGN

In this section, we present our design of `CRUISE`. We begin with a high-level overview. We follow with details on simplifications we made to support checkpoint files, and our approaches for lock management, spill-over, and RDMA support.

4.1 The Role of CRUISE

In Figure 1, we show a high-level view of the interactions between components in SCR and `CRUISE`. On the left, we show the current state-of-the-art with SCR, and on the right, we show SCR with `CRUISE`. In both cases, all compute nodes can access a parallel file system. Additionally, each compute node has some type of node-local storage media such as a spinning disk, a flash memory device, or a RAM disk.

In the SCR-only case, the MPI application writes its checkpoints directly to node-local storage, and it invokes the SCR library to apply cross-node redundancy schemes to tolerate lost checkpoints due to node failures. For the highest level of resiliency, SCR writes a selected subset of the checkpoints to the parallel file system. By using SCR, the application incurs a lower overhead for checkpointing but maintains high resiliency. However, SCR cannot be employed on clusters with insufficient node-local storage.

In the SCR-`CRUISE` case, checkpoints are directed to `CRUISE`. All application I/O operations are intercepted by the `CRUISE` library. File names prefixed with a special mount name are processed by CRUISE, while operations for other file names are passed to the standard POSIX routines. CRUISE manages file data in a pre-allocated persistent memory region. Upon exhausting this resource, CRUISE transparently spills remaining file data to node-local storage or the parallel file system. This configuration enables applications to use SCR on systems where there is only memory or where node-local storage is otherwise limited.

As an additional optimization, CRUISE can expose the file contents stored in memory to remote direct memory access. When SCR determines that a checkpoint set should be written to the parallel file system, an asynchronous file-transfer agent running on a dedicated I/O node can extract this data via RDMA using an CRUISE API that lists the memory addresses of the blocks of the files.

4.2 Data Structures

The CRUISE file system is maintained in a large block of persistent memory. The size of this block can be specified at compile time or run time. So long as the node does not crash, this memory persists beyond the life of the process that creates it so that a subsequent process may access the checkpoints after the original process has failed. When a subsequent process mounts CRUISE, the base virtual address of the block may be different. Thus, internally all data structures are referenced using byte offsets from the start of the block. The memory block does not persist data through node failure or reboot. In those cases, a new persistent memory block is allocated, and SCR restores any lost files by way of its redundancy schemes.

Figure 2 illustrates the format of the memory block. The block is divided into two main regions: a metadata region that tracks what files are stored in the file system, and the data region that contains the actual file contents. The data region is further divided into fixed-size blocks, called *data-chunks*. Although not drawn to scale in Figure 2, the memory consumed by the metadata region only accounts for a small fraction of the total size of the block.

We assume that a CRUISE file system only contains a few checkpoints at a time, which simplifies the design of the required data structures. As discussed in Section 2.2, SCR deletes older node-local checkpoints once a new checkpoint has been written, freeing up space for newer checkpoints to be stored. Thus, we are safe to assume a small number of files exist at any time.

Because CRUISE handles a limited number of files for each process, we design our metadata structures to use small, fixed-size arrays. Each file is then assigned an internal *FileID* value, which is used as an index into these arrays. CRUISE manages the allocation and deallocation of FileIDs using the free_fid_stack. When a new file is created, CRUISE pops the next available FileID from the stack. When a file is deleted, its associated FileID is pushed back onto the stack. For each file, we record the file name in the *File List* array, and we record the file size and the list of data-chunks associated with the file in an array of *File Metadata* structures. The FileID is the index for both arrays.

CRUISE adds the name of a newly created file to the File List in its appropriate position, and sets a flag to indicate that this position is in use. For metadata operations that only provide the file name, such as open(), rename(), and unlink(), CRUISE scans the File List for a matching name to discover the FileID, which can then be used to index into the array of File Metadata structures. For calls which return a POSIX file descriptor, like open(), we associate a mapping from the file descriptor to the FileID so that subsequent calls involving the file descriptor can index directly to the associated element in the File List and File Metadata structure arrays.

The File Metadata structure is logically similar to an *inode* in traditional POSIX file systems, but it does not keep all of the metadata kept in inodes. The File Metadata structure simply holds information pertaining to the size of the file, the number of data-chunks allocated to the file, and the list of data-chunks that constitute the file.

Finally, the free_chunk_stack manages the allocation and deallocation of data-chunks. The size and number of data-chunks are fixed when the file system is created. Each data-chunk is assigned a *ChunkID* value. The free_chunk_stack tracks ChunkIDs that are available to be assigned to a file. When a file requires a new data-chunk, CRUISE pops a value from the stack and records the ChunkID in the File Metadata structure. When a chunk is freed, e.g., after an unlink() operation, CRUISE pushes the corresponding ChunkID back on the stack.

4.3 Spill Over Capability

Some HPC applications use most of the memory available on each compute node, and some also save a significant fraction of that memory during a checkpoint. In such cases, the memory block allocated to CRUISE may be too small to store the checkpoints from the processes running on the node. For this reason, we designed CRUISE to transparently spill over to secondary storage, such as a local SSD or the parallel file system.

During initialization, a fixed-amount of space on the spill-over device is reserved in the form of a file. As with the memory block, the user specifies the location and size of this file. The file is logically fragmented into a pool of data-chunks, and the allocation of these chunks is managed by the free_spillover_stack, which is kept in the persistent memory block. For each chunk allocated to a file, the File Metadata structure also records a field to indicate whether the chunk is in the memory or the spill-over device. When allocating a new chunk for a file, CRUISE allocates a chunk from the spill-over storage only when there are no remaining free chunks in memory.

4.4 Simplifications

We made simplifications over POSIX semantics in CRUISE for directories, permissions, and time stamps.

CRUISE does not support directories. However, CRUISE maintains the illusion of a directory structure by using the entire path as the file name. This support is sufficient for SCR and simplifies the implementation of the file system. When files are transferred from CRUISE to the parallel file system, the directory structure can be recreated since the full paths are stored.

CRUISE does not support file permissions. Since compute nodes on HPC systems are not shared by multiple users at the same time, there is no need for administering file permissions or access rights. All files stored within CRUISE can only be accessed by the user who initiated the parallel application. SCR restores normal file permissions when files are transferred from CRUISE to the parallel file system.

CRUISE does not track time stamps. SCR manages infor-

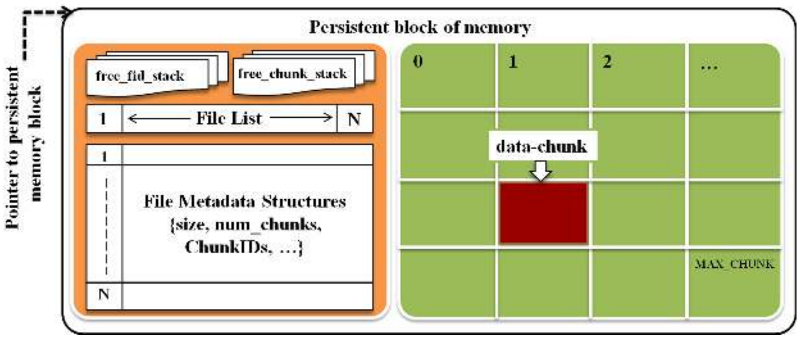

Pointer to persistent memory block
Persistent block of memory
free_fid_stack
free_chunk_stack
1
File List
N
1
File Metadata Structures {size, num_chunks, ChunkIDs, ...}
N
0
1
2
...
data-chunk
MAX_CHUNK

Figure 2: Data Layout of `CRUISE` on the Persistent Memory Block

mation about which checkpoints are most recent and which can be deleted to make room for new checkpoint files, so time stamps are not required. Typically, versioning mechanisms tend to be a mere sequential numbering of checkpoints, in the order in which they were saved. Updating time stamps on file creation, modification, or access incurs unnecessary overhead, so we remove this feature from `CRUISE`.

4.5 Lock Management

For some flexibility between performance and portability, the persistent memory block may either be shared by all processes running on a compute node, or there may be a private block for each process. The patterns of checkpoint I/O supported by SCR do not require shared-file access between MPI processes; in fact, SCR prohibits it. Given this, we can assume that no two processes will access the same data-chunk, nor will they update the same File Metadata structure. However when using a single shared block, multiple processes interact with the stacks that manage the free FileIDs and data-chunks. When operating in this mode, the push and pop operations must be guarded by exclusive locks.

Since stack operations are on the critical path, we need a light-weight locking mechanism. We considered two potential mechanisms for locking common data structures. One option is to use System V IPC semaphores and the other is to use Pthread spin-locks. Semaphores provide a locking scheme with a high-degree of fairness, and processes sleep while waiting to acquire the lock, freeing up compute resources. However, the locking and unlocking routines are heavy-weight in terms of the latency incurred. Spin-locks, on the other hand, provide a low-latency locking solution, but they may lack fairness and can lead to wasteful busy-waiting.

When using SCR, all processes in the parallel job synchronize for the checkpoint operation to complete before starting additional computation. This synchronization ensures some degree of fairness between processes across checkpoints. Furthermore, in the case of HPC applications, busy-waiting on a lock does not reduce performance since users do not oversubscribe the compute resources. Thus, we elected to use spin-locks in `CRUISE` to protect the stack operations.

4.6 Remote Direct Memory Access

RDMA allows a process on a remote node to access the memory of another node, without involving a process on the target node. The main advantage of RDMA is the *zero-copy communication* capability provided by high-performance interconnects such as InfiniBand. This allows the transfer of data directly to and from a remote process' memory, bypassing kernel buffers. This minimizes the overheads caused by context switching and CPU involvement.

Several researchers have studied the benefits of RDMA-based asynchronous data movement mechanisms [4, 5, 23]. An I/O server process can pull checkpoint data from a compute node's memory without requiring involvement from the application processes, and then write the data to slower storage in the background. This reduces the time for which an application is blocked while writing data to stable storage.

A vast majority of the asynchronous RDMA-based data movement libraries have two sets of components: one or more local RDMA agents that reside on each compute node, and smaller pool of remote RDMA agents hosted on storage nodes or dedicated data-staging nodes. Typically, each data-staging RDMA agent provides data movement services for a small group of compute nodes rather than serving all of them, making this a scalable solution. On receiving a request to move a particular file to the parallel file system, the compute-node RDMA agent reads a portion of the file from disk to its memory space, prepares it for RDMA, and then signals the RDMA agent on the data-staging node. However, the additional memory copy to read the file data into memory for RDMA incurs a significant overhead.

Given that the data managed in `CRUISE` is already in memory, this additional memory copy operation can be avoided by issuing in-place RDMA operations. To achieve this, we expose an interface for discovering the memory locations of files for efficient RDMA access in `CRUISE`. The local agent can then communicate the memory locations to the remote agent. This method eliminates the additional memory copies and enables the remote agent to access the files without further interaction with the local agent.

Figure 3 illustrates the protocol for the interface, which works by the following description: **(1)** On initialization, the local and remote RDMA agents establish a network connection for RDMA transfers. **(2)** The local RDMA agent uses the function `get_data_region()` exposed by `CRUISE` to get the starting address of the memory region in which `CRUISE` stores its data chunks, and the size of this memory region. The local RDMA agent then registers the memory region for RDMA operations. **(3)** Following this, the local RDMA agent sleeps until it receives a request from SCR to flush a checkpoint file to the parallel file system.

(4) On receiving a request from SCR, the local agent invokes `get_chunk_meta_list()` exposed by `CRUISE`, which returns a list of metadata information about each data chunk in the file. This includes the logical ChunkID, the memory

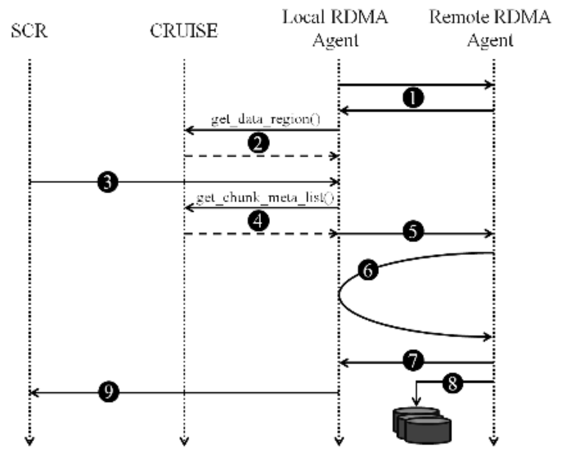


Figure 3: Protocol to RDMA files out of CRUISE

address of the chunk if it is in memory, the offset of the chunk if it is in a spill-over file, and a flag to indicate if the chunk is located inside the memory region or the spill-over file. If a chunk has been spilled-over to an SSD, the local agent issues a `read()` to copy that particular chunk to its address space before initiating an RDMA transfer. **(5)** Then, the local agent sends a control message to the remote agent with the information about the memory addresses to transfer. **(6)** The remote process reads the data chunks directly from the data region managed by `CRUISE`, without involving the local RDMA agent or the application processes.

(7) After the data has been read from the list of addresses, the remote agent sends a control message to the local agent informing it that it is safe for these buffers to be replaced for subsequent transfers. **(8)** The remote agent writes the data it receives into the parallel file system. Note that it is the duty of the remote agent to pipeline the loop of steps (5)-(8) to make optimum use of the network bandwidth and to overlap the communication and I/O phases. **(9)** When the file transfer is complete, the local agent informs SCR to complete the transfer protocol.

5. IMPLEMENTATION OF CRUISE

Here, we illustrate the implementation of the `CRUISE` file system by detailing initialization and two representative operations: the `open()` metadata operation and the `write()` data operation.

5.1 Initializing the File System

To initialize `CRUISE`, a process must mount `CRUISE` with a particular prefix by calling a user-space API routine. At mount time, `CRUISE` creates and attaches to the persistent memory block. It initializes pointers to the different data structures within this block, and it clears any locks which may have been held by previous processes. If the block was newly created, it initializes the various resource stacks. Once `CRUISE` has been mounted at some prefix, e.g., `/tmp/ckpt`, it intercepts all I/O operations for files at that prefix. For all other files, it forwards the call to the original I/O routine.

5.2 open() Operation

Figure 4 lists pseudo-code for the `open()` function. When `CRUISE` intercepts any file system call, it first checks to see if the operation should be served by `CRUISE` or if it should be passed to the underlying file system. In `open()`, `CRUISE` compares the *path* argument to the prefix at which it was mounted. `CRUISE` intercepts the call if the file prefix matches the mount point; otherwise it invokes the real `open()`.

```
open(const char *path, int flags, ...)
if path matches CRUISE mount prefix then
    lookup corresponding FileID
    if path not in File List then
        pop new FileID from free_fid_stack
        if out of FileIDs then
            return EMFILE
        end if
        insert path in File List at FileID
        initialize File Metadata for FileID
    end if
    return FileID + RLIMIT_NOFILE
else
    return __real_open(path, flags, ...)
end if
```

Figure 4: Pseudo-code for `open()` function wrapper

```
write(int fd, const void *buf, size_t count)
if fd more than RLIMIT_NOFILE then
    FileID = fd - RLIMIT_NOFILE
    get File Metadata for FileID
    compute number of additional data-chunks
    required to accommodate the write
    if additional data-chunks needed then
        pop data-chunks from free_chunk_stack
        if out of memory data-chunks then
            pop data-chunks from
            the free_spillover_stack
        end if
        store new ChunkIDs in File Metadata
    end if
    copy data to chunks
    update file size in File Metadata
    return number bytes written
else
    return __real_write(fd, buf, count)
end if
```

Figure 5: Pseudo-code for `write()` function wrapper

When `CRUISE` intercepts `open()`, it scans the File List to lookup the FileID for a file name matching the *path* argument. If it is not found, `CRUISE` allocates a new FileID from the `free_fid_stack`, adds the file to the File List, and initializes its corresponding File Metadata structure. As a file descriptor, `CRUISE` returns the internal FileID plus a constant `RLIMIT_NOFILE`. RLIMITs are system specific limits imposed on different types of resources, including the maximum number of open file descriptors for a process. The `CRUISE` variable `RLIMIT_NOFILE` specifies a value one greater than the maximum file descriptor the system would ever return. `CRUISE` differentiates its own file descriptors from system file descriptors by comparing them to this value.

5.3 write() Operation

Figure 5 shows the pseudo-code for the `write()` function. `CRUISE` first compares the value of *fd* to `RLIMIT_NOFILE` to determine whether *fd* is a `CRUISE` or system file descriptor. If it is a `CRUISE` file descriptor, `CRUISE` converts *fd* to a FileID by subtracting RLIMIT_NOFILE. Using the FileID, CRUISE looks up the corresponding File Metadata structure to obtain the current file size and list of data-chunks allocated to the file. From the current file pointer position and the length of the write operation, CRUISE determines whether additional data-chunks must be allocated. If necessary, it acquires new data-chunks from free_chunk_stack. If the persistent memory block is out of data-chunks, CRUISE allocates chunks from the secondary spill-over pool. It appends the ChunkIDs to the list of chunks in the File Metadata structure, and then it copies the contents of *buf* to the data-chunks. CRUISE also updates any relevant metadata such as the file size.

6. FAILURE MODEL WITH SCR

CRUISE is designed with the semantics of multilevel checkpointing systems in mind. The core principle of multilevel checkpointing is to use light-weight checkpoints, such as those written to CRUISE, to handle the most common failures. Less frequent but more severe failures restart the application from a checkpoint on the parallel file system. In this section, we detail the integration of CRUISE with SCR.

SCR supports HPC applications that use the Message Passing Interface (MPI). SCR directs the application to write its files to CRUISE, and after the application completes its checkpoint, SCR applies a redundancy scheme that protects the data against common failure modes. The redundancy data and SCR metadata are stored in additional files written to CRUISE. On any process failure, SCR relies on the MPI runtime to detect the failure and kill all remaining processes in the parallel job. Note that processes can fail or be killed at any point during their execution, so they may be interrupted while writing a file, and they may hold locks internal to CRUISE.

If a failure terminates a job, SCR logic in the batch script restarts the job using spare nodes to fill in for any failed nodes. During the initialization of the SCR library by the new job, each process first mounts CRUISE and then invokes a global barrier. During the mount call, CRUISE clears all locks. The subsequent barrier ensures that locks are not allocated again until all processes return from the mount call. After the barrier, each process attempts to read an SCR metadata file from CRUISE. SCR tracks the list of checkpoint files stored in CRUISE, and it records which files are complete. It deletes any incomplete files, and it attempts to rebuild any missing files by way of its redundancy encoding. If SCR fails to rebuild a checkpoint, it restores the job using a checkpoint from the parallel file system.

Note that because CRUISE stores data in persistent memory, like System V shared memory, data is not lost due to simple process failure. All processes in the first job can be killed, and processes in the next job can reattach to the memory and read the data. However, data is lost if the node is killed or rebooted. In this case, CRUISE creates a new, empty block of persistent memory, and SCR is responsible for restoring missing files using its redundancy schemes.

CRUISE also relies on external mechanisms to ensure data integrity. CRUISE relies on ECC hardware to protect file data chunks stored in memory, and it relies on the integrity provided by the underlying file system for data chunks stored in spill over devices. For this latter case, we only need to ensure that CRUISE synchronizes data to the spill over device when the application issues a sync() call or closes a file.

7. EXPERIMENTAL EVALUATION

Here we detail our experimental evaluation of CRUISE. We performed both single- and multi-node experiments to investigate the throughput and scalability of the file system.

7.1 Experimentation Environment

We used several HPC systems for our evaluation.

OSU-RI is a 178-node Linux cluster running RHEL 6 at The Ohio State University. Each node has dual Intel Xeon processors with 4 CPUs and 12 GB of memory. *OSU-RI* also has 16 dedicated storage nodes, each with 24 GB of memory and a 300GB OCZ VeloDrive PCIe SSD. We used the GCC compilers for our experiments, version 4.6.3.

Sierra and *Zin* are Linux clusters at Lawrence Livermore National Laboratory that run the TOSS 2.0 operating system, a variant of RHEL 6.2. Both of these are equipped with Intel Xeon processors. On Sierra, each node has dual 6-core processors and 24 GB of memory; and on Zin, each node has dual 8-core processors and 32 GB of memory. Both clusters use the InfiniBand QDR interconnect. The total node counts on the clusters are 1,944 and 2,916 respectively. We used the Intel compiler, version 11.1.

Sequoia is an IBM Blue Gene/Q system with 98,304 compute nodes. Each node has 16 compute cores and 16 GB of memory. The compute nodes run IBM's Compute Node Kernel and are connected with the IBM Blue Gene torus network. We used the native IBM compiler, version 12.1.

7.2 Microbenchmark Evaluation

In this section, we give results from several experiments to evaluate the performance of CRUISE. First, we explore the impact of NUMA effects on intra-node scalability. Next, we evaluate the effect of data-chunk sizes on performance. Finally, we evaluate the spill-over capability of CRUISE. All results presented are an average of five iterations.

7.2.1 Non-Uniform Memory Access

With the increase in the number of CPU cores and chip density, the distance between system memory banks and processors also increases. If the data required by a core does not reside in its own memory bank, there is a penalty incurred in access latency to fetch data from a remote memory bank. In order to evaluate this cost, we altered CRUISE so that memory pages constituting the data-chunks are allocated in a particular NUMA bank. Table 2 lists the outcome of our evaluation on a single node of OSU-RI.

OSU-RI nodes have 8 processing cores; 4 cores share a memory bank. The table shows the CRUISE bandwidth obtained by allocating a shared memory block for 4 process running on the first four CPU cores, either on the local bank, on the remote bank, or by interleaving pages across the two banks. The "local bank" case always delivers the best bandwidth, the "remote bank" case always performs the worst, and the "interleaved" case strikes a balance between the two. The difference is most exaggerated with 4 processes, for which local bandwidth is 8.3 GB/s compared to only 5.7 GB/s for remote. Thus, CRUISE bandwidth drops by more than 30% if we are not careful to allocate data-chunk memory appropriately. To this end, we determine on which core a process is running when it mounts CRUISE. We use this information to determine from which NUMA bank to allocate data chunks for this process. HPC applications

	Single Memory Block			N-Memory Blocks		
# Procs (N)	Local Bank	Remote Bank	Mixed	Local Bank	Remote Bank	Mixed
1	3.74	2.63	3.09	3.74	2.63	3.09
2	6.54	4.51	5.16	6.58	4.50	5.33
3	7.84	5.28	6.33	7.84	5.29	6.33
4	8.29	5.70	6.81	8.28	5.69	6.80

Table 2: Impact of Non-Uniform Memory Access on Bandwidth (GB/s)

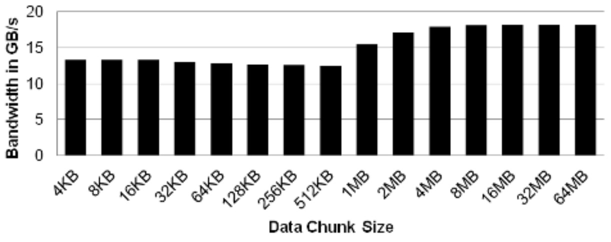


Figure 6: Impact of Chunk Sizes

typically pin processes to cores, so processes do not migrate from one NUMA bank to another during the run.

7.2.2 *Impact of Chunk Sizes*

One important parameter that affects the performance of `CRUISE` is the size of the data-chunk used to store file data. The chunk size determines the unit of data with which a `write()` or `read()` operation works. To study the impact of chunk sizes, we used the same benchmark from before in which 12 processes each write 64 MB of data to a file in `CRUISE` on a single node of Sierra. We then vary the chunk size from 4 KB up to 64 MB. In Figure 6, the x-axis shows the chunk size and the y-axis indicates the aggregate bandwidth obtained. As the graph indicates, we see performance benefits with larger chunk sizes. These benefits can be attributed to the fact that a file of a given size requires fewer chunks with increasing chunk sizes, which in turn leads to fewer bookkeeping operations and fewer calls to `memcpy()`. However, the aggregate bandwidth obtained here saturates that of the memory bank at 18.2 GB/s when chunks larger than 16 MB are used. Although this trend might remain the same across different system architectures, the actual thresholds could vary. To facilitate portability, we leave the chunk size as a tunable parameter.

In addition to having relatively larger chunks for performance reasons, it is also beneficial when draining checkpoints using RDMA as discussed in Section 4.6. One-sided RDMA `put` and `get` operations are known to provide higher throughput on high-performance interconnects such as InfiniBand when transferring large data sizes.

7.2.3 *Spill-over to SSD*

With the next set of experiments, we use a system with local SSD to evaluate the data spill-over capability in `CRUISE`. As discussed in Section 4.3, if the file data is too large to fit entirely in memory, `CRUISE` spills the extra data to secondary storage. In such scenarios, we can theoretically estimate the file system throughput using the following formula:

$$T_{spillover} = \frac{size_{tot}}{\frac{size_{MEM}}{T_{MEM}} + \frac{size_{SSD}}{T_{SSD}}}$$

Where, $T_{spillover}$ is the throughput with spill-over enabled; $size_{tot}$ is the total size of the checkpoint; $size_{MEM}$ is the size of the checkpoint stored in memory; $size_{SSD}$ is the size stored to the SSD; and T_{MEM} and T_{SSD} are the native throughput of memory and the SSD device.

We developed tests to study the performance penalties involved with saving parts of a checkpoint in memory and the rest to an SSD. Table 3 lists seven different test scenarios for a 512 MB-per-process checkpoint. Test #1 is the ideal scenario where 100% of the file is stored in memory, and Test #7 is the worst-case scenario where `CRUISE` must store the entire checkpoint to disk. With Tests #2-6, the size of the file that spills to the SSD increases by a factor of two.

All of these tests were run on a single storage node of OSU-RI that has a high-speed SSD installed. We first ran Tests #1 and #7 to measure the native throughput of memory and the SSD on the system, and we substituted these values into the above formula to compute the expected performance of the other cases. We then limited the memory available to `CRUISE` according to the test case, and conducted the other tests to measure the actual throughput. The theoretical and actual results are tabulated in Table 3.

Test #	% in SSD	Spill Size (MB)	Theoretical Throughput	Actual Throughput
1	0	0	15074.17	15074.17
2	3.125	16	10349.12	10586.61
3	6.25	32	7879.33	8134.46
4	12.5	64	5333.61	5312.26
5	25	128	3240.00	3110.58
6	50	256	1815.06	2163.93
7	100	512	965.67	965.67

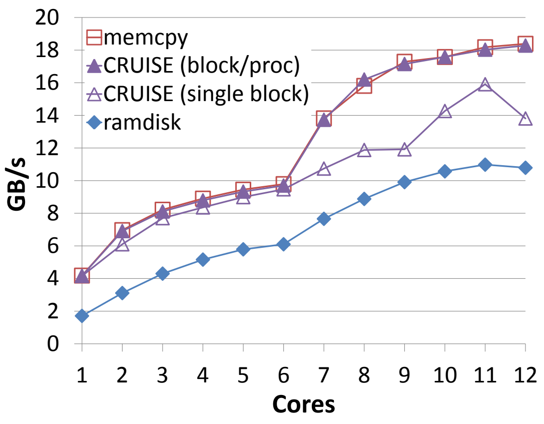
Table 3: `CRUISE` throughput (MB/s) with Spill-over

The experiment clearly shows that with an increase in the percentage of a checkpoint that has to be spilled to the SSD or any such secondary device, the total throughput of the checkpointing operation reduces. For instance, in case of Test#6, with exactly half the checkpoint spilling to the SSD, the total throughput is reduced by almost 86%. Also, the actual results closely match the theoretical estimates, which validates our basic formula.

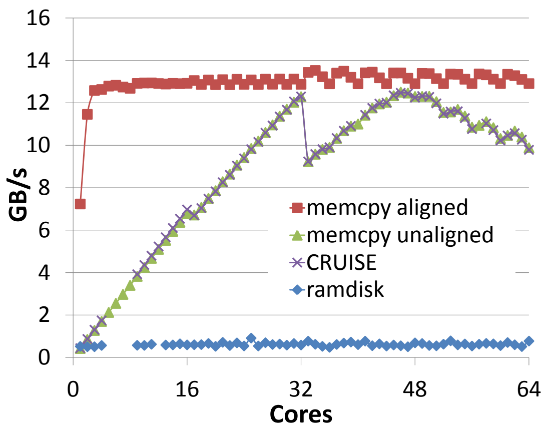
7.3 Intra-Node Scalability

In Figure 7, we show the intra-node scalability of `CRUISE` compared with RAM disk and a `memcpy()` operation on a single node of Sierra and Sequoia. The x-axis indicates the number of processes on the node, and the y-axis gives the aggregate bandwidth of the I/O operation in GB/s summed across all processes. Each process is bound to a single CPU-core of the compute node and writes and deletes a file five times, reporting its average bandwidth. On Sierra, the file size was 100 MB; on Sequoia, the file size was 50 MB.

The performance of the memory-to-memory copies repre-



(a) Sierra node



(b) Sequoia node

Figure 7: Intra-Node Aggregate Bandwidth Scalability

sents an upper bound on the performance achievable with our in-memory file system. To measure this bound, our benchmark simply copies data from one user-level buffer to another using standard `memcpy()` calls (red lines in Figure 7). The maximum aggregate bandwidth tops out around 18 GB/s on Sierra and roughly 13 GB/s on Sequoia.

One notable trend in the plot for Sierra is the double-saturation curve. Sierra is a dual-socket NUMA machine with 6 cores per NUMA bank. As the process count increases from 1 to 6, all processes are bound to the first socket and the performance of the local NUMA bank begins to saturate. Then, as the process count increases to 7, the seventh process runs on the second socket and uses the other NUMA bank leading to a jump in aggregate performance. Finally, this second NUMA bank begins to saturate as the process count is increased further from 7 to 12.

On Sequoia, each node has 16 compute cores, each of which supports 4-way simultaneous multi-threading. Therefore, we can evaluate the aggregate throughput for up to 64 processes on a node. On this system, we found a significant difference in `memcpy` performance depending on how buffers are aligned. If source and destination buffers are aligned at 64-byte boundaries, a fast `memcpy` routine is invoked that utilizes Quad Processing eXtension (QPX) instructions. Otherwise, the system falls back to a more general, but slower `memcpy` implementation. We plot results for both versions. The aligned memory copies (red line) saturate the physical memory bandwidth with a small number of parallel threads. It delivers a peak bandwidth of 13.5 GB/s with 32 processes. The unaligned variant (green line) scales linearly up to 32 processes where it reaches its peak performance of 12 GB/s.

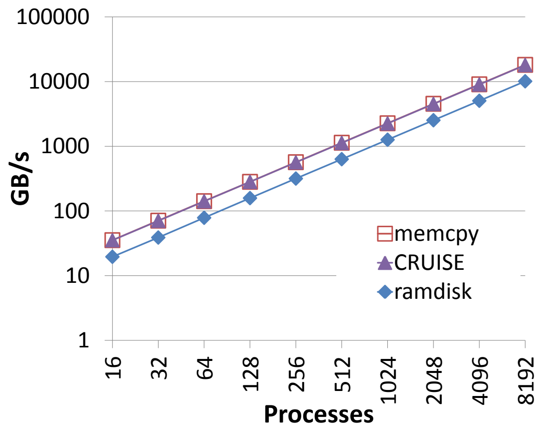
We do not see the double-saturation curves as in the case of Sierra, because the compute nodes on Blue Gene/Q systems have a crossbar switch that connects all cores to all of memory, so there are no NUMA effects. However, there are some interesting points where trends change significantly. The Blue Gene/Q architecture configures hardware as though the total number of tasks is rounded up to the next power of two in certain cases. These switch points apparently impact the memory bandwidth available to the tasks, particularly when going from 16 to 17 processes per node and again from 32 to 33. Beyond 32 processes per node, memory bandwidth initially drops but increases to another saturation point with about 45 processes. For process counts from 45 to 64, memory bandwidth steadily decreases again. We are still investigating the reason why memory bandwidth is affected this way. Having said that, applications are unlikely to run with process counts other than powers of two on a node.

We now examine the RAM disk performance (blue lines). With each iteration, each process in our benchmark writes and deletes a file in RAM disk. On Sierra, the aggregate bandwidth for RAM disk is nearly half of that for `memcpy`. On Sequoia, the performance is even worse. The memory copy performance increases with increasing cores, but the RAM disk performance is flat at ~ 0.6 GB/s.

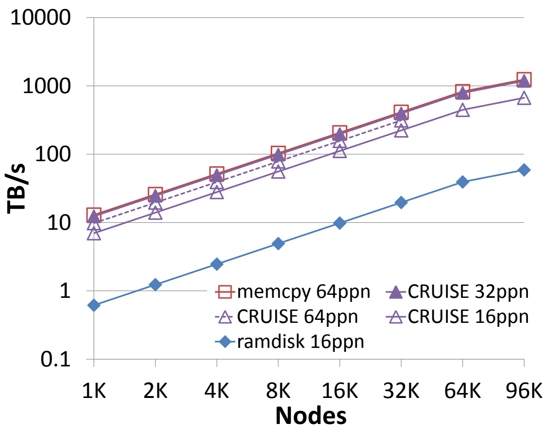
On Sierra, we evaluated the performance of `CRUISE` with a private block per process (purple, filled triangle) and with all processes on the node sharing a single block (purple, hollow triangle). There is a clear difference in performance between these modes. When using private blocks, the performance of `CRUISE` is close to that of `memcpy`, achieving nearly the full memory bandwidth. With a single shared block, `CRUISE` closely tracks the `memcpy` performance up to 6 processes, but then it falls off that trend with higher process counts.

A portion of the difference is due to locking overheads. However, experimental results showed these effects to be small for the 64 MB data-chunk size used in these tests. Instead, the majority of the difference appears to be due to the costs of accessing non-local memory. To resolve this problem, we intend to modify `CRUISE` to manage a set of free chunks for each NUMA bank and then select chunks from the appropriate bank depending on the location of the process making the request.

On Sequoia, we currently do not make an effort to align buffers in `CRUISE`. `CRUISE` has control over the alignment of the data-chunks, but it has no control over the offset of the buffers passed by the application. Thus, the performance of `CRUISE` (purple line) closely follows that of the unaligned `memcpy` (green line). We could modify `CRUISE` to fragment data-chunks and use aligned buffers more often. This would boost performance at the cost of using more storage space, but it could be a worthwhile optimization for large writes.



(a) Zin Cluster (Linux)



(b) Sequoia Cluster (IBM Blue Gene/Q)

Figure 8: Aggregate Bandwidth Scalability of CRUISE

7.4 Large-Scale Evaluation

CRUISE is designed to be used with large-scale clusters that span thousands of compute nodes. We evaluated the scaling capacity of this framework, and we show the results in Figure 8. We conducted these evaluations on Zin and Sequoia. For each of these clusters, we measured the throughput of CRUISE with increasing number of processes. In these experiments, we configured CRUISE to allocate a persistent memory block per process. On Zin, each process writes a 128MB file; on Sequoia, each writes a 50MB file. We compare CRUISE to RAM disk and a memory-to-memory copy of data within a process' address space using `memcpy()`. Since CRUISE requires at least one memory copy to move data from the application buffer to its in-memory file storage, the `memcpy` performance represents an upper-bound on throughput.

On Zin (Figure 8(a)), the number of processes writing to CRUISE was increased by a factor of two up to 8,192 processes along the x-axis. The y-axis shows the bandwidth(GB/s) in log-scale. As the graphs indicate, a perfect-linear scaling can be observed on this cluster. Furthermore, CRUISE takes complete advantage of the memory system's bandwidth (the CRUISE plot overlaps the `memcpy` plot). The throughput of CRUISE at 8,192 processes is 17.6 TB/s, which is only slightly below the `memcpy` throughput of 17.7 TB/s. The throughput of RAM disk is nearly half that of CRUISE at 9.87 TB/s. These runs used 17.5% of the available compute nodes. Extrapolation of this linear scaling to the full 46,656 processes would lead to a throughput for CRUISE of over 100 TB/s.

Figure 8(b) shows the scaling trends on Sequoia. Because Sequoia is capable of 4-way simultaneous multi-threading, a total of 6,291,456 parallel tasks can be executed. The x-axis provides the node-count for each data point, and the y-axis shows the bandwidth(TB/s) in log-scale. For clarity, we only show the configurations that deliver the best results for aligned `memcpy` and RAM Disk. We show the results when using 16, 32, and 64 processes per node for CRUISE. At the full-system scale of 6 million processes (64 processes/node), the aggregate aligned `memcpy` bandwidth reaches 1.21 PB/s. As observed in Figure 7(b), CRUISE nearly saturates this bandwidth to deliver a throughput of 1.16 PB/s when running with 32 processes per node. This is 20x faster than the system RAM disk, which provides a maximum throughput of 58.9 TB/s, and it is 1000x faster than the 1 TB/s parallel file system provided for the system.

8. RELATED WORK

Linker support to intercept library calls has been around for a while. *Darshan* [8] intercepts an HPC application's calls to the file system using linker support to profile and characterize the application's I/O behavior. Similarly, *fakechroot* [2] intercepts `chroot()` and `open()` calls to emulate their functionality without privileged access to the system.

Other researchers have investigated saving files in memory for performance. The MemFS project from Hewlett Packard [19] dynamically allocates memory to hold files. However, there is no persistence of the files after a process dies and MemFS requires kernel support. McKusick et al. present an in-memory file system [16]. This effort also requires kernel support, and it requires copies from kernel buffers to application buffers which would cause high overhead.

MEMFS is a general purpose, distributed file system implemented across compute nodes on HPC systems [27]. Unlike our approach, they do not optimize for the predominant form of I/O on these systems, checkpointing. Another general purpose file system for HPC is based on a concept called containers which reside in memory [15]. While this work does consider optimizations for checkpointing, its focus is on asynchronous movement of data from compute nodes to other storage devices in the storage hierarchy of HPC systems. Our work primarily differs from these in that CRUISE is a file system optimized for fast node-local checkpointing.

Several efforts investigated checkpointing to memory in a manner similar to that of SCR [7, 9, 13, 23, 29, 30]. They use redundancy schemes with erasure encoding for higher resilience. These works differ from ours in that they use system-provided in-memory or node-local file systems, such as RAM disk, to store checkpoints. Rebound checkpoints to volatile memory but focuses on single many-core nodes and optimizes for highly-threaded applications [6].

9. SUMMARY AND FUTURE WORK

In this work, we have developed a new file system called `CRUISE` to extend the capabilities of multilevel checkpointing libraries used by today's large scale HPC applications. `CRUISE` runs in user-space for improved performance and portability. It performs over twenty times faster than kernel-based RAM disk, and it can run on systems where RAM disk is not available. `CRUISE` stores file data in main memory and its performance scales linearly with the number of processors used by the application. To date, we have benchmarked its performance at 1 PB/s, at a scale of 96K nodes with three million MPI processes writing to it.

`CRUISE` implements a spill-over capability that stores data in secondary storage, such as a local SSD, to support applications whose checkpoints are too large to fit in memory. `CRUISE` also allows for Remote Direct Memory Access to file data stored in memory, so that multilevel checkpointing libraries can use processes on remote nodes to copy checkpoint data to slower, more resilient storage in the background of the running application.

As a next step, we would like to study the impact of in-memory checkpoint compression to conserve storage space. Furthermore, it is of our interest to investigate various caching policies, when using compression and spill-over capabilities, to improve I/O of frequently accessed file data.

Acknowledgments

The authors would like to thank the anonymous reviewers for their comments and suggestions that helped us improve the paper. This project is supported in part by NSF grants CCF-0937842 and OCI-1148371. (LLNL-CONF-592884)

References

[1] The ASC Sequoia Draft Statement of Work. `https://asc.llnl.gov/sequoia/rfp/02\_SequoiaSOW\_V06.doc`, 2008.

[2] fakechroot. `https://github.com/fakechroot/fakechroot/wiki`.

[3] Filesystem in Userspace. `http://fuse.sourceforge.net`.

[4] H. Abbasi, J. Lofstead, F. Zheng, S. Klasky, K. Schwan, and M. Wolf. Extending I/O through high performance data services. In *IEEE Cluster*, 2007.

[5] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng. DataStager: Scalable Data Staging Services for Petascale Applications. In *HPDC*, 2009.

[6] R. Agarwal, P. Garg, and J. Torrellas. Rebound: Scalable Checkpointing for Coherent Shared Memory. *SIGARCH Comput. Archit. News*, 2011.

[7] L. Bautista-Gomez, D. Komatitsch, N. Maruyama, S. Tsuboi, F. Cappello, and S. Matsuoka. FTI: High Performance Fault Tolerance Interface for Hybrid Systems. In *SC*, 2011.

[8] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Understanding and Improving Computational Science Storage Access through Continuous Characterization. 2011.

[9] B. Eckart, X. He, C. Wu, F. Aderholdt, F. Han, and S. Scott. Distributed Virtual Diskless Checkpointing: A Highly Fault Tolerant Scheme for Virtualized Clusters. *IEEE International Parallel and Distributed Processing Symposium Workshops*, 2012.

[10] E. N. Elnozahy and J. S. Plank. Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery. *IEEE Transactions on Dependable and Secure Computing*, 2004.

[11] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, T. Kordenbrock, and R. Brightwell. Increasing Fault Resiliency in a Message-Passing Environment. *Sandia National Laboratories, Tech. Rep. SAND2009-6753*, 2009.

[12] J. N. Glosli, K. J. Caspersen, J. A. Gunnels, D. F. Richards, R. E. Rudd, and F. H. Streitz. Extending Stability Beyond CPU Millennium: A Micron-Scale Atomistic Simulation of Kelvin-Helmholtz Instability. In *SC*, 2007.

[13] F. Isaila, J. Garcia Blas, J. Carretero, R. Latham, and R. Ross. Design and Evaluation of Multiple-Level Data Staging for Blue Gene Systems. *TPDS*, 2011.

[14] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman. ZOID: I/O-Forwarding Infrastructure for Petascale Architectures. In *PPoPP*, 2008.

[15] D. Kimpe, K. Mohror, A. Moody, B. V. Essen, M. Gokhale, K. Iskra, R. Ross, and B. R. de Supinski. Integrated In-System Storage Architecture for High Performance Computing. In *Workshop on Runtime and Operating Systems for Supercomputers*, 2012.

[16] M. McKusick, M. Karels, and K. Bostic. A Pageable Memory-Based Filesystem. In *Proceedings of the United Kingdom UNIX Users Group Meeting*, 1990.

[17] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer. *IEEE Transactions on Device and Materials Reliability*, 2005.

[18] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *SC*, 2010.

[19] H. Packard. MemFSv2 - A Memory-based File System on HP-UX 11i v2 . In *Technical Whitepaper*, 1990.

[20] F. Petrini. Scaling to Thousands of Processors with Buffer Coscheduling. In *Scaling to New Height Workshop*, Pittsburgh, PA, 2002.

[21] I. R. Philp. Software Failures and the Road to a Petaflop Machine. In *1st Workshop on High Performance Computing Reliability Issues (HPCRI)*, 2005.

[22] R. Ross, J. Moreira, K. Cupps, and W. Pfeiffer. Parallel I/O on the IBM Blue Gene/L System. Technical report, Blue Gene/L Consortium Quarterly Newsletter.

[23] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. R. de Supinksi, N. Maruyama, and S. Matsuoka. Design and Modeling of a Non-blocking Checkpointing System. In *SC*, 2012.

[24] B. Schroeder and G. Gibson. Understanding Failure in Petascale Computers. *Journal of Physics Conference Series: SciDAC*, June 2007.

[25] B. Schroeder and G. A. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. In *DSN*, June 2006.

[26] SCR. Scalable Checkpoint/Restart Library. `http://sourceforge.net/projects/scalablecr/`.

[27] J. Seidel, R. Berrendorf, M. Birkner, and M.-A. Hermanns. High-Bandwidth Remote Parallel I/O with the Distributed Memory Filesystem MEMFS. In *EuroPVM/MPI*. 2006.

[28] E. Vivek Sarkar, editor. *ExaScale Software Study: Software Challenges in Exascale Systems*. 2009.

[29] G. Wang, X. Liu, A. Li, and F. Zhang. In-Memory Checkpointing for MPI Programs by XOR-Based Double-Erasure Codes. In *EuroPVM/MPI*, 2009.

[30] G. Zheng, L. Shi, and L. V. Kalé. FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In *IEEE Cluster*, 2004.