Thinking Beyond the RAM Disk for In-Memory Checkpointing of HPC Applications

Raghunath Rajachandrasekar, Adam Moody, Kathryn Mohror,
Dhabaleswar K. Panda

OSU Technical report
OSU-CISRC-1/13-TR02

January, 2013

Network-Based Computing Laboratory
Dept. of Computer Science and Engineering
The Ohio State University
Columbus, OH

Lawrence Livermore National Laboratory
Livermore, CA

Thinking Beyond the RAM Disk for In-Memory Checkpointing of HPC Applications

Abstract

With the massive growth in scale and complexity of high performance computing (HPC) systems, long-running scientific parallel applications periodically save the state of their execution to files called checkpoints, which are used to recover from system failures. Checkpoints are stored on external parallel file systems, which persist data even through catastrophic machine failures. Limited bandwidth and contention, however, makes this a time-consuming operation. Multi-level checkpointing libraries, such as the Fault Tolerance Interface or the Scalable Checkpoint / Restart Library, have been developed to alleviate this bottleneck by caching most checkpoint files in storage close to the compute nodes and storing less frequent checkpoints to the slower, but more resilient parallel file system. However, most large scale HPC systems offer no storage on the compute nodes other than main memory, which significantly constrains the domain where existing multi-level checkpointing libraries can be employed.

In this paper, we implement a novel user-space file system that stores file data in main memory and transparently spills over to other storage like the parallel file system as needed. This technique extends the reach of multi-level checkpointing libraries to systems and applications where it otherwise could not be used. We discuss our design, the semantic assumptions we made, and other design alternatives. Our file system scales linearly with node count and achieves an aggregate throughput of 18 TB/s when using just 17.5% of a large HPC system. It decreases the time to write node-local checkpoints by a factor of 1.79 compared to RAM disk, and it can be ported to platforms where RAM disk does not exist.

Keywords HPC, checkpoint/restart, filesystems, persistent-memory, SSD, fault-tolerance

EuroSys '13 April 15-17, 2013, Prague

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 on the order of hours or days due to hardware breakdowns and soft errors [12, 17, 23, 24, 27].

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, 21]. 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 [11, 18].

Multi-level checkpointing systems [7, 18] utilize node-local storage for low-overhead, frequent checkpointing, and only write a select few checkpoints to the parallel file system. The use of node-local storage is appealing because it scales with the size of the application; as more compute nodes are used, more storage is available. Checkpoints can be taken more frequently so less work is lost upon failure. 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 multi-level 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. Our file system supports the 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 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:

- A thorough discussion and performance evaluation of design alternatives for our in-memory checkpointing system
- A detailed design of CRUISE
- A new mechanism for honoring memory usage bounds, namely, spill-over
- A performance and scalability evaluation of CRUISE and our spill-over mechanism

We give background in Section 2. We evaluate design alternatives for CRUISE in Section 3. In Sections 4 and 5, we detail our design and implementation of CRUISE. We follow this with a performance evaluation of CRUISE in Section 6. In Section 7, we survey related research.

2. Background

In this section, we give background on the checkpointing library for which we designed CRUISE and on the characteristics of checkpoint I/O workloads.

2.1 Scalable Checkpoint/Restart (SCR) Library

We developed CRUISE to enhance the capabilities of the Scalable Checkpoint/Restart (SCR) library [25]. SCR is a multi-level checkpointing system that enables MPI applications to use node-local storage to attain high checkpoint and restart I/O bandwidth [18]. SCR achieves high I/O bandwidth by caching checkpoints in node-local storage instead of the parallel file system. It caches checkpoint files in whatever storage is available, e.g., RAM disks, magnetic hard-drives, or SSDs. 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 more catastrophic failures.

To extend multi-level checkpointing to more systems and applications, the following design goals guided the development of CRUISE. First, we wanted to provide a file system on machines with no local storage other than memory. Second, we wanted a framework to support compression and spill over for applications whose checkpoints are too large to fit in the available local storage. Third, we wanted to enable remote access to checkpoint data stored in compute node memory so that it could be copied to slower, more resilient storage in the background. Finally, we wanted to develop a file system that could perform near memory speeds to allow for very low checkpointing overhead.

2.2 Checkpoint/Restart I/O Characteristics

The characteristics of checkpoint/restart I/O workloads differ from normal I/O workloads in several ways 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.

Predominantly sequential and dense writes. Typically, checkpoints are a large volume of data that is written sequentially to a file. However, there are exceptions. For instance, an application may use a high-level I/O library such as NetCDF [4] or HDF5 [9]. These libraries may use random writes to make updates to the header portion of a file. As another 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. A file system could optimize for this case by keeping track of the holes and the locations of the real data and use less space on the storage device. Because we assume checkpoints are dense files, we do not need to incur the overhead of supporting this optimization in CRUISE.

A single file per process. Many applications save state in a unique file per process. This checkpointing style is a natural fit for multi-level 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 greatly reduces the need for locking.

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 are assumed to be rare events relative to the number of checkpoints taken. With CRUISE, we plan to eventually utilize this property to conserve storage space by compressing files as they are written.

Transient nature of checkpoint data. Since an application restarts from its most recent checkpoint, older checkpoints can be discarded as newer checkpoints are written. Multi-level checkpointing libraries like SCR take advantage of this property by only caching the most recent checkpoints in node-local storage. Since SCR tracks which files are most recent, there is no need to track file time stamps in CRUISE.

Globally coordinated operation. Commonly, parallel applications coordinate to ensure that all message passing ac-

tivity has completed before taking 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.

3. Design Alternatives

CRUISE logically needs two layers of software: the first layer to intercept POSIX calls made by the application or checkpoint library, and the second layer to interact with the storage medium to manage file data. Each layer has several design alternatives that present trade-offs between imposed overheads, performance, portability, and the capability to support our design goals. This section discusses the merits and demerits of the alternatives that we considered.

3.1 Intercepting the Application's I/O Operations

One of our design goals was to develop a solution that is transparent to HPC applications. To achieve this, we want to intercept and process existing application checkpoint I/O operations such as `read()`, `fread()`, `write()`, and `fwrite()`, and metadata operations such as `open()`, `close()`, and `lseek()`. This subsection discusses two options that we considered for implementing the interception layer of CRUISE.

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.

Ignoring these portability issues for the moment, another problem is relatively poor performance for checkpointing workloads. First, because I/O data traverses between user-space and kernel-space multiple times, the use of 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, so that there are many context switches between the application and the kernel.

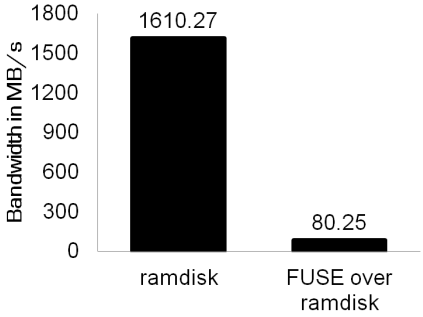

Figure 1. High-overheads in throughput incurred by FUSE

Figure 1 quantifies 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. 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. 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.

Because one of our design goals is to optimize CRUISE for low overhead for checkpointing workloads, and due to portability issues, we decided not to adopt this approach.

3.1.2 Linker-Assisted I/O Call Wrappers

The other alternative we considered for intercepting application I/O was to implement CRUISE as a set of wrapper functions over 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 the `LD_PRELOAD` mechanism. This method works without significant overheads 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 relevant POSIX I/O routines that an application might use. Nevertheless, we adopted this approach because of its low overhead and good portability.

3.2 In-Memory Checkpoint Storage Management

Another design goal for CRUISE is to store application checkpoint files in memory. Here, we discuss three options that we considered for this capability.

3.2.1 Kernel-Provided RAM disk

The RAM disk (a.k.a RAM disk) is a kernel-provided virtual file system that is backed by the volatile physical mem-

ory available to a node, and not by a hard-disk or any such storage device. RAM disk can be mounted like any other file system, and the data stored in it persists for the lifetime of the mount. The memory allocated to RAM disk is managed by the kernel, 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 an natural choice for in-memory data storage. However, the RAM disk implementation is unable to take complete advantage of the high-bandwidth of the memory subsystem.

Figure 2 illustrates the I/O throughput of different levels in the storage hierarchy. We show the performance for the Network File System (NFS), spinning magnetic hard-disk (HDD), parallel file system, solid-state disk (SSD), and RAM disk. In addition, we show the throughput of a memory-to-memory copy operation (`memcpy()`). From these numbers, 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. For these reasons, we chose not to use RAM disk to store checkpoint data.

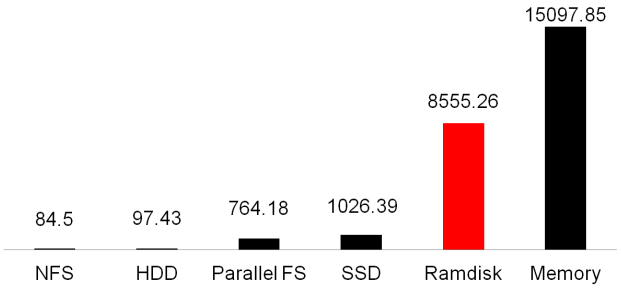

Figure 2. I/O Throughput (MB/s) at different levels of the storage hierarchy

3.2.2 A RAM disk -Backed Memory-Map

The drawbacks noted above regarding performance and RDMA capability could be addressed by memory mapping a file residing in RAM disk. This was an attractive option given that it could take complete advantage of the bandwidth offered by the physical memory subsystem simply by copying checkpoint data from application buffers to the memory-maps using `memcpy()` calls. 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.

Downsides to this approach are its space-demands and consistency concerns. Given that the file backing the memory-map also 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. We deemed the overheads incurred by this approach as too severe to be effective, and furthermore, some systems do not provide RAM disk on their compute nodes, so we chose not to use RAM disk for `CRUISE`.

3.2.3 Byte-Addressable Persistent Memory Segment

The third approach we considered was to 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. After allocating a block of such memory, our framework manages data-placement, garbage collection, and other such file system activities without relying on the kernel or any other system component. The difficulty lies in implementing the numerous functions and semantics of a POSIX-like file system. However, as we discuss in the following section, this is mitigated somewhat by using properties of checkpoint workloads combined with our target to provide a file system sufficient to support multi-level checkpointing libraries.

The major advantages of this design choice are the fine-grained management of the data and access to the entire bandwidth of the memory device. This design should also work with future byte-addressable Non-Volatile Memory (NVM) or Storage-Class Memory (SCM) architectures. Additionally, this design is portable across a wide-range of current HPC systems. This includes systems such as the recent BG/Q from IBM that provides byte-addressable persistent memory, and all Linux clusters that provide System V IPC shared memory segments. Because of these advantages, we chose this option for in-memory storage management for `CRUISE`.

4. Architecture and Design of `CRUISE`

4.1 `CRUISE` Design Overview

The contents of the file system are maintained in a large block of persistent memory. The size of this block is specified by the application via compile time constants, environment variables, or function parameters when the file system is created. The data in this memory block persists beyond the life of the process that creates it so that a subsequent process may attach to the block and access the data even after the original process has failed. We do not require data to persist through node failure or reboot, only through process lifetimes. When a subsequent process attaches to the block, the base virtual address of the block may be different, so internally all data structures are referenced using byte offsets from the start of the block.

Figure 3 illustrates the format of the memory block. The block is divided into two main regions: a meta data 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 3 for legi-

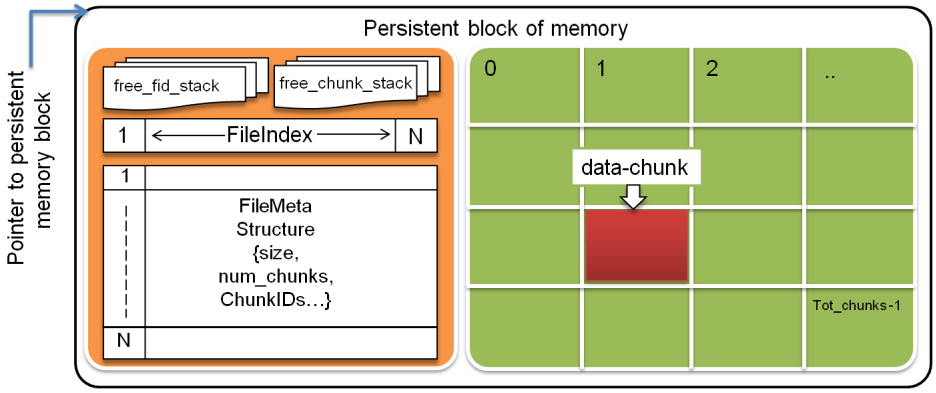

Figure 3. Data Layout of CRUISE on the Persistent Memory Block

bility, the memory consumed by the meta data region only accounts for a small fraction of the total size of the memory block.

4.2 CRUISE Components

We assume that our file system will not hold many files. At most, it will only contain a few checkpoints worth of files created by the processes running on the node. As discussed in Section 2.2, in globally-coordinated checkpointing with SCR, all processes of a parallel application write individual checkpoints simultaneously. Additionally, checkpoints stored locally are deleted once a new checkpoint has been written, freeing up space in the local node for newer checkpoints to be stored. We can safely assume that CRUISE needs to only handle a limited number of files for each process, and so we design our meta data structures to use small, fixed-size arrays. Each file is then assigned an internal *FileID* value, which is used to index into these arrays.

The allocation and deallocation of FileIDs is governed by 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 an array called the *File List*, and we record the file size and the list of data-chunks associated with the file in an array of *File Metadata* structures. Both arrays are indexed by FileID.

The name of a newly created file is added to the File List in its appropriate position, and a flag is also set to indicate that this position is in use. For meta data calls 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 is then 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 carry a majority of the Metadata carried by inodes. The File Metadata structure 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 ChunkID 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 Simplifications

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 the file system is simplified by not requiring it to track directory trees. When files are transferred from CRUISE to the parallel file system, the directory structure can be recreated since the full paths are stored.

CRUISE also does not support 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.

Nor does CRUISE track time stamps. SCR and other checkpoint libraries manage information about which checkpoints are most recent and which can be deleted to make room for new checkpoint files, so time stamps are not required. Updating time stamps on file creation, modification,

or access can incur a significant amount of overhead, so we remove this feature from CRUISE.

4.4 Lock Management

For improved portability, the persistent memory block may either be shared by all processes running on the same 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 accesses. 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 do 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 inter-process locks.

Given that the 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 lack fairness and can cause excessive busy-waiting.

When using SCR, all parallel processes in the job synchronize for the checkpointing 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.

4.5 Data Spill-Over to SSD

Some HPC applications tend to use most of the memory available on each compute node, and some of these also must save a significant fraction of that memory during a checkpoint. In such cases, the memory allocated for exclusive use by CRUISE might not be large enough to hold the checkpoints from the processes running on the node. We need a fail-over mechanism to handle such cases.

In order to fulfill this design goal, we implemented a data spill-over mechanism to complement the in-memory file system. We use a secondary storage device for this purpose, such as an SSD local to each compute node or the parallel file system. During initialization, we reserve a fixed-amount of space on the spill over device in the form of a file. As with the memory block, the user can specify the location and size of the file that CRUISE creates. The file is then logically fragmented into a pool of data-chunks. The allocation of these chunks is managed by another stack called the free_spillover_stack, which is kept in the persistent memory block. In addition to storing the list of ChunkIDs allocated to a file in the File Metadata structure, we also record a field indicating whether this chunk is in memory or the spill over device. When allocating a new chunk for a file, CRUISE allocates a chunk from the spill-over only when there are no remaining free chunks in memory.

4.6 Supporting Remote Direct Memory Access

Another important design goal of CRUISE was to support Remote Direct Memory Access (RDMA) to the file data stored in memory. RDMA allows a process on a remote node to access the memory of another node, without involving any process on the target node. The main advantage of RDMA is the *zero-copy communication* capability provided by high-performance interconnects such as InfiniBand and iWARP. This allows the transfer of data directly to and from a remote process's memory, bypassing any kernel buffers in the process. This minimizes the overheads caused by context switching and CPU involvement.

Several researchers have studied the benefits of RDMA-based asynchronous checkpointing mechanisms [5, 22]. 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 more stable storage.

As an optimization for such capabilities, we expose file data stored in CRUISE for RDMA access. To do this, CRUISE provides an interface to obtain the memory locations of the data-chunks stored for a file. The function takes a file descriptor as input and returns a list of addresses of the data-chunks that constitute the file. After the checkpoint files have been written, SCR can query this information, prepare the regions for RDMA, and pass the information along to a remote process.

5. Implementation of CRUISE

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

5.1 Initializing the Filesystem

Before CRUISE processes any I/O calls, a process must mount CRUISE at a particular prefix by calling a user-space API routine. At mount time, CRUISE creates and attaches to the persistent memory block for the process. 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, CRUISE simply forwards the call to the original I/O routine.

```
1: open(const char *path, int flags, ...)
2: if path matches CRUISE mount prefix then
3:         lookup corresponding FileID
4:         if path not in File List then
5:                 pop new FileID from free_fid_stack
6:                 if out of FileIDs then
7:                         return EMFILE
8:                 end if
9:                 insert path in File List at FileID
10:                initialize File Metadata for FileID
11:        end if
12:        return FileID + RLIMIT_NOFILE
13: else
14:        return __real_open(path, flags, ...)
15: end if
```

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

5.2 `open()` Operation

Figure 4 gives the 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. During the `open()` call, CRUISE compares the *path* argument to the prefix at which it was mounted. The call is intercepted if the file prefix matches the mount point; otherwise the real `open()` call is invoked.

When CRUISE intercepts an `open()` call, 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-generated 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 it corresponds to a CRUISE or system-generated file descriptor. If it is a CRUISE file descriptor, it converts *fd* to a FileID by subtracting `RLIMIT_NOFILE`. Using the FileID, CRUISE looks up the corresponding File Metadata structure to obtain information on 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 the `free_chunk_stack`. If the persistent memory block is out of data-chunks, CRUISE requests chunks from the `free_spillover_stack` to allocate 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 of *buf* to the data-chunks. Relevant meta data such as the file size is also updated.

6. Experimental Evaluation

6.1 Experimentation Environment

We used several HPC compute systems for our evaluation.

Cluster-A. Cluster A is a 160-node Linux cluster running RHEL 6. Each node has dual Intel Xeon processors with 4 cores. The compute nodes have 12 GB of memory and are connected with InfiniBand QDR. Cluster A 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.

Clusters *B, C, and D* are Linux clusters that run the TOSS 2.0 operating system, derived from Red Hat Enterprise Linux Server release 6. They are all equipped with Intel Xeon processors. On Cluster-B, each node has dual 6-core processors and 24 GB of memory; on Cluster-C and Cluster-D, each node has dual 8-core processors and 32 GB of memory. All three clusters use the high-speed InfiniBand QDR interconnect. The total node counts on the clusters are 1,944, 1,296, and 2,916 for Cluster-B, Cluster-C, and Cluster-D, respectively. In our experiments, we used the Intel compiler, version 11.1.

6.2 Microbenchmark Evaluation

In this section, we give the results from several experiments to evaluate the inherent performance capabilities of CRUISE. First, we explore the impact of NUMA effects on intra-node scalability. Next, we evaluate the effect of data-chunk

```
1:  write(int fd, const void *buf, size_t count)
2:  if fd more than RLIMIT_NOFILE then
3:          FileID = fd - RLIMIT_NOFILE
4:          get File Metadata for FileID
5:          compute number of additional data-chunks required to accommodate write
6:          if additional data-chunks needed then
7:                  pop data-chunks from free_chunk_stack
8:                  if out of memory data-chunks then
9:                          pop data-chunks from free_spillover_stack
10:                 end if
11:                 store new ChunkIDs in File Metadata
12:         end if
13:         copy data to chunks
14:         update file size in File Metadata
15:         return number bytes written
16: else
17:         return __real_write(fd, buf, count)
18: end if
```

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

sizes on performance. Finally, we evaluate the spill-over capability of CRUISE.

6.2.1 NUMA Impact on Intra-Node Scalability

Because CRUISE is a node-local file system, it is critical to understand its behavior with increasing numbers of processes on a single node. With the increase in the number of CPU cores and chip density, the distance between system memory banks and the processors also increases. If the data required by a core does not reside in its own memory bank, there is a significant access latency penalty to fetch data from a remote memory bank. In this section, we study the impact of such Non-Uniform Memory Access (NUMA) architectures on the intra-node scalability of CRUISE.

In Figure 6, we show the scalability of CRUISE compared with RAM disk and memcpy with increasing numbers of processes on a single node of Cluster-B and Cluster-C. 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. In our benchmark, each process writes and deletes a 100 MB file five times and reports its average write bandwidth.

One notable trend in these plots is the double saturation curves. We recall that Cluster-B is a dual-socket NUMA machine with 6 cores on one NUMA bank and 6 on another. Similarly, Cluster-C is a dual-socket 16 core machine with two NUMA banks. As the process count increases from 1 to 6 on Cluster-B (1 to 8 on Cluster-C), all processes are bound to the first socket. All three tests apparently allocate memory from the local NUMA bank, and its performance begins to saturate. Then as the process count is increased to 7 on Cluster-B (9 on Cluster-9), the additional process runs on the other socket and uses memory from 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.

Now looking at the performance of the individual tests, we first consider the scalability of memory-to-memory copies (red line). The processes that write to memory simply copy data from one user-level buffer to another using standard `memcpy()` calls. This theoretically represents the best bandwidth that can be obtained from the memory subsystem, and it can be considered as an upper bound of what any in-memory file system might achieve. The maximum aggregate bandwidth tops out around 18 GB/s on Cluster-B, and around 35 GB/s on Cluster-C.

With this in mind, we now examine the ramdisk performance (blue line). We used the same benchmark as above but replaced the `memcpy()` calls with standard POSIX `write()` calls to write data to a file in ramdisk. We also added a call to `fsync()` after writing all data, and of course we open and close the file before calling `unlink()` to delete it. Although the data is eventually stored in the same memory subsystem, the aggregate bandwidth is nearly cut in half when writing data through the ramdisk file system interface. This also confirms the observation illustrated in Figure 2.

Now consider the plots for CRUISE. On Cluster-B, we evaluated its performance when using a private block per process (purple line) and when all processes on the node shared a single block (green line). There is a clear difference in performance between these modes. The private block case very closely tracks the performance of `memcpy()`, seemingly achieving full memory bandwidth on the system. The overall scalability of the file system is bounded only by the memory-

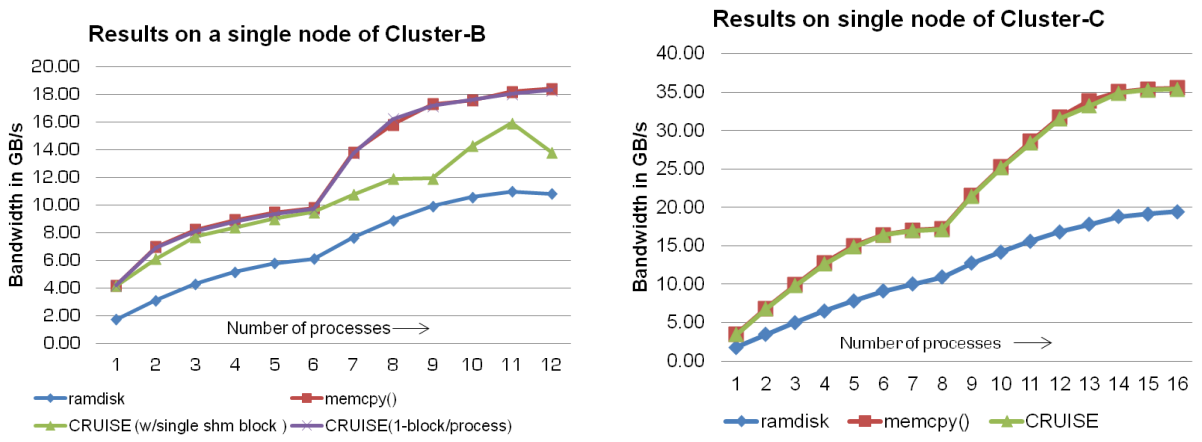

Figure 6. Impact of Non-Uniform Memory Access architecturesCRUISE

bandwidth offered by the system, and not by its inherent design.

The plot for the shared block mode also closely tracks the `memcpy()` performance up to 6 processes, but then it starts to fall off as the process count increases further. Although some of this is due to locking overheads, these overheads are small, especially for the 64 MB data-chunk size used in these tests. Instead, we believe this effect is due to the costs of non-local memory access, in which processes acquire data-chunks physically located on the remote NUMA node. This problem does not occur in the private-block case because by default the operating system allocates physical pages close to the core the process runs on. To address this finding, we plan to modify `CRUISE` to consider NUMA locations when allocating data-chunks.

6.2.2 Impact of Chunk Sizes

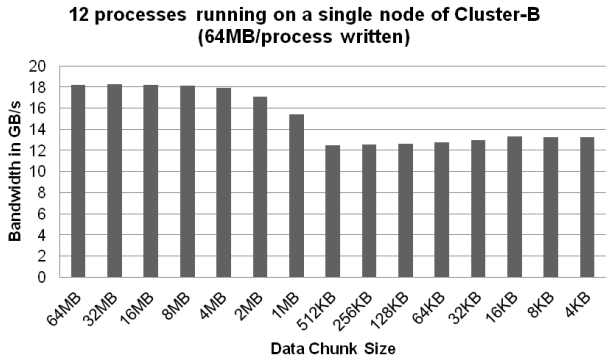

Figure 7. Impact of Chunk Sizes

One important parameter that determines the performance of our file system is the size of the data-chunk used to store file data. The chunk size also 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. We then vary the chunk size, starting from 4 KB and increasing it to 64 MB. In Figure 7, the X-axis shows the chunk size and the Y-axis indicates the aggregate bandwidth obtained. As the graph indicates, we see significant performance benefits with larger chunk sizes. This performance gain 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()`.

A drop in the throughput is observed as the chunk size nears 256 KB. This can be attributed to the on-chip cache hierarchy of the node on which these experiments were conducted, which had 256 KB of shared-L2 cache available for use by all cores on the socket. The throughput drops with increasing chunk size until it hits the L2 cache capacity, which saturates the throughput.

Similar to the contention at the cache level, the contention at the NUMA bank level shows up with very large chunk sizes. In our experiments, this overhead starts showing up after the chunk size is larger than 16 MB. 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.

6.2.3 Spill-over to SSD

With the next set of experiments, we evaluate the data spill-over capability that has been developed to complement CRUISE functionality. As discussed in Section 4.5, the spill-over technique acts as a fail-over mechanism, when HPC applications require a major amount of the physical memory available to the node, leaving a small fraction for the purpose of checkpointing. In such scenarios, it is possible to theoretically estimate the loss in checkpointing throughput that would be incurred when an application consumes a large amount of memory. The simple equation below explains how this can be done.

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

Where, $T_{spillover}$ is the throughput of the entire checkpoint with spill-over enabled, $size_{tot}$ is the total size of the checkpoint, $size_{CRUISE}$ is the size of the checkpoint that will be stored in CRUISE, $size_{SSD}$ is the size that will go into the SSD, T_{CRUISE} and T_{SSD} are the native throughput of CRUISE and the SSD device. Using such a model, users can proactively decide the ratio of memory reserved for applications and the checkpointing system in a balanced manner. We have compared the theoretical throughput obtained from this model, with experimental results, below.

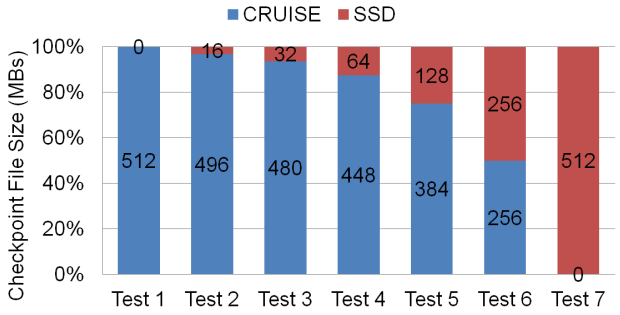

Figure 8. File-distribution patterns for SSD spill-over tests

To study the performance penalties involved in saving parts of a checkpoint in-memory, and rest of it in a spill-over device, we have developed a set of test-cases. Figure 8 lists 7 different test-cases based on a 512 MB-per-process checkpoint scenario. Along X-Axis are the different tests, and the Y-Axis represents the size of the checkpoint file. The stacked-bars indicate the size of the checkpoint stored inside CRUISE (blue part), and that stored in an SSD (red part). Test #1 is the most ideal scenario where 100% of the file is stored in memory, and Test.#7 is the worst-case scenario where the application consumed all of the node's memory forcing the entire checkpoint to be sent to disk. With tests 2-6, the size of the file that spills-over to the SSD increases by a factor of 2.

All these tests were run on a single storage node of Cluster-A that has an high-speed SSD installed. The native throughput of CRUISE and the SSD on this system where measured (Test#1 and #7) before running the other test cases, or computing the theoretical values. For each of the other tests, 12 processes write a 512 MB checkpoint file each. The memory available on CRUISE was limited according to the test case, and the aggregate throughput was measured. Both the theoretical and actual results have been tabulated in Table 1.

Test #	% in SSD	Theoretical Throughput	Actual Throughput
1	0	15074.17	15074.17
2	3.125	10349.12	10586.61
3	6.25	7879.33	8134.46
4	12.5	5333.61	5312.26
5	25	3240.00	3110.58
6	50	1815.06	2163.93
7	100	965.67	965.67

Table 1. CRUISE throughput (MB/s) with Data Spill-over

It is clearly understood, that with an increase in the percentage of a checkpoint that has to be spilled-over to the SSD or any such secondary device, the total throughput of the checkpointing operation reduces. Also, the actual results match fairly well with the theoretical ones that were computed using the equation.

6.3 Scalability Evaluation

CRUISE is designed to be used with large-scale clusters that span several-thousands of compute nodes. We have evaluated the scaling capacity of this framework, the results of the evaluation are shown in Figure 9. These evaluations were conducted on two different clusters that were described in Section 6.1: Cluster-B and Cluster-D. For each of these clusters, the throughput of CRUISE was measured, with increasing number of processes. In these experiments, each processes writes a 128MB file to the in-memory file system, with each process having its own persistent memory segment. To better understand the trends, we also compared this with the scaling capacity of the RAM disk. A memory-to-memory copy of data within a process' address space takes complete advantage of the memory sub-system's raw-bandwidth. In order to establish an upper-bound of achievable throughput, we have shown the scaling trends of the `memcpy()` operation as well.

On Cluster-B (Figure 9(a)), the number of processes writing to CRUISE was steadily increased by a factor of two (X-Axis), to run up to 6 144 processes. The Y-Axis shows the Bandwidth(GB/s) in log-scale. As the graphs indicate, CRUISE has a perfectly-linear scalability. It is able to take complete advantage of the memory system's bandwidth (the line of which is overlapping with the of CRUISE). At 6 144 processes, the throughput of CRUISE is 9.56 TB/s, losing a mere 0.44% of the 9.58 TB/s `memcpy()` throughput. The throughput of RAM disk at this scale was nearly half that

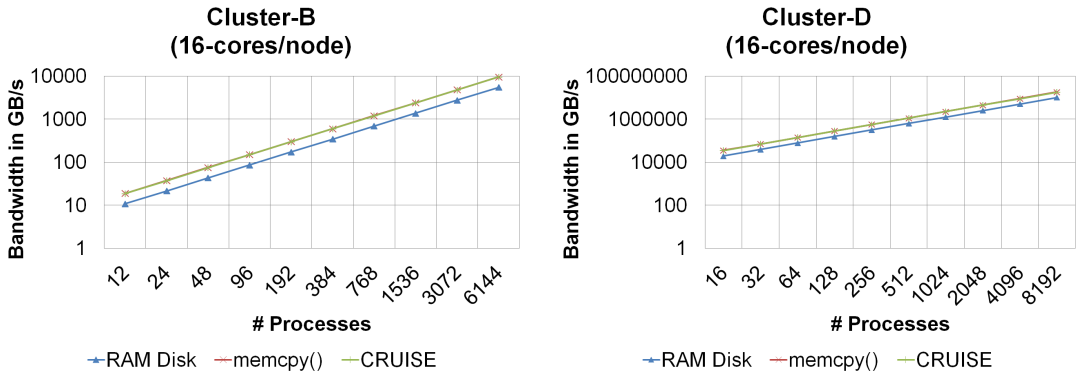

Figure 9. Scalability of CRUISE

of CRUISE at 5.52 TB/s, losing over 42% of the bandwidth provided by the memory system owing to heavy-weight semantics and multiple context-switches.

Likewise, on the Cluster-D (Figure 9(b)), the number of processes writing to CRUISE was increased up to 8 192 processes. A similar perfect-linear scaling can be observed on this cluster as well. The throughput of CRUISE at this scale is 18,031 MB/s (18.03 TB/s), with a loss of just 111 MB/s from the 18,142 MB/s throughput of `memcpy()`. Here again, the throughput of RAM disk was nearly half that of CRUISE, at 10,111 MB/s. These runs used just 17.5% of the number of nodes available on this cluster. At full-capacity of 46,656 processes, such linear-scaling would achieve a throughput of over 100 TB/s.

7. Related Work

The idea of using linker support to wrap and intercept library calls has been around for a while. The *Darshan* [8] project provides a runtime that intercepts an HPC application's calls to the filesystem using linker support to profile and characterize the application's I/O behavior. Similarly, the *fakechroot* [2] project 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. Another effort investigates the benefits of in-memory filesystems [16]. However, this effort also requires kernel support, and requires copies from kernel buffers to application buffers which would cause high overhead. MEMFS is a general purpose, distributed filesystem implemented across compute nodes on HPC systems [26]. Unlike our approach, they do not optimize for the predominant form of I/O on these systems, checkpointing. Another general purpose filesystem for HPC systems 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 is a filesystem optimized for fast node-local checkpointing. Several systems have looked at storing checkpoints in memory for high performance. Several efforts investigated checkpointing to memory in a manner similar to that of SCR [7, 10, 13, 22, 28, 29]. They use redundancy schemes with erasure encoding for higher resilience. Rebound focuses on single many-core nodes and optimizes for highly-threaded applications [6]. These works differ from ours in that they use existing in-memory or node-local filesystems to store checkpoints.

Summary and Future Work

In this work, we have developed a new file system called CRUISE to extend the capabilities of multi-level checkpointing libraries used by today's large scale HPC applications. CRUISE runs in user-space for improved performance and portability. It performs nearly twice as fast as the kernel-based RAM disk, and it can run on systems where RAM disk is not available. CRUISE stores file data in main memory and it scales linearly with the number of processors used by the application. To date, we have benchmarked its performance at 18 TB/s using just 17.5% of the nodes on a large scale cluster. At full scale, we expect CRUISE to reach write bandwidths of 100 TB/s.

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 multi-level 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 future work, we plan to conserve storage space by compressing checkpoint files as they are written. We will extend the storage logic in `CRUISE` to account for the effects of Non-Uniform Memory Access when allocating memory to maximize performance. Finally, we will investigate various caching policies when using compression and spill over capabilities to improve I/O to frequently accessed file data.

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] The NetCDF Users Guide. http://www.unidata.ucar.edu/software/netcdf/docs/netcdf.pdf.

[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*, 39, 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 *Proceedings of the 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, WS, USA, 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] A. Cheng and M. Folk. HDF5: High Performance Science Data Solution for the New Millennium. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2000.

[10] 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. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops*, 2012.

[11] 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*, 1(2):97 – 108, April-June 2004.

[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 *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (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 '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 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 (ROSS'12)*, 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*, 5(3): 329–335, September 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. MemFS v2 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] 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.

[22] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'12*, November 2012.

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

[24] B. Schroeder and G. A. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. In *In Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2006.

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

[26] J. Seidel, R. Berrendorf, M. Birkner, and M.-A. Hermanns. High-Bandwidth Remote Parallel I/O with the Distributed Memory Filesystem MEMFS. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. 2006.

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

[28] G. Wang, X. Liu, A. Li, and F. Zhang. In-Memory Checkpointing for MPI Programs by XOR-Based Double-Erasure Codes. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting*, 2009.

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