CRFS: A Lightweight User-Level Filesystem for Generic Checkpoint/Restart

Xiangyong Ouyang, Raghunath Rajachandrasekar, Xavier Besseron,
Hao Wang, Jian Huang, Dhabaleswar K. Panda
Department of Computer Science and Engineering
The Ohio State University
{ouyangx, rajachan, besseron, wangh, huangjia, panda}@cse.ohio-state.edu

***Abstract*—Checkpoint/Restart (C/R) mechanisms have been widely adopted by many MPI libraries [1–3] to achieve fault-tolerance. However, a major limitation of such mechanisms is the intensive IO bottleneck caused by the need to dump the snapshots of all processes into persistent storage. Several studies have been conducted to minimize this overhead [4, 5], but most of these proposed optimizations are performed inside specific MPI stack or checkpointing library or applications, hence they are not portable enough to be applied to other MPI stacks and applications.**

In this paper, we propose a filesystem based approach to alleviate this checkpoint IO bottleneck. We propose a new filesystem, named Checkpoint-Restart Filesystem (CRFS), which is a lightweight user-level filesystem based on FUSE (Filesystem in Userspace). CRFS is designed with Checkpoint/Restart I/O traffic in mind to efficiently handle the concurrent write requests. Any software component using standard filesystem interfaces can transparently benefit from CRFS's capabilities. CRFS intercepts the checkpoint file write system calls and aggregates them into fewer bigger chunks which are asynchronously written to the underlying filesystem for more efficient IO. CRFS manages a flexible internal IO thread pool to throttle concurrent IO to alleviate IO contention for better IO performance. CRFS can be mounted over any standard filesystem like ext3, NFS and Lustre. We have implemented CRFS and evaluated its performance using three popular C/R capable MPI stacks: MVAPICH2, MPICH2 and OpenMPI. Experimental results show significant performance gains for all three MPI stacks. CRFS achieves up to 5.5X speedup in checkpoint writing performance to Lustre filesystem. Similar level of improvements are also obtained with ext3 and NFS filesystems. To the best of our knowledge, this is the first such portable and light-weight filesystem designed for generic Checkpoint/Restart data.

***Keywords*-checkpoint-restart; userspace filesystem; write aggregation;**

I. Introduction

High performance computing clusters keep growing in terms of scale and complexity, which inevitably leads to more frequent failures of individual components. As a consequence, fault-tolerance has become a necessity.

Checkpoint-Restart (C/R) is the most widely deployed fault-tolerance mechanism. Hence it's highly desirable that MPI, the *de facto* standard for parallel programming, has built-in support for C/R. Many MPI implementations [1–3] provide blocking Checkpoint/Restart [6] support via a checkpoint library such as BLCR [7]. A typical checkpoint cycle consists of three phases. **Phase 1:** Build a consistent global state of the application. This is usually done by suspending the communication between all processes [8], or by identifying the in-transit messages using markers [9]. **Phase 2:** Use the checkpoint library to dump the memory snapshot of individual processes to a separate checkpoint file. **Phase 3:** Resume communications between processes and continue execution.

BLCR library performs a system-level checkpoint to save the entire context of each MPI process to a separate file. These checkpoint image files are later used to recover the application from a failure and restore its processes to a previous consistent state. Although effective to achieve fault-tolerance, Checkpoint/Restart mechanisms have gained notoriety for their bulky persistent storage space demands, intense I/O operations and heavy overheads they impose on application performance [10, 11].

We have carried out detailed profiling to gain insights into the IO overhead during checkpoint IO. We reveal that the high overhead can be ascribed to multiple reasons: (1) The sheer amount of data to be saved to stable storage; (2) For MPI stacks that rely on BLCR library to do checkpoint, BLCR performs large number of inefficient and relatively small writes to save their snapshots, which is especially unfavorable for the underlying filesystem to deliver good performance. These simultaneous write accesses cause severe IO contentions not only at inter-node level among processes from different compute nodes, but also at intra-node level as multi-core architecture is able to host more and more concurrent processes in a node. (3) In addition to

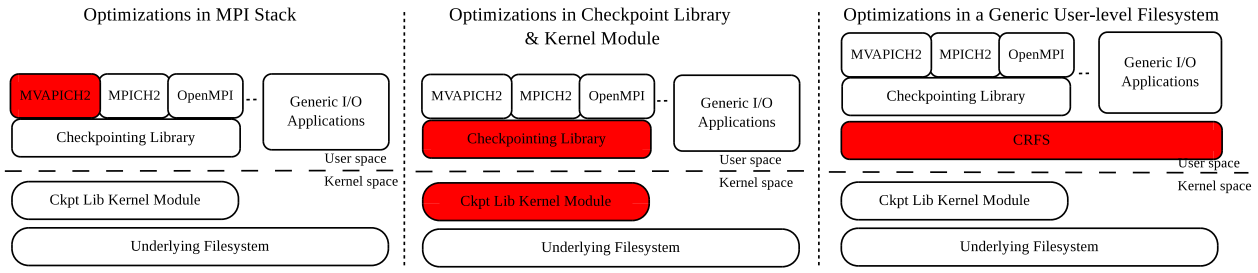


Fig. 1. Checkpoint/Restart Optimizations at Different Levels. Optimizations are Performed in Highlighted Components. Optimizations in MPI stack can only benefit certain MPI implementation. Optimizations in checkpoint library and Kernel module are not portable. Optimizations in a generic user-level filesystem is portable and can transparently benefit a wide range of MPI stacks and applications.

degraded IO throughput, the IO contention also leads to a large variation for individual process to complete checkpoint writing. Even if some processes finish their checkpoint writing quicker than others, they are forced to coordinate with the slower counterparts before they can resume execution.

A lot of efforts have been conducted to tackle the IO bottleneck incurred by C/R. The studies in [4, 12] modified the MPI implementation and BLCR library to alleviate the IO contention. Although effective, this approach only works for a specific MPI stack and requires patching BLCR kernel module, which isn't portable to be applied to generic environments. The authors of [11] proposed a parallel log-structured filesystem (PLFS) to improve the writing throughput. However this solution only deals with N-1 scenario where multiple processes write to the same shared file, hence it cannot handle MPI system-level checkpoint where each process is checkpointed to a separate image file.

In this paper we propose a scalable and portable solution which can significantly help a wide range of MPI stacks to reduce the IO bottleneck in Checkpoint/Restart. We want to address several questions:

- What are the primary factors to which the C/R overheads can be attributed?
- Can optimizations be performed in a user-level filesystem to improve C/R performance?
- Can such a user-level solution transparently benefit a wide range of MPI implementations, while taking advantage of various filesystems such as ext3, Lustre [13] or NFS?

We have implemented CRFS, a filesystem aimed for improving Checkpoint/Restart performance for a wide range of MPI stacks in a portable and transparent manner. In contrast to other alternative optimizations implemented in MPI libraries or in checkpoint libraries (as illustrated in left and middle part of Figure 1), CRFS is implemented as a user-level filesystem based on FUSE [14] which is readily available in most of the clusters. A wide spectrum of upper layer components, including any MPI implementation and other generic I/O applications, can transparently benefit from CRFS's capabilities. CRFS internally aggregates write streams from multiple processes into fewer bigger chunks, which are asynchronously written to back-end filesystem for more efficient IO. CRFS manages a flexible IO thread pool to throttle concurrent writing to alleviate the stress on back-end filesystems. CRFS can be mounted on top of any existing filesystem, such as ext3, PVFS2, NFS, and Lustre to leverage their capacity.

We have evaluated CRFS's performance in reducing checkpoint writing overhead with three popular C/R capable MPI stacks: MVAPICH2 [1], MPICH2 [3] and OpenMPI [2]. CRFS achieves significant improvement in all three MPI stacks to reduce checkpoint writing overhead. For application LU class C, CRFS is 5.5X faster than native Lustre filesystem in checkpoint writing. Checkpoint time with Lustre is reduced by 29% for LU class D. Up to 8X speedup is obtained if CRFS is used with ext3.

In summary, our contribution in this paper is as follows:

- Through detailed profiling, we have identified the dominant factors that determine the cost of Checkpoint-Restart.
- We have designed and implemented CRFS, a user-space filesystem, which optimizes concurrent checkpoint writing with the principle of write-aggregation. A wide range of upper layer components including any MPI stack and general IO applications can transparently benefit from CRFS's optimizations.
- We have conducted a comprehensive evaluation of the proposed design, and demonstrated a significant improvement in checkpoint writing performance with three popular MPI stacks.
- We have conducted intensive profiling to reveal the source of improvement achieved by CRFS.

The paper is organized as follows. In section II we give a background about the key components involved in our design. In section III we conduct profiling to reveal the causes of low IO efficiency in checkpoint writing. In Section IV we propose Checkpoint/Restart Filesystem (CRFS) that can enhance checkpoint IO performance for a wide range of applications. In section V, we present our experiments and evaluation. Related work is discussed in Section VI, and in section VII we present the conclusion and future work.

II. Background

A. Filesystem in Userspace FUSE)

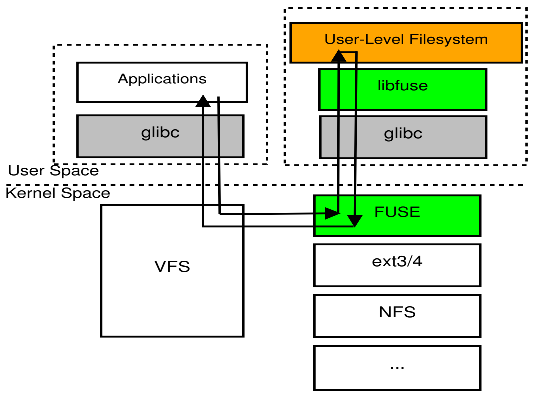


Fig. 2. FUSE Architecture (Courtesy of [15])

Filesystem in Userspace (FUSE) [14, 15] is a software that allows to create a virtual filesystem in the user level. As illustrated in Fig. 2, it relies on a kernel module to perform privileged operations at the kernel level and provides a userspace library that eases communication with this kernel module.

FUSE is widely used to create filesystems that do not really store the data itself but relies on other resources to effectively store the data. Thus, a FUSE-based virtual filesystem organizes data and presents it to the users through a classic filesystem interface.

B. Berkeley Lab Checkpoint/Restart (BLCR)

Berkeley Lab Checkpoint/Restart (BLCR) [7] allows programs running on Linux systems to be checkpointed by writing the process image to a file and then later be restarted from the saved process image file. BLCR by itself can only checkpoint/restart processes on a single node. But it provides callbacks to be extended by applications, so that a parallel application can also be checkpointed.

C. Checkpointing Mechanisms in MPI

In this paper, we have used three popular MPI libraries to evaluate the performance of CRFS - MVAPICH2, MPICH2 and OpenMPI. All three libraries have similar Checkpoint-Restart mechanisms, based on the BLCR checkpointing library. As a first step, when a Checkpoint request is made or when an automatic Checkpoint is scheduled, the MPI library flushes out the communication channel to build a consistent global state. Once the communication has been suspended, the BLCR library is used to dump a snapshot of the memory contents of all the MPI processes in the current job to stable storage. Once all the process states have been recorded, the communication channel is reactivated again for the job to resume execution. In the event of a failure, the BLCR library is used to restart the job from the process snapshots that were saved in the previous step. This simple three-step checkpoint-restart mechanism is uniform across the three MPI libraries.

III. Checkpoint Writing Profiling

In order to gain insights into the checkpoint writing characteristics, we executed NAS parallel benchmark [16] with MVAPICH2 [1], and took a checkpoint to node-local ext3 filesystem. We extended the BLCR library to record the information for all write operations, including number of writes, size of a write and time cost for each write.

TABLE I
Checkpoint Writing Profile (LU.C.64, write to ext3, Courtesy of [4])

Write Size	% of Writes	% of Data	% of Time
0-64	50.86	0.04	0.17
64-256	0.61	0.00	0.00
256-1K	0.25	0.01	0.00
1K-4K	9.46	1.53	0.01
4K-16K	36.49	11.36	44.66
16K-64K	0.74	0.77	6.55
64K-256K	0.49	3.79	11.80
256K-512K	0.25	3.58	1.75
512K-1M	0.61	17.72	14.72
$>$ 1M	0.25	61.21	20.35

Table I represents the checkpoint profiling for application LU.C.64 running on 8 compute nodes with 8 processes per node. Around 8 seconds are taken for a checkpoint to complete. During a checkpoint, each process generates a 23 MB snapshot to be saved, and a total of 7800 *write()* system calls are performed by all the 8 processes on a same node. We observed that a large number of small writes (64 bytes to 1 KB) cost a tiny fraction of time because they are quickly absorbed by the VFS page cache. On the other hand, 37% of write accesses are in the medium range (4 K to 64 K) and are responsible for 50% of all checkpoint time, even though they only deliver 13% of the total data. During checkpoint writing, multiple processes simultaneously issue their write requests and each medium request needs new pages to be allocated in page cache. These concurrent write streams cause severe contentions in the VFS layer, leading to degraded performance. There are a few (less than 1%) very large writes ($\geq$256 KB) that contribute the majority of the data (80%). However large sequential writes are relatively efficient and they only cost 35% of checkpoint time. This observation implies that optimizations are needed to reduce medium write requests. In Section IV we will propose a filesystem-based design to coalesce medium writes into fewer more efficient larger write accesses.

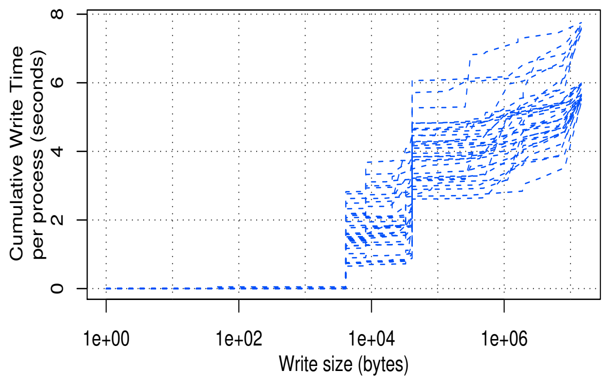


Fig. 3. Cumulative Write Time for Each Process (LU.C.64, ext3)

The contentions induced by the concurrent write requests not only degrade write throughput, but also create a large variation for individual processes to complete their writing. This is shown in Figure 3. Each line represents the time spent by a process to perform write operations, shown in a cumulative manner with respect to the write size. We observe a large variation in the process's completion time, ranging from 4 seconds to 8 seconds. Some processes are able to finish their writing very quick, however they have to coordinate with slower counterparts to re-establish communication channels before resuming execution (as is discussed in Section I). Consequently all processes are delayed by the progress of the slowest processes, resulting in a longer checkpoint time overhead.

IV. CRFS Design and Implementation

In this section we propose our approach to tackle the challenge of checkpoint IO. Unlike previous work that performs optimization inside certain checkpoint library [4] or within a specific MPI [5], which restricts its usability due to limited portability, we propose a general purpose user level filesystem, called CRFS, which can greatly enhance checkpoint writing performance for a wide range of applications including any MPI stacks. CRFS internally aggregates write requests from multiple processes into bigger-sized chunks and writes these chunks asynchronously to back-end filesystems. It maintains an IO thread pool with dedicated threads responsible for actual file writing. By configuring these IO threads, it's able to throttle concurrent write requests for better IO performance.

As depicted in Figure 4, CRFS is built on top of FUSE [14] and runs in user space as a stackable filesystem. CRFS relies on other filesystems to store the real file data, such as ext3/4, NFS, and Lustre [13]. Users can perform any POSIX filesystem operations in CRFS as in any other regular filesystems. These system calls are intercepted by FUSE kernel module and then routed to CRFS, where proper actions are taken by calling corresponding functions from the underlying filesystem and returning the results. We will describe in detail how CRFS handles various filesystem calls in the following sections.

A. File Open

At the beginning of a checkpoint, each application process calls *open()* to create a new checkpoint file. This system call is caught by FUSE kernel module and redirected to CRFS. CRFS maintains a hash table to keep track of opened files. Each opened file is associated with an entry that contains metadata to be used in later I/O operations. A new entry is inserted into the table for a newly opened file. If the file is already opened, the reference counter in its table entry is incremented by one. After inserting the entry CRFS invokes the corresponding functions from the underlying filesystem to open/create the required file.

B. File Write

Concurrent file writing is a major performance bottleneck usually seen when checkpointing parallel applications [10, 11]. CRFS performs write aggregation to coalesce concurrent writes from parallel application processes into fewer larger chunks, and asynchronously write these bigger chunks to back-end storage system. With this, CRFS is able to improve file writing efficiency and reduce concurrent write contentions.

CRFS manages a buffer pool initialized at mount time. The buffer pool is divided into fixed-sized chunks. When an application writes data to CRFS, the *write()* system call is captured by FUSE and the control is handed to CRFS. CRFS goes to the buffer pool to grab a free buffer chunk, and associates this chunk to the file. The data to be written is copied into this chunk, the file's metadata entry is updated and the *write()* returns. The metadata entry includes information such as: size of valid data in the chunk, size of the chunk, append point in this chunk, offset of this chunk in the original file, and ownership identities. All subsequent writes to the target

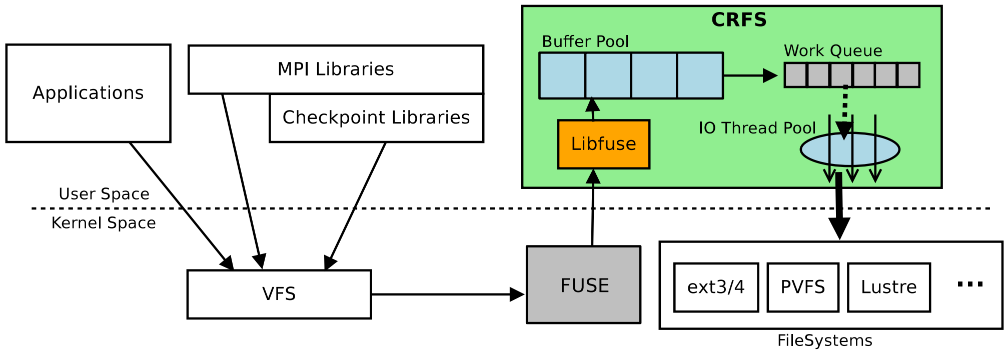


Fig. 4. CRFS Design Diagram

file will be coalesced into this chunk until the chunk becomes full. This is possible because checkpoint data is written sequentially during a checkpoint. After the chunk becomes full, it is en-queued into the Work Queue. At this point we increment the "write chunk count" by one in the metadata entry to mark the outstanding full chunk writes for this file. After that, the next free chunk is allocated to this file to accommodate the following writes.

Work Queue and IO Throttling: Data chunks are eventually handed over to the Work Queue for actual writing. CRFS manipulates a pool of worker IO threads waiting on the work queue. Whenever a chunk is enqueued, an IO thread wakes up and fetches the chunk off the queue. Each chunk is tagged with metadata information including target file handler, offset into the file, valid data size in the chunk, etc.. The IO thread then calls a *write()* with the underlying filesystem to write the data to its actual file. Once completed, the "complete chunk count" in the file's metadata entry is incremented. Then the chunk is returned to the buffer pool to be reused, and the IO thread goes back to the work queue for the next chunk.

CRFS can configure the IO thread number to throttle the outstanding buffer chunk write requests. With that, we can mitigate the IO contentions at back-end filesystems to attain a better performance. In Section V-B we carry out experiments to find a proper IO thread level.

C. File Close

When a process has finished checkpointing, a *close()* is called on its checkpoint file and is eventually routed to CRFS. If there is any remaining data in the buffer chunk associated with this file it's enqueued into the Work Queue. Then the calling thread is blocked until the "complete chunk count" becomes equivalent to "write chunk count", which means all outstanding buffer chunk writes have been finished. After that the call returns.

D. Other Filesystem Operations

1) File Read: File reading is required when a process is to be restarted from a checkpoint file. For *read()* we directly pass it to the underlying filesystem without any additional operation.

2) File Sync: The *fsync()* system call flushes a file's modified in-core data to the storage device where the file resides, which includes data coalesced in a file's buffer chunk, and data in page cache for the underlying filesystems. Upon a *fsync()*, we first enqueue the current buffer chunk associated with the file, then wait for all outstanding chunk writes to complete. After that a *fsync()* is called on the underlying filesystem to flush all dirty data in page cache to stable storage.

3) Other File Operations: For a user-level filesystem to be usable, many other filesystem operations are required to be supported, such as *mkdir*, *rmdir*, *link*, *truncate*, *chmod*, *utime*, and so on. CRFS directly passes these calls to the underlying filesystem without additional processing.

V. Performance Evaluation

We have implemented CRFS as a filesystem with special optimizations for generic checkpointing that can be used by a wide range of MPI stacks. In this section we conduct extensive experiments to evaluate its performance from various perspectives including: (a) Raw performance of CRFS to aggregate concurrent write streams; (b) Checkpoint writing performance of CRFS with different MPI stacks (MVAPICH2 [1], MPICH2 [3], OpenMPI [2]) using different back-end filesystems (ext3, Lustre [13], NFS). We will also show some detailed profiling to reveal the reasons why CRFS leads to benefits.

A. Experimental Setup

In the evaluation, a 64-node InfiniBand Linux cluster was used. Each node has eight processor cores on two

Intel Xeon 2.33 GHz Quad-core CPUs. Each node has 6 GB main memory and a 250GB ST3250620NS disk drive. The nodes are connected with Mellanox MT25208 DDR InfiniBand HCAs for high performance MPI communication. The nodes are also connected with a 1 GigE network for interactive logging and maintenance purposes. Each node runs Linux 2.6.30 with FUSE library 2.8.1. We enable the "big_writes" option for FUSE to perform large writes to deliver full performance. CRFS is mounted upon different filesystems: ext3, Lustre 1.8.3, and NFS at different runs. Lustre 1.8.3 is configured using 1 Metadata server and 3 Object Storage Servers with InfiniBand transport. For the case with NFS, the single NFS server exposes its disk via NFSv3 protocol using IPoIB transport.

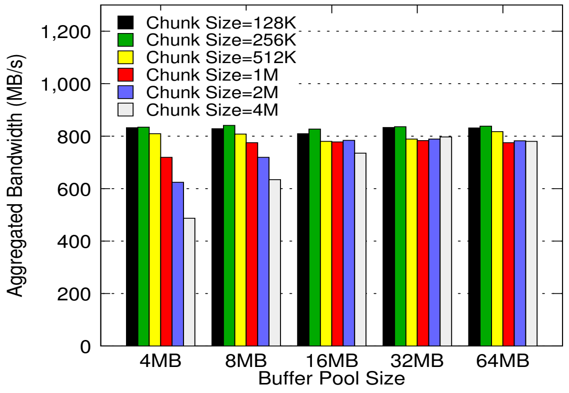


Fig. 5. CRFS Raw Write Bandwidth (8 processes on a single node)

B. CRFS: Raw Performance

CRFS relies on FUSE to intercept filesystem operations. The write requests are coalesced into CRFS's buffer pool, then the filled data chunks are handed to the work queue and processed by the IO threads asynchronously. Multiple factors can affect the overall performance of this pipeline, including FUSE internal overhead, buffer pool size, chunk size and IO thread numbers. In this test we ran 8 parallel processes in a node each writing 1 GB data into CRFS. Once a filled chunk is picked up by an IO thread it is discarded without being written to a back-end filesystem. With this we can measure the raw performance of CRFS to aggregate write streams, precluding the impacts of different back-end filesystems. In real checkpoint writing we observe that too many IO threads tend to generate high level of contentions when they concurrently write chunks to backend filesystems, leading to degraded performance, while too few IO threads cannot unleash the full potentials of the filesystem. After extensive experimental runs we find that 4 IO threads generally yield the best throughput for most of the situations. Due to space constraints we haven't included the detailed studies in the paper. We stick to 4 IO threads to throttle a high degree of concurrent IO in all experiments hereafter.

Benchmark	MPI Library	Total Checkpoint Size (MB)	Process Image Size (MB)
LU.B.128	MVAPICH2-IB	903.2	7.1
	OpenMPI-IB	909.1	7.1
	MPICH2-TCP	497.8	3.9
LU.C.128	MVAPICH2-IB	1,928.7	15.1
	OpenMPI-IB	1,751.7	13.7
	MPICH2-TCP	1,359.6	10.7
LU.D.128	MVAPICH2-IB	13,653.9	106.7
	OpenMPI-IB	13,864.9	108.3
	MPICH2-TCP	13,261.2	103.6

TABLE II
CHECKPOINT SIZES OF DIFFERENT APPLICATIONS WITH VARIED MPI STACKS. MVAPICH2 AND OPENMPI USE INFINIBAND TRANSPORT. MPICH2 USES TCP TRANSPORT.

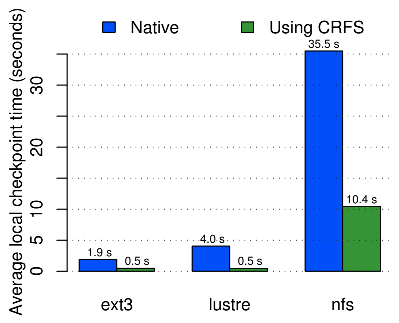
Figure 5 shows the measured write throughput at different buffer pool sizes with varied chunk sizes. Given a 16 MB buffer pool, CRFS can always achieve more than 700 MB/s aggregation throughput on a single node for chunk sizes larger than 128 KB. This is sufficient to saturate most available parallel filesystems if multiple nodes concurrently write their checkpoints. Larger chunk size is generally more favorable for the underlying filesystems to exhibit full potentials, therefore we would fix chunk size to be 4 MB in all experiments hereafter.

For a given chunk size=4 MB, it can be observed that write throughput rises as buffer pool becomes bigger and starts to flatten when buffer pool is greater than 32 MB. CRFS shouldn't occupy too much memory since a real parallel application can use a large portion of the available memory. Hence we fix the buffer pool to be 16 MB in all experiments hereafter.

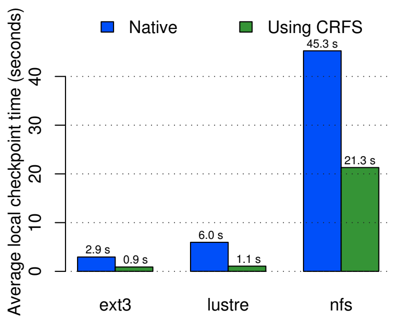
C. CRFS: Checkpointing Performance

In this section, we evaluate the performance of CRFS to checkpoint real applications. We ran NAS parallel benchmark LU of class B, C and D with 128 processes on 16 compute nodes. We chose ext3, NFS and Lustre-1.8.3 as the three underlying filesystems. The checkpoint was either directly written to native filesystems, or processed by CRFS before writing. In order to demonstrate the portability of CRFS that can benefit a wide range of applications, the experiments were repeated using three popular MPI implementations that support Checkpoint/Restart: MVAPICH2 1.6rc3, OpenMPI 1.5.1 and MPICH2 1.3.2p1.

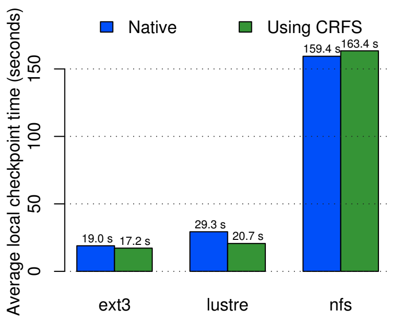
Table II shows the checkpoint sizes at varied application scales. The three MPI stacks are tagged with "IB" (InfiniBand) or "TCP" to indicate the transport



(a) LU.B.128

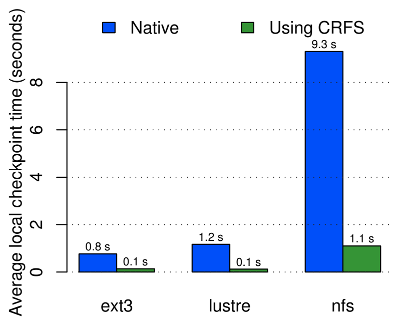


(b) LU.C.128

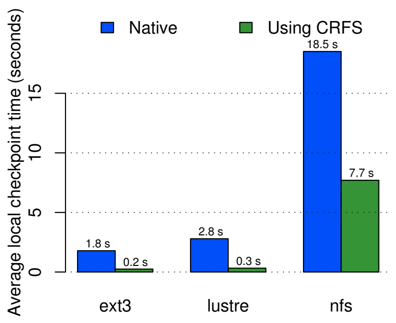


(c) LU.D.128

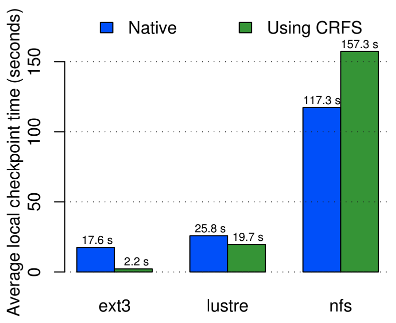
Fig. 6. Checkpoint Writing Time with MVAPICH2 (Lower is Better)



(a) LU.B.128

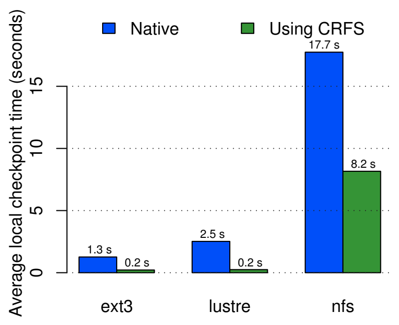


(b) LU.C.128

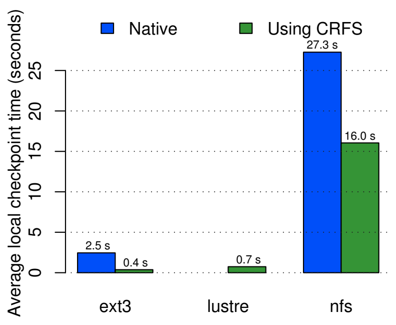


(c) LU.D.128

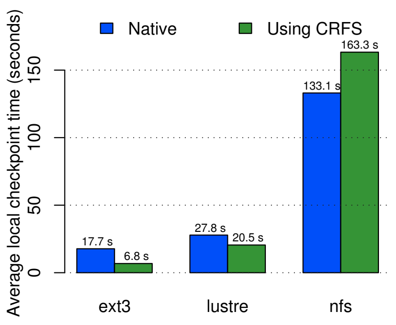
Fig. 7. Checkpoint Writing Time with MPICH2 (Lower is Better)



(a) LU.B.128



(b) LU.C.128



(c) LU.D.128

Fig. 8. Checkpoint Writing Time with OpenMPI (Lower is Better). *We could not get the result of native Lustre for LU.C.128, despite many attempts.*

they use. Generally MVAPICH2 and OpenMPI produce checkpoint images slightly bigger than MPICH2. This is because they use InfiniBand transport which requires more memory to maintain the communication channels. MPICH2, on the other hand, uses TCP transport and has a lower memory footprint.

The measured checkpoint time includes the time for BLCR to write the checkpointed data and the time to close the file (so there is no pending data in CRFS) for all the processes. The values plotted in the following figures are the average checkpoint time among all the processes for one given checkpoint, and the average for at least 5 checkpoints in the same conditions.

Figure 6 gives the checkpoint writing time for MVAPICH2. It clearly indicates that CRFS is able to diminish checkpoint writing overhead for different underlying filesystems at a wide range of application memory footprint. For example, for application class C, CRFS with Lustre reduces the writing time from 6.0 seconds to 1.1 seconds, which stands for a 5.5X speedup. With ext3 and NFS filesystems the improvements are 3.2X and 2.1X, respectively. For bigger problem size of class D the improvement is less dramatic because the majority of overhead is dominated by the absolute amount of data to dump. Even in this case CRFS is 30% faster than native Lustre and drives down the writing time from 29.3 seconds to 20.7 seconds. We observe that NFS becomes an outlier at this problem size. NFS isn't a good candidate to store checkpoint since its single server design doesn't match the intensive concurrent IO requirements. The node-level optimizations carried out by CRFS cannot benefit NFS server to handle the high level of concurrent IO tension, and the additional overhead within CRFS (such as multiple buffer copies) starts to manifest. Therefore CRFS+NFS performs slightly worse than the native NFS.

Figure 7 and Figure 8 exhibit the performance benefits gained by CRFS when running with MPICH2 and OpenMPI, respectively. A similar level of improvement is obtained here. In the case of MPICH2, for example, with application class C and Lustre filesystem, CRFS achieves a 9.3X speedup to complete checkpoint writing. The speedup is 2.4X with NFS for the same problem size. We see the same abnormality with NFS when problem size becomes larger at class D. At this problem size, CRFS optimizations cannot make a positive impact, instead its overhead appears more prominent for NFS.

In Figure 8(b), the bar for LU.C.128 with OpenMPI using native Lustre is missing. Despite several tries, the checkpoint in OpenMPI always failed for these conditions.

D. CRFS: Multiplexing Scalability

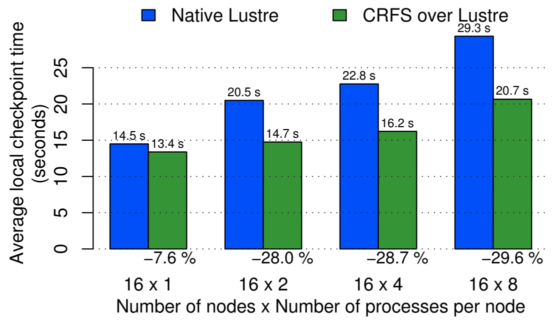


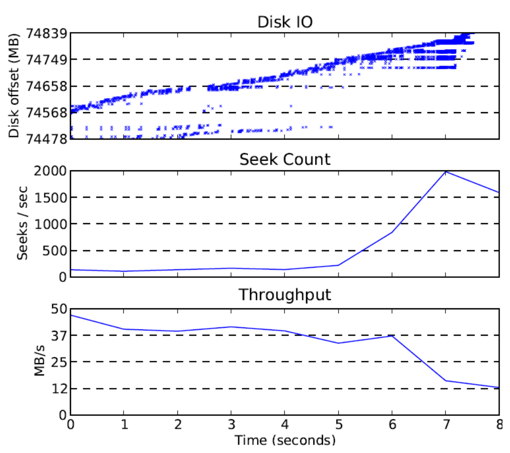
Fig. 9. CRFS Scalability at Different Level of Process Multiplexing (LU.D, Lustre)

As revealed in previous sections, IO contentions between concurrent processes are the major cause of the pathologically poor performance of checkpoint writing. In this section we vary the number of processes on each compute node for the same problem size, and evaluate CRFS's scalability in terms of process multiplexing to handle checkpoint IO. We run the same application LU.D on 16 compute nodes with 16, 32, 64, and 128 processes, leaving 1, 2, 4 and 8 processes per node, respectively. We measure the checkpoint writing time to Lustre filesystem, either natively or with CRFS optimizations. Figure 9 depicts the results run with MVAPICH2. Similar results are obtained with OpenMPI and MPICH2. The numbers atop each bar is the checkpoint writing time, and the numbers below the bar groups are percentage reduction in writing time when CRFS is applied. With 16 application processes (1 process per node), CRFS doesn't give significant benefit since there is little IO concurrency per node. As process multiplexing rises, CRFS starts to show its advantages. When 32 processes are used (2 processes per node), CRFS reduces the checkpoint writing time by 28%. For 64 and 128 processes (4,8 processes per node), the overhead reduction is 28.7% and 29.7%, respectively. This result indicates that CRFS can effectively reduce the node-level IO multiplexing contention and diminish checkpoint writing overhead.

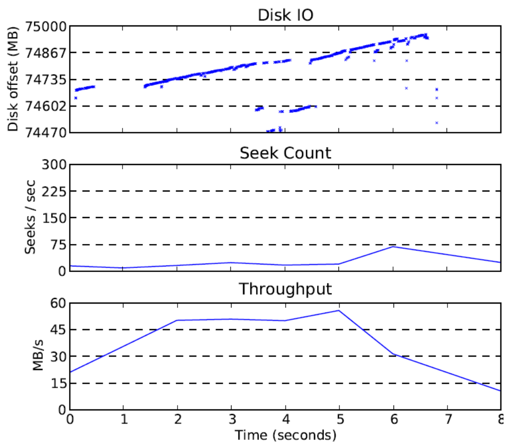
E. CRFS: Reasons of Improvements

The aforementioned experiments have demonstrated CRFS's capability to improve checkpoint writing efficiency. We have also further explored the reasons why CRFS can bring about such benefits. We used "blktrace" to collect the block IO layer access traces during checkpoint writing to local ext3. We ran application LU.C.64 with 64 processes on 8 compute nodes using MVAPICH2. The results collected from one node are illustrated in Figure 10. The top part of Figure 10(a) shows the disk IO pattern of checkpoint writing. We see a high degree of randomness caused by concurrent writing from 8 processes on the same node. This enforces a lot of disk head seeks (the middle part of Figure 10(a)), and results in a lower effective write throughput. In contrast, CRFS is able to coalesce the concurrent write requests and perform relatively sequential writes, as can be seen in top part of Figure 10(b). Consequently it can avoid a lot of disk head seeks and deliver a better write throughput.

By merging concurrent write requests and reducing the level of IO contention, CRFS brings about another benefit to diminish the uncertainty of checkpoint writing completion time. Figure 11 compares the cumulative checkpoint write time for all the processes. We see a wide variation in the completion time if writing directly to ext3. In this mode, the slow writing processes hinder the overall progress of all processes in the application, resulting in a prolonged checkpoint completion time. On the contrary, CRFS effectively drives down the write contentions at back-end filesystem so as to significantly minimize this variation. As a consequence, all processes converge and finish their writing at about the same time, as shown in Figure 11. This helps achieve a



(a) Write to ext3



(b) Write to ext3+CRFS

Fig. 10. Block IO Layer Trace on One Node. Checkpoint Writing of LU.C.64 on 8 Nodes, ext3

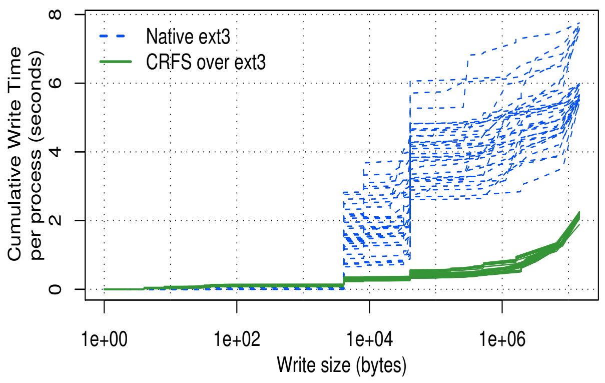


Fig. 11. Cumulative Write Time for Each Process (LU.C.64, ext3 vs. ext3+CRFS)

quicker resumption of the application execution after a checkpoint is carried out.

F. Restart

During restart, BLCR library reads from checkpoint files and restores the in-memory context for every process. CRFS forwards every read request to the back-end filesystem, and does not impose any additional overhead on file reads. What's more, CRFS doesn't change any file layout during checkpoint write phase. Consequently an application can be restarted directly from the back-end filesystem, without the need to mount CRFS. In our experiments, we did not observe any noticeable improvement in the application restart time when CRFS is mounted atop an underlying filesystem. Given that, and the lack of space, we did not include any numbers characterizing the restart performance.

VI. Related work

In the field of High Performance Computing systems, many efforts have been carried out to provide fault tolerance to MPI applications. Generally, the application state is periodically saved and used to restart the application when a failure occurs and the checkpoint is coordinated among the processors [9]. CoCheck [17], Starfish [18], LAM/MPI [19], among others, implement this class of checkpointing. These coordinated checkpoint approaches share a downside in that all processes must save their process images in a coordinated manner, which imposes a heavy burden on the IO subsystem.

The overhead of Checkpoint/Restart on file IO has been extensively studied by many researchers [10, 20]. A lot of efforts have been conducted to tackle the IO bottleneck incurred by C/R. The work in [4, 12] modified the MPI implementation and BLCR library to alleviate the IO contention. Although effective, this approach only works for a specific MPI stack and requires patching BLCR kernel module, which isn't portable to be applied to generic environments. The authors of [11] proposed a parallel log-structured filesystem (PLFS) to improve the writing throughput. However, this solution only deals with N-1 scenario where multiple processes write to the same shared file, hence it cannot handle MPI system-level checkpoint where each process is checkpointed to a separate image file. The authors in [21] modified the PVFS [22] filesystem to serialize all file writing requests for checkpointing. This approach isn't portable because it requires changing the filesystem, and impedes data reading throughput because additional remapping is needed for every read request to find its data. Stdchk [23] tries to scavenge spare storage resources from all participating nodes to form a dedicated storage space for checkpoint data. Our work differs in that we implement a user-level filesystem that can coalesce concurrent write accesses for better IO performance. The study in [24] proposes a CLL algorithm to reduce checkpoint overhead. It's a user-level optimization, and its buffer management incurs significant overhead to synchronize the copier thread and application thread on every common page access. On the contrary, our work doesn't involve such overhead due to the filesystem based approach.

VII. Conclusions and Future Work

In this paper we have conducted extensive profiling to identify the dominant factors that determine the cost of Checkpoint-Restart. Based on the findings, we have designed and implemented CRFS, a user-space filesystem, which optimizes concurrent checkpoint writing with the principle of write aggregation. The generic filesystem-based architecture enables a wide range of software components, including any MPI stack and general IO application, to transparently benefit from CRFS's optimizations. We have conducted comprehensive evaluations of the proposed design and demonstrated a significant improvement in checkpoint writing performance with three popular MPI stacks: MVAPICH2, MPICH2 and OpenMPI. Significant improvements are achieved in all the three MPI stacks to reduce checkpoint writing overhead. Experimental results show that the checkpoint time with Lustre is reduced by 29% for LU class D. Up to 8X speedup is obtained if CRFS is used with ext3.

As part of our future work, we plan to explore how CRFS can optimize inter-node concurrent IO writing to further reduce the IO contentions. We will also investigate other general IO applications that will transparently benefit from CRFS.

VIII. Funding Acknowledge

This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CCF-0621484, #CCF-0833169, #CCF-0916302, #OCI-0926691 and #CCF-0937842; grant from Wright Center for Innovation #WCI04-010-OSU-0; grants from Intel, Mellanox, Cisco, QLogic, and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Advanced Clustering, Appro, QLogic, and Sun Microsystems.

IX. Software Distribution

The proposed design will be available as a standalone filesystem that will be distributed with an upcoming MVAPICH2 release.

References

[1] "MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE," http://mvapich.cse.ohio-state.edu/.

[2] "Open MPI: Open Source High Performance Computing," http://mvapich.cse.ohio-state.edu/.

[3] "MPICH2: High-Performance and Widely Portable MPI," http://www.mcs.anl.gov/research/projects/mpich2/.

[4] X. Ouyang, K. Gopalakrishnan, and D. K. Panda, "Accelerating Checkpoint Operation by Node-Level Write Aggregation on Multicore Systems," *ICPP 2009*, September 2009.

[5] J. Hursey, J. Squyres, T. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for open mpi," in *12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems*, March 2007.

[6] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[7] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters," in *SciDAC*, 6 2006.

[8] Q. Gao, W. Yu, W. Huang and D. K. Panda, "Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand," in *International Conference on Parallel Processing (ICPP)*, August 2006.

[9] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.

[10] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley, "Memory exclusion: Optimizing the performance of checkpointing systems," in *Software: Practice and Experience*, 1999.

[11] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: a checkpoint filesystem for parallel applications," in *Proc. of SC '09*, 2009.

[12] X. Ouyang, K. Gopalakrishnan, T. Gangadharappa, and D. K. Panda, "Fast Checkpointing by Write Aggregation with Dynamic Buffer and Interleaving on Multicore Architecture," *HiPC 2009*, December 2009.

[13] "Lustre Parallel Filesystem," http://wiki.lustre.org/.

[14] "Filesystem in Userspace," http://fuse.sourceforge.net/.

[15] "A flow-chart diagram which shows how FUSE works," http://en.wikipedia.org/wiki/Filesystem_in_Userspace.

[16] F. C. Wong and R. P. M. etc., "Architectural requirements and scalability of the NAS parallel benchmarks," in *Proc. of Supercomputing '99*, 1999, p. 41.

[17] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," in *Proc. of the 10th International Parallel Processing Symposium (IPPS '96)*, 1996.

[18] A. Agbaria and R. Friedman, "Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations," *High-Performance Distributed Computing, International Symposium on*, vol. 0, p. 31, 1999.

[19] S. Sankaran and J. M. Squyres and B. Barrett etc, "The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing," *LACSI*, Oct. 2003.

[20] I.R. Philp, "Software failures and the road to a petaflop machine," in *First Workshop on High Performance Computing Reliability Issues (HPCRI)*, February 2005.

[21] Milo Polte and Jiri Simsa etc. , "Fast log-based concurrent writing of checkpoints ," in *PDSI 2008 workshop in conjunction with SC08* , Nov. 2008.

[22] "PVFS2," http://www.pvfs.org/.

[23] S. Al-Kiswany, M. Ripeanu, S. Vazhkudai, and A. Gharaibeh, "stdchk: A Checkpoint Storage System for Desktop Grid Computing," in *ICDCS 2008.*, June 2008.

[24] K. Li, J. F. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Trans. Parallel Distrib. Syst.*, 1994.