Towards Scalable Application Checkpointing with Parallel File System Delegation

Dulcardo Arteaga Ming Zhao
School of Computing and Information Science
Florida International University, Miami, USA
{darte003,ming}@cis.fiu.edu

***Abstract*—The ever-increasing scale of modern high-performance computing (HPC) systems presents a variety of challenges to the parallel file system (PFS) based storage in these systems. The scalability of application checkpointing is a particularly important challenge because it is critical to the reliability of computing and it often dominates the I/Os in a HPC system. When a large number of parallel processes simultaneously perform checkpointing, the PFS metadata servers can become a serious bottleneck due to the large volume of concurrent metadata operations. This paper specifically addresses this PFS metadata management issue in order to support scalable application checkpointing in large HPC systems. It proposes a new technique named PFS-delegation which delegates the management of the PFS storage space used for checkpointing to applications, thereby relieving the load of metadata operations on the PFS during their checkpointing. This proposed technique is prototyped on PVFS2, a widely used PFS implementation, and evaluated on a HPC cluster using a representative parallel I/O benchmark, IOR. Experiments with up to 128 parallel processes show that the PFS-delegation based checkpointing is significantly faster than the traditional shared-file and file-per-process based checkpointing methods (7% and 10% speedup when the underlying PVFS2 uses a centralized metadata server; 22% and 31% speedup when using distributed metadata servers). The results also demonstrate that the PFS-delegation based checkpointing substantially reduces the total number of metadata operations handled by the metadata servers during the checkpointing.**

Keywords: delegation; checkpointing; metadata management; high performance computin; paralell filesystems and IO.

I. INTRODUCTION

High-performance computing (HPC) systems are important platforms for solving challenging problems in many disciplines. Such systems typically use parallel file systems (PFSs) to perform I/Os in parallel across storage devices and provide high-throughput to the applications. As the scale of modern HPC systems continue to grow and as the applications in these systems become increasingly data intensive, a variety of challenges arise to the PFS-based storage in HPC systems. One particularly important challenge is the scalability issue of application-initiated checkpointing. HPC applications often use checkpoints to record their execution state persistently so that when failures happen they can resume the computing from their previous checkpoints without losing all the progress.

Efficient checkpointing is critical to both the reliability and performance of an HPC application. However, when a large number of parallel processes simultaneously perform checkpointing, the PFS metadata servers can become a serious bottleneck due to the large volume of concurrent metadata operations. Traditionally, an application's parallel processes use either a *shared file* or a *file per process* to store the checkpointing data, both of which can incur substantial overhead in metadata management. The shared-file method can involve a large number of operations on the shared file's attributes and locks, whereas the file-per-process method also requires a large number of file creations. As HPC applications and systems continue to grow in size, such metadata management overhead is becoming an increasingly serious issue to the scalability of application checkpointing.

This paper focuses on the aforementioned PFS metadata management issue for large-scale application checkpointing and proposes a new technique named *PFS-delegation* to address it. This technique delegates the management of the PFS storage space used for checkpointing to applications, thereby relieving the load of metadata operations on the PFS during their checkpointing. Specifically, an application can use PFS-delegation to reserve a chunk of the parallel storage space for checkpointing and it can then manage and access the checkpoints in its reserved space without involving the PFS metadata servers. In this way, the amount of PFS metadata operations incurred during the checkpointing is minimized *regardless of how many processes are involved* and *regardless of how many checkpoints are performed*.

The proposed PFS-delegation technique is prototyped on PVFS2, a widely used PFS implementation. It is evaluated in a cluster environment with four PVFS2 servers and up to 128 parallel processes using a representative parallel I/O benchmark, IOR (v2.10.2) [1]. The results show that the PFS-delegation based checkpointing is significantly faster than the traditional shared-file and file-per-process based checkpointing methods. When the underlying PVFS2 uses a centralized metadata server, the speedup is 7% versus shared-file and 10% versus file-per-process; when using distributed metadata servers, the speedup is 22% and 31% respectively. The results also demonstrate substantial

reductions on the number of metadata operations handled by the metadata servers during the checkpointing.

The rest of this paper is organized as follows. Section II introduces the background, Section III describes the overall architecture, Section IV presents the implementation details, Section V discusses the experimental evaluation, Section VI examines the related work, Section VII offers additional discussions, and Section VIII concludes the paper.

II. BACKGROUND

In a typical HPC system, data are managed and provisioned through a parallel file system (PFS), which supports high-performance parallel I/O for applications to access their data on the storage devices. The PFS provides the bridge between the computing infrastructure (compute nodes) and the storage infrastructure (storage networks and devices), which are typically connected through a high-speed communications network (e.g., Gigabit Ethernet, Infiniband, Myrinet).

A modern PFS (e.g., GPFS [2], PVFS [3], Lustre [4], IBRIX [5], and Panasas [6]) typically consists of clients, data servers, and metadata servers. In a HPC system, the PFS clients often run on the compute nodes and provide the interface to the storage system which is managed by the metadata and data servers. A metadata server stores the meta-information about files, including file naming, directory hierarchy, data distribution, access permissions, and file locking. The data of files are stored through data servers, which are connected to the storage devices through either direct links or a shared storage-area network (SAN). The data layout of a file specifies how the data is distributed on a list of servers using algorithms such as round robin and random.

File accesses typically first go through the metadata server to obtain the appropriate access permission and the data layout on the data servers. A large read or write on the file is usually striped across multiple data servers to achieve high throughput via I/O parallelism. Because centralized metadata management can become a bottleneck for metadata access, some PFS also employ multiple metadata servers [2][4] or completely distribute the metadata management along with the data servers [5]. PFS clients often cache the retrieved metadata (and in some cases, data) locally to further reduce the overhead from metadata (and data) accesses.

Application-initiated checkpointing is a major source of I/O traffic in a HPC system, which is estimated to account for about 80% of the I/O usage in today's HPC systems [7]. To a storage system, checkpointing is often treated in the same way as other types of I/Os such as regular application computation inputs and outputs. It, however, has specific and unique I/O characteristics. First, checkpointing is mainly *large sequential* writes and the use of checkpoint data is also often sequential reads. Small, random reads and writes are rare in accessing checkpointing data. Second, checkpointing I/Os issued by different applications and different parallel processes of the same application are *highly independent*. There is typically no sharing of the checkpointing data. Third, checkpointing I/O is *highly bursty*. A parallel application typically synchronizes its checkpointing operation across all of its parallel processes. At the checkpointing time, a large volume of I/Os flow from the computing nodes to the storage infrastructure simultaneously.

Future large-scale HPC applications will employ hundreds of thousands to millions of processors which will generate a tremendous amount of concurrent accesses to checkpoints on a PFS. The challenge to scalability arises from this need of accessing a large number of checkpoints simultaneously from all the compute nodes. Traditionally, an HPC application's parallel processes checkpoint their data on the PFS either via a *shared file* (a.k.a., *N-1 access pattern*), where all processes write to the same shared file, or using a different *file per process* (a.k.a., *N-N access pattern)*, where each process writes to a different file [8]. In the case of N-1 access pattern, there exist two variations, *N-1 segmented*, where each process writes its data to a separate sequential region of the shared file, and *N-1 strided*, where all process write to the same set of regions of the shared file but each process writes a different part of these regions [13].

With the shared-file approach, a single file's metadata are shared among a large number of clients, which can become a bottleneck when accessed by a large number of checkpointing applications simultaneously. The file-per-process approach can eliminate this bottleneck; however, the creation and use of hundreds of thousands to millions of files, typically within the same directory, on the PFS introduces significant overhead in metadata management [9][10]. In practice, the N-1 strided access pattern is found more convenient by users and hence is more commonly used than the N-1 segmented access pattern and the N-N access pattern [13]. However, the N-1 strided access pattern can be much less efficient than the other patterns because it requires concurrent accesses to the same regions in the shared file which often have to be serialized on the underlying storage.

Typical PFSs are designed for general-purpose usage and cannot differentiate checkpointing I/Os from others. As a result, such a PFS is unable to recognize the unique I/O characteristics and needs of checkpointing in order to reduce unnecessary cost and improve its performance. Specifically, there is *no need to maintain synchronization* across different processes on the checkpoint data, since the data are not shared by the processes. It is unnecessary to use client-side caching as applications rarely immediately read back their checkpoints (it only happens during the recovery procedure after a compute node failure). Hence, there is also *no need to maintain consistency* between the client-side caches and server-side storage. On the other hand, optimization is necessary for the PFS to support the

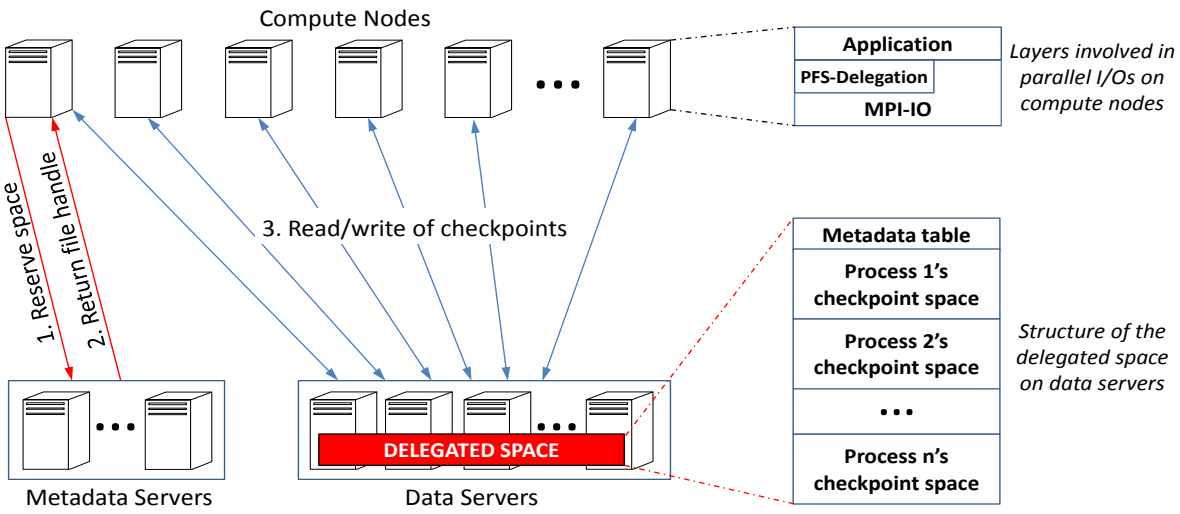

Figure 1. Architecture of PFS-delegation

challenging I/Os from simultaneous checkpointing by large numbers of parallel processes. In particular, the overhead of PFS metadata management needs to be improved, no matter whether the parallel checkpointing is done in a shared-file or file-per-process manner. This specific problem is addressed by the PFS-delegation technique proposed in this paper in order to support scalable checkpointing.

III. ARCHITECTURE

We propose *PFS-delegation* to offload the management of portions of the PFS storage space to applications to relieve metadata management bottleneck at the PFS (Figure 1). PFS-delegation pre-allocates a certain region of the parallel storage space to each parallel application for its processes to store checkpoints. This pre-allocated space is striped across the PFS data servers. To the PFS, it appears as merely a single logical file, whereas the management of checkpoints inside of this file, including naming and data layouts, is entirely delegated to the application with the support from PFS-delegation. In this way, the application's use of possibly large numbers of processes and checkpoints in its delegated space is completely hidden from the PFS, whereas the metadata management on the PFS is incurred for only a single logical file.

With PFS-delegation, an application partitions the delegated storage space across all the processes for them to access checkpoints in parallel. For writing a checkpoint, each process flushes out its data sequentially in its portion of the delegated space which is then striped across the involved data servers; for loading a checkpoint, each process also reads the data sequentially from the data servers in parallel. The size of the delegated storage space can be determined based on both the storage needs of the application and the allocation policy of the system. When an application uses up its allocated space, it will roll over to the beginning of the space for storing new checkpoints.

PFS-delegation can be conceivably implemented using two complementary approaches with different levels of transparency to the underlying PFS. In the first approach, PFS-delegation leverages the existing interface of the PFS to realize delegation and only requests the creation of the necessary logical file from the metadata servers. This approach can be made entirely transparent to the underlying PFS and thereby supports different PFS deployments without modifications. But its effectiveness may be limited by the PFS protocol's restrictions. For example, the PFS may not support efficient reservation of a large chunk of storage space. Alternatively, PFS-delegation can be also implemented by extending existing PFS protocols to provide additional API for checkpointing applications to directly request storage space allocation and delegation. In this paper, we focus on an implementation based on the first approach.

IV. IMPLEMENTATION

As a proof-of-concept, we have implemented PFS-delegation upon Parallel Virtual File System 2 (PVFS2) [2], a widely-used open-source PFS implementation, in order to support scalable application checkpointing. Specifically, the implementation of PFS-delegation entails two components: first, reserve the storage space to be delegated to an application on the involved data servers of the PFS; second, provide the application full read and write access to the delegated space on the PFS.

A. Reserving Delegated Storage Space

The reservation process is made by creating one large logical file across the PVFS2 data servers. The layout of this file determines which data servers will be involved to

provide the reserved storage space. The size of this file determines the size of the reserved space. This reservation process is executed only once per application, before the application starts checkpointing, using a command-line management utility *pfs-reserve*. The size of the reserved space should be determined by considering three different factors: the size of a checkpoint, the number of checkpoints to preserve, and the storage space allocation policy. Based on the understanding of an application's behavior, we could estimate how much space an application needs to reserve. But the actual reservation might be constrained by the storage space availability based on the understanding of the storage allocation policy.

The layout of the delegated space can be specified in the same way as defining a regular file's layout in MPI-IO. PFS-delegation uses the MPI-IO interface and the relevant hints to specify the data layout for the delegated space as follows.

```
MPI_Info info;
MPI_Info_create(&info);

/* number of servers to be used for reservation */
MPI_Info_set(info, "striping_factor", "4");

/* the striping unit in bytes */
MPI_Info_set(info, "striping_unit", "65536");

/* the logical file representing delegated space */
MPI_File_open(MPI_COMM_WORLD, "/pfs/checkpoint",
        MPI_MODE_CREATE | PI_MODE_RDWR, info, &fh);
```

A set of data servers is used to allocate the delegated PFS storage space, which is defined by the MPI_Info *striping_fator*. In the data layout we can also specify the stripe size by MPI_Info *striping_unit*. In the PVFS2-based implementation, the distribution of the logical file that represents the delegated space on the data servers is distributed using a simple-stripe scheme. It divides the logical file's data into stripes of 64KB which are mapped into the data files across the data servers in a round robin manner.

To make sure that the reserved space is available on the data servers, a naïve approach could simply populate the reserved space with blank data. However, this approach would incur substantial overhead when creating the reserved space, although it is only a one-time overhead. More efficient approaches are possible if the underlying PFS supports space reservation on the data servers without actually populating the data. Specifically, on PVFS2, because it is layered on top of the local file systems of the data servers, we leverage the support of sparse files of these file systems to implement efficient space reservation. To reserve a chunk of space on a PVFS2 data server we need to write only the last byte of the corresponding datafile. In this way we will have a datafile with the desired size without populating it with any data. PVFS2 would treat this file in the same way as other regular files whereas its internal data organization is managed by the PFS-delegation. By doing this on each data server, we will be able to create an empty logical file with the desired size to represent the reserved space.

However, we recognize that the use of sparse files does not really reserve the space from the file system and the storage can be out of space before a sparse file reaches its claimed size. To avoid such a problem the reservation can be done using the *fallocate* method which preallocates blocks to a file by marking them as unitialized and guarantees that the space is allocated to the file. On PFSs that support fallocate (e.g., GPFS [2]), this method can be directly used to reserve space on the parallel storage. For PVFS2, if the underlying local file systems (e.g., EXT4) support fallocate, then this method can also be used to reserve space on each data server individually. In this way, PFS-delegation can both support fast reservation and guarantee that the total reserved space will always be available for checkpointing.

After the delegated space is successfully reserved on the underlying PFS, it is partitioned internally based on the number of parallel processes of the application so that every single process has a portion in the delegated space to write the checkpointing data. The information about this partitioning together with the assignment to the processes is stored in a metadata table structure. Because this structure is critical to understand the data organization of the delegated space, it needs to be available to the application at any time. PFS-delegation uses a small portion of storage at the beginning of the delegated space to persistently store this information. In this way, if an application fails, in order to start the recovery process it needs to first read this metadata table to locate the checkpoints stored in the reserved space and then read them back. The structure of this metadata table is as follows.

```
struct  metadata_table
{
        PFS_offset      offset_start;
        PFS_offset      offset_end;
        PFS_offset      offset_next;
        int             revision;
};
```

The metadata information consists of four fields. The first two *offset_start* and *offset_end* refer to the offsets that define a portion of the reserved space that is dedicated to a specific client. These two fields are set when the delegated space is created. The third field *offset_next* corresponds to the offset where the next checkpoint should be written to. The last field *revision* indicates the current revision number. Note that in this metadata table example, we assume that each checkpoint from the same application is of the same size. To support checkpoints of different sizes, the metadata table needs to also track the offsets of individual

checkpoints stored in the application's delegated space. The metadata table can be further extended to track per-process data size so that we can also support the case where each parallel process of the application produces a different amount of data in the checkpoint, which happens when the application does incremental checkpointing.

B. Accessing Checkpoints in Delegated Space

PFS-delegation provides an interface for parallel applications to write/read checkpointing data to/from the delegated space. This interface is in addition to the typical interfaces such as MPI-IO [16] that an application uses to access regular files on a PFS. PFS-delegation itself also makes use of MPI-IO for creating and accessing the reserved space on a PFS. Therefore, it is transparent to the underlying PFS but not to the application which uses it for checkpointing. However, this interface is concise and easy to use and it would not affect how an application accesses other files outside of the delegated space. The provided functions in this interface are listed as follows.

```
size_t PFS_write_file (void *buffer, size_t count,
       struct options *opt);

size_t PFS_read_file (void *buffer, size_t count,
       struct options *opt);

size_t PFS_read_file_revision (void *buffer, size_t
       count, struct options *opt,int revision);
```

The above functions are all that an application needs in order to access the checkpoints in its delegated space. Specifically, the *PFS_write_file* function is used for performing the writes of a checkpoint on the delegated space; the *PFS_read_file* function is used for reading the last valid checkpoint from the delegated space; and the *PFS_read_file_revision* function is used for reading a specific past checkpoint stored in the delegated space.

Internally in PFS-delegation, the above functions are implemented as follows. To write a checkpoint in the delegated space, PFS-delegation needs to first read the metadata table and look for the offset where the application should write to. Then it uses the MPI-IO function to perform the writes and after the writing is completed PFS-delegation also needs to update the information in the metadata table accordingly, including the revision number and the offset. Note that only one process (specifically, the one with the MPI rank 0) of the application needs to perform the lookup and update of the metadata table so *access to the metadata table would not become a bottleneck*. If a failure occurs while performing a checkpoint operation, the offset would not be updated and the previous checkpoint would still be the latest valid checkpoint in the delegated space. On the other hand, to read from a checkpoint in the delegated space, PFS-delegation needs to first get the corresponding offset from the metadata table using a single process and then read the checkpoint data using MPI-IO in parallel.

In order to deal with inconsistencies between the metadata table and checkpoint data possibly caused by failures during writing, PFS-delegation can employ typical techniques such as checksum. Every time a checkpoint is executed, to guarantee that it is completed the metadata update is done at the end and a checksum is automatically calculated and saved together with the new metadata in the table. During the recovery, before reading the last checkpoint, the corresponding checksum is first read from the metadata table in order to validate that the data is not corrupted.

V. EVALUATION

A. Experiment Setup

The experiment testbed is built with a set of physical machines hosted on a cluster of eleven DELL PowerEdge 2970 servers. The cluster is connected by a Gigabit Ethernet and each node has two six-core 2.4GHz Opteron CPUs, 32GB of RAM, and one 500GB 7.2K RPM SAS disk. All physical machines run 2.6.24-16-server kernel in Ubuntu 8.0.4. PVFS2 is set up on these machines, four of which acting as PVFS2 data/metadata servers and the others as clients running parallel applications. The PVFS2 servers all use EXT3 as the local file system.

The experiments study the performance of the three checkpointing methods, *shared-file*, *file-per-process*, and *PFS-delegation*, in order to investigate whether the proposed PFS-delegation can improve large-scale application checkpointing. *Shared-file* based checkpointing (using N-1 segmented access pattern) saves the checkpoints of all the processes in a single shared file; *file-per-process* based checkpointing saves the checkpoint of each process into a separated file; *PFS-delegation* saves all the checkpoints from all processes in the delegated storage space on the PFS.

IOR (v2.10.2) [1] is chosen as the benchmark which uses MPI-IO to generate large sequential writes and simulate the typical checkpointing I/O patterns in HPC applications. IOR inherently supports checkpointing using shared-file with N-1 segmented pattern and file-per-process N-N pattern. As discussed in Section II, N-1 strided pattern is more commonly used by applications for checkpointing, although its performance is typically worse than the other patterns [13]. In our future work we will also evaluate PFS-delegation against N-1 strided, but it is reasonable to believe that PFS-delegation would outperform N-1 strided if it outperforms N-1 segmented.

The original IOR code was augmented in order to use the PFS-delegation interface to perform checkpointing. We use IOR to do nine consecutive checkpoints from all parallel processes, each process generating 20MB of data with a single write. A delay of two minutes is included

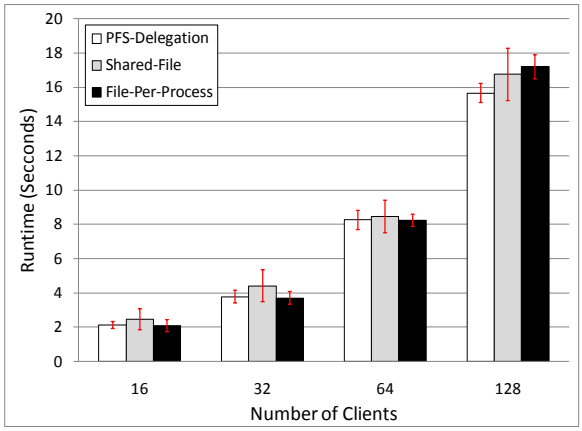

Figure 2. Checkpointing time with centralized metadata server

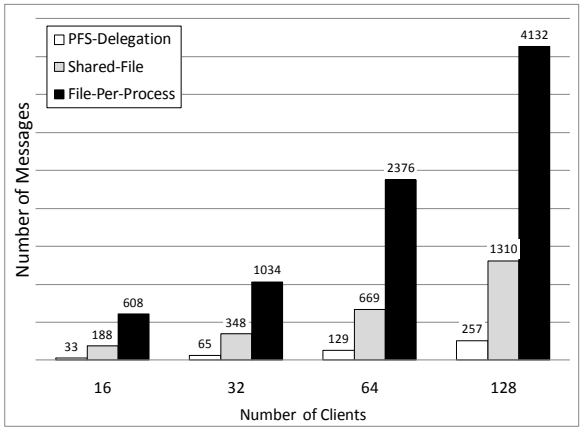

Figure 3. Total number of metadata operations with centralized metadata server

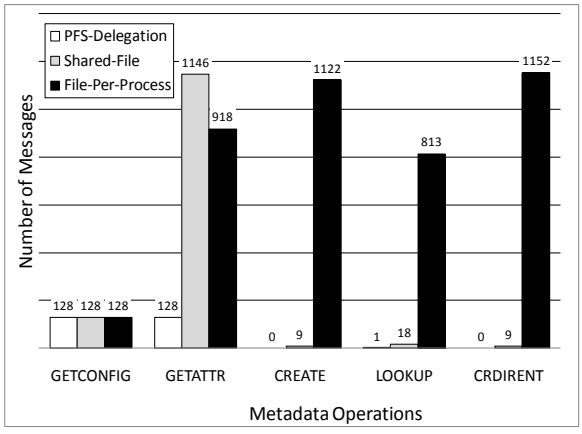

Figure 4. Different metadata operations with 128 processes and centralized metadata server

between two consecutive checkpoints in order for the writes to be completely flushed before the new checkpoint.

We considered two different typical setups of PFS metadata servers, *Centralized Metadata Server*, which uses one server dedicated as metadata server and three as data servers, and *Distributed Metadata Servers*, in which each of the four PFS servers act as both metadata server and data server. The number of parallel IOR processes scale from 16 to 128 for both configurations.

B. *Centralized Metadata Server*

Figure 2 shows the checkpointing time for the centralized metadata server setup using the three different checkpointing methods. The data reported are the average times and standard deviations across all IOR parallel processes and across nine consecutive checkpoints. The results show that when the number of processes is small (less than 64), the performance of the different checkpointing methods is similar. However, when the number of checkpointing parallel processes reaches 128, PFS-delegation based checkpointing is evidently faster than both the other two methods (7% faster than shared-file and 10% faster than file-per-process).

To understand the advantage of PFS-delegation based checkpointing, we also measure the number of metadata operations involved during the checkpointing by tracking the relevant PVFS2 messages on the metadata servers. Specifically, these PVFS2 messages include *GETATTR*, which is used to get file attributes; *CREATE*, which is used for file creation; *LOOKUP*, which is used for looking up the directory entry of a file; and *CRDIRENT*, which is used to create the directory entry for a new file.

Figure 3 shows the total number of metadata messages captured during a nine-checkpoints run by 16 to 128 parallel processes. The results show that PFS-delegation based checkpointing can substantially reduce the volume of metadata operations. The total number of metadata messages for PFS delegation is always only 20% of that for share-file and 30% of file-per-process, regardless of the number of parallel processes. The absolute difference is even more drastic when the number of processes is high. With 128 checkpointing processes, the number of metadata operations is reduced by 1053 messages when compared to shared-file and 3875 messages when compared to file-per-process.

Noticed that the advantage of PFS-delegation in terms of runtime is not as significant as in terms of the number of metadata operations, which is because of two factors. First, the checkpointing runtime is dominated by data operations rather than metadata operation. Consequently, the runtime is not affected much even though the number of metadata operations is substantially reduced by using PFS-delegation. Second, the metadata server's CPU utilization is low during the experiments, which is in fact around 1% most of the time. Therefore, even though shared-file and file-per-process require much more metadata operations, the metadata sever is able to handle them in time without affecting the application's runtime.

To further examine the reduction of metadata operations in PFS-delegation, we break down the numbers of different

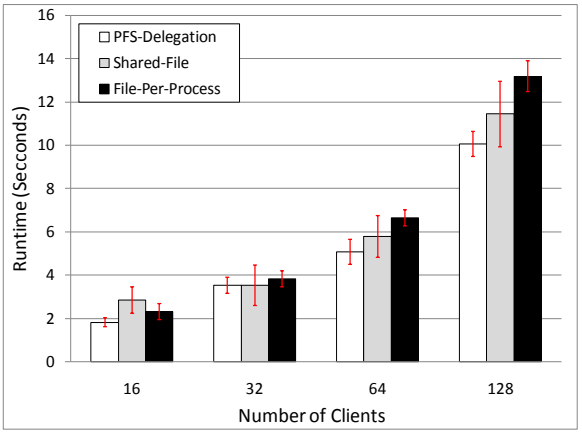

Figure 5. Checkpointing time with distributed metadata servers

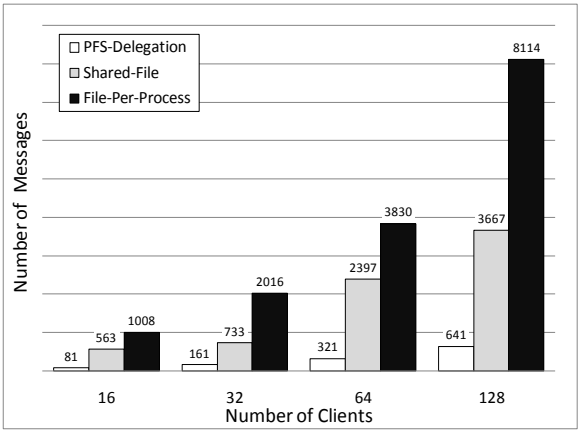

Figure 6. Total number of metadata operations with distributed metadata servers

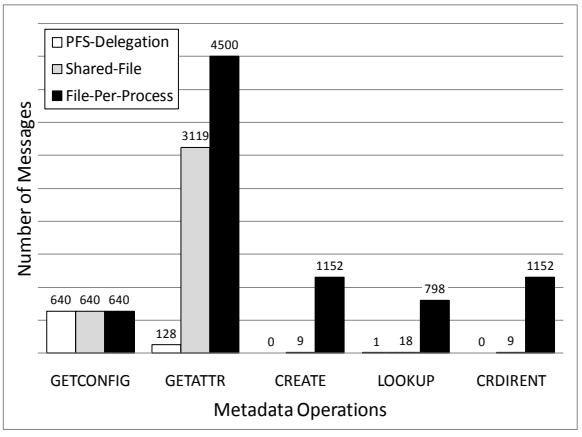

Figure 7. Different metadata operations with 128 processes and distributed metadata servers

relevant PVFS2 messages in Figure 4 when the number of checkpointing processes is 128. Here we can see that the number of *GETATTR* operations used in PFS-delegation is much less than the other two methods because the attributes of the logical file representing the delegated space remain the same during all checkpointing operations. Shared-file, file-per-process and PFS-delegation perform a GETATTR operations each time that is going to perform create, write and read operations, these operations are necessary to get information about the parent directory or file where the writes are going to be performed, depending on the mode that is being used the number of such operations can vary by a significant amount. For the other metadata operations, *CREATE*, *LOOKUP*, and *CRDIRENT*, we can see that file-per-process involves much more of these operations than shared-file and PFS-delegation, because it has to deal with a large number of checkpoint files, while only a few files need to be created for shared- file and no file needs to be created for PFS-delegation (assuming the delegated space is already reserved on the PFS).

C. *Distributed Metadata Server*

Figure 5 shows the checkpointing time for the distributed metadata server setup with the three different checkpoint methods. In this setup, PFS-delegation starts to outperform shared-file and file-per-process with 64 concurrent checkpointing processes. Specifically, it is 14% faster than shared-file and 31% faster than file-per-process. When the number of checkpointing processes reaches 128, the improvement is even more significant, in which PFS-delegation outperforms shared-file by 22% and file-per-process by 31%.

Figure 6 shows the total number of metadata messages captured during nine-checkpoints run on all the four metadata servers. These numbers are much higher than the centralized metadata server setup because now metadata operations need to be performed on all four PVFS2 servers. Nonetheless, the total number of metadata operations in PFS-delegation is still only 20% of shared-file and 10% of file-per-process with different numbers of checkpointing processes. Figure 7 shows the numbers of different relevant PVFS2 messages when the number of checkpointing processes is 128. We can make a similar observation as in the centralized metadata server setup: the number of *GETATTR* operations in PFS-delegation is much less than in shared-file and file-per-process, while the numbers of *CREATE*, *LOOKUP*, and *CRDIRENT* operations are much less than file-per-process.

In our experiments the distributed metadata servers are co-located with the data servers, which may cause the metadata operations to be slowed down by the data operations. Therefore, the performance from using shared-file and file-per-process based checkpointing may be better if the metadata servers can be hosted on dedicated nodes, separately from the data servers. However, the number of metadata operations would still remain the same even with dedicated, distributed metadata servers. Therefore, it is reasonable to believe that PFS-delegation would still outperform shared-file and file-per-process in this case.

VI. RELATED WORK

Oldfield *et al.* also recognized the limitations of general-purpose PFSs and the need for application-specific optimizations on parallel file access. They proposed the idea of a light-weight file system (LWFS) which does not provide many traditional PFS services, such as naming, consistency, and directory structures [9][10][15]. Instead, the LWFS only provides clients with direct and secure access to the storage, and other high-level services needed by applications can be included as libraries. The use of LWFS to support scientific applications is specifically studied in the context of checkpointing, which enables applications to skip the unnecessary metadata management and create and dump process state to storage efficiently.

LWFS allows applications to implement additional features on their own or through libraries, but its deployment requires the replacement of existing PFS deployments in HPC systems. Because such systems have spent considerable investments in purchasing, deploying, and fine-tuning their PFS setups, it is difficult to test and adopt a completely new PFS on the existing infrastructure. In comparison, this paper proposes to extend contemporary PFS implementations to support efficient metadata management for large-scale checkpointing. The proposed PFS-delegation technique is either transparent or requiring minimal modifications to the existing PFSs. Therefore, this paper's approach is complementary to LWFS as they are suited for different usage scenarios.

Google file system (GFS) is a special file system designed for workloads that have many large, sequential appending-only writes. It also strips away many unnecessary traditional file system services and only provides clients with flat namespace and relaxed consistency, and without client-side caching [11]. Nonetheless, GFS is only applicable to a specific set of applications that have similar characteristics as Google search and it cannot satisfy the needs of a wide variety of HPC applications. The technique proposed in this paper is applicable to different applications by leveraging typical PFSs widely used in HPC systems.

Another closely related work is the Parallel Log Structured File System (PLFS) [13][14], which proposes to improve the performance of an application's N-1 strided checkpointing by mapping this access pattern to the N-N access pattern through an interposition layer between the application and PFS. However, in order to realize the mapping, PLFS requires creating a large number of directories and files to rearrange the data belonging to the application-perceived N-1 checkpoint file. These operations can result in many metadata accesses such as directory creation, file creation, and setting attributes, thereby causing substantial metadata management overhead. In comparison, on one hand, PFS-delegation does not need the N-1-to-N-N mapping because it uses N-1 segmented access pattern which does not have the performance problems that N-1 strided has [13]. On the other hand, PFS-delegation reduces the total number of metadata operations by using only a single file for an application's entire reserved space and to store all of its checkpointing data. Hence it is reasonable to expect PFS-delegation to have much less overhead in metadata management.

PLFS implements access pattern mapping transparently to applications by providing a MPI-IO driver called *ad_plfs* and supports the same MPI-IO interface that applications typically use for parallel I/Os. Alternatively, PLFS also allows applications to use it through the POSIX interface without modifications by using FUSE [30] to implement its mapping at user space. Although PFS-delegation currently requires applications to be modified to use special APIs for checkpointing, it is conceivable that it can also take advantage of these techniques to make its use completely transparent to the applications.

In addition, there are other I/O mapping techniques in the related work for improving application checkpointing performance on a PFS such as Lustre [12] and GPFS [2]. For example, a library can be used to redirect the checkpointing I/Os in a way that each client only communicates with a single server. Such a data mapping technique is complementary to the metadata management issue tackled by this paper and it can also be easily incorporated into the PFS-delegation implementation.

There is also related work on reducing the checkpointing overhead through various approaches. The checkpointing-to-memory approach [27] proposes to reduce the application execution time associated with checkpointing operations by saving the checkpoint data in the memory of a different node. The copy-on-write checkpointing algorithms [22][23] reduce the overhead by buffering the checkpointing data to a separate address space via virtual memory, allowing the application's execution to continue while the data is flushed to stable storage. Another approach to buffering the checkpointing data is using the storage on a fast overlay network [21][24], allowing applications to quickly move the performance-limiting I/Os off the compute nodes. Incremental checkpointing [19] intends to reduce the size of checkpoint data by saving only the memory that has been touched since the last checkpoint operation.

Finally, there are several checkpointing models proposed in the literature which try to define the optimal checkpointing interval based on various parameters. One of them is a first order model that defines the optimal checkpointing interval in terms of checkpointing overhead and mean time to interrupt [28]. A different model proposed more recently also considers failures occurred during checkpointing and recovery [26]. There are also models that include bandwidth and computation time as parameters to calculate the optimal checkpointing interval [23][25].

VII. DISCUSSIONS

This paper proposes a new approach to address metadata management overhead and support scalable checkpointing in large HPC systems. The main limitation of our current prototype implementation is that it requires applications to be modified to use the new checkpointing APIs, although this new interface is concise and convenient to use. In our future work we will provide PFS-delegation transparently to applications by implementing it as a new MPI-IO driver and presenting the unmodified MPI-IO interface. In this way, an application can perform their preferred N-1 or N-N checkpointing without any change, while PFS-delegation automatically maps the application's I/Os to its perceived checkpoint files to the I/Os to the single delegated space.

As discussed in Section VI, this method of achieving application transparency is the same as the related work on PLFS [13]. We will in fact investigate the possibility of implementing PFS-delegation upon the PLFS code base. However, PFS-delegation can have much lower metadata management because of its use of single delegated space for storing all checkpointing data instead of using many separate files. We also believe that our current experiment results would still hold for this new application-transparent PFS-delegation implementation, because it does not change the interactions with the underlying PFS metadata servers or the internal management of the delegated space.

Our current implementation also requires users to use a special utility for offline reading a specific checkpoint from the delegated space. Such a utility is necessary because it needs to interpret the metadata table, locate the offset of the checkpoint in the delegated space, and then retrieve the desired data. In our future work we will implement the internal structure of a delegated space using technologies such as netCDF [18] and HDF5 [18] which provide self-describing, machine-independent data formats. For example, we can represent the metadata table structure with a netCDF data model. In this way, users can conveniently access the checkpoint data offline using the widely used netCDF or HDF5 tools to automatically interpret data in the delegated space.

As discussed in Section II, the use of synchronization in PFSs can be a performance bottleneck for checkpointing which in fact does not involve any data sharing. PFS-delegation can also address this problem by eliminating the use of locking when accessing checkpoints in the delegated space. However, this improvement is not reflected in our PVFS2-based evaluation, because PVFS2 does not support locking at all, due to the exact same scalability concern. Nonetheless, locking is still widely used in other PFSs in order to support full application semantics. In our ongoing work, we are extending our PFS-delegation implementation to support such PFSs (e.g., Lustre [4]) and conducting a more comprehensive evaluation on the potential benefits of the PFS-delegation approach.

The PFS-delegation approach proposed in this paper is generally applicable to different applications. A special application worth mentioning is the Software Persistent Memory (SoftPM) [29], a lightweight facility proposed for HPC applications to conveniently manage persistent data. SoftPM presents a persistent memory interface to applications which allows them to allocate persistent memory in the same way as allocating volatile memory and to easily restore, browse, and interact with past versions of persistent memory state. Internally, SoftPM implements the persistent memory upon the underlying storage system, recognizing and leveraging its characteristics to realize persistency and optimize performance. When SoftPM uses parallel storage as the backend, it can make use of PFS-delegation for saving and loading persistent data and achieving good performance with large scale.

In our experiments we have resources to run only up to 128 clients and 4 servers at the same time, but we expect to see the same level of advantage from PFS-delegation when the number of clients and servers scale up proportionally in a larger HPC system. In our future work we will try to evaluate our solution in a real production HPC system that has much larger scale that our testbed.

VIII. CONCLUSIONS

This paper presents a new technique, PFS-delegation, for addressing the metadata management overhead in large-scale application checkpointing. It allows an application to reserve a chunk of the PFS storage space for storing checkpoints and then delegates the management of these checkpoints completely to the application. In this way, the overhead of metadata management perceived by the PFS for the application's checkpointing can be drastically reduced to the level of a single logical file, regardless how many processes are involved and how many checkpoints are performed. This technique is a step towards supporting scalable application checkpointing in large HPC systems, which is critical to both the reliability and performance of these applications.

The proposed PFS-delegation is implemented as a library upon MPI-IO. It can support different PFSs without changing their code or existing deployments. It requires only small modification of an application for it to use the PFS-delegation interface. A prototype of this technique is developed upon a widely used PFS, PVFS2, and evaluated with experiments using a typical HPC I/O benchmark, IOR. Results show that the PFS-delegation based checkpointing significantly outperforms the shared-file and file-per-process based checkpointing in terms of both the application runtime and the load of metadata operations.

ACKNOWLEDGMENT

This research is sponsored by National Science Foundation under grant CCF-0938045 and Department of Homeland Security under grant 2010-ST-062-000039. The

authors are also thankful to the anonymous reviewers for their useful comments. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] H. Shan and J. Shalf, "Using IOR to Analyze the I/O Performance of HPC platforms", in Cray Users Group Meeting, May 2007.

[2] F. Schmuck and R. Haskin, "GPFS: A Shared-disk File System for Large Computing Clusters", in Proc. of the USENIX Conference on File and Storage Technologies, 2002.

[3] Parallel Virtual File System, version 2, URL: http://www.pvfs.org/pvfs2

[4] "Lustre File System: High-Performance Storage Architecture and Scalable Cluster File System", Sun Microsystems White Paper, October 2008.

[5] IBRIX Fusion, URL: http://www.ibrix.com/productsoverview.

[6] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable Performance of the Panasas Parallel File System", in Proc. of the USENIX Conference on File and Storage Technologies, February 2008.

[7] K. Davis and F. Petrini, "Tutorial: Achieving Usability and Efficiency in Large-Scale Parallel Computing Systems", in Proc. European Conference on Parallel Computing, August 2004.

[8] J. Borrill, L. Oliker, J. Shalf, and H. Shan, "Investigation of Leading HPC I/O Performance Using a Scientific Application Derived Benchmark", in Proc. of the ACM/IEEE Conference on Supercomputing, 2007.

[9] R. A. Oldfield, "Investigating Lightweight Storage and Overlay Networks for Fault Tolerance", in Proc. of the High Availability and Performance Computing Workshop, October 2006.

[10] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth, "Modeling the Impact of Checkpoints on Next-Generation Systems", in Proc. Of Conference of Mass Storage System and Technologies, 2007.

[11] S. Ghemawat, H. Gobioff, and S. Leung, "The Google File System", in Proc. Symposium on Operating Systems Principles, October 2003.

[12] P. Dickens and J. Logan, "A high performance implementation of MPI-IO for a Lustre file System environment", in Concurrency Computation Practice and Experience journal, 2009.

[13] J. Bent, H. Chen, D. Gunter, G. Grider, S. Gutierrez, A. Manzanares, B. McClelland, D. Montoya, J. Nunez, A. Torrez, M. Wingate, G. Gibson, M. Polte, and P. Nowoczinski, "PLFS: A Checkpoint filesystem for Parallel Applications", in Proc. of Conference Supercomputing System, 2009.

[14] J. Bent, H. Chen, D. Gunter, G. Grider, S. Gutierrez, A. Manzanares, B. McClelland, D. Montoya, J. Nunez, A. Torrez, M. Wingate, G. Gibson, M. Polte, and P. Nowoczinski, "PLFS Update presentation. Presentation on HEC-FSIO", 2010.

[15] L. W. Author, R. A. Oldfield, and R. Riesen, "Lightweight I/O for Scientific Applications", in Proc. of Conference on cluster computing of IEEEE International, 2006.

[16] MPI-IO Library, URL: http://www.mcs.anl.gov/research/projects/mpich2/.

[17] netCDF (network Common Data Form), URL: http://www.unidata.ucar.edu/software/netcdf/.

[18] HDF5 (Hierarchical Data Format), URL: http://www.hdfgroup.org/HDF.

[19] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems", in Proc. of the 18th Annual International Conference on Supercomputing, New York, NY, 2004. ACM Press.

[20] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent check pointing", in Proc. of the 11th Symposium on Reliable Distributed Systems, Houston, TX, October 1992, IEEE Computer Society Press.

[21] A. Gavrilovska, K. Schwan, O. Nordstrom, and H. Seifu, "Network processors as building blocks in overlay networks", in Proc. of the 1th Symposium on High Performance Interconnects (HOTJ03), pages 83-88, August 2003.

[22] K. Li, J. S. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs", IEEE Transactions on Parallel and Distributed Systems, 5(8):874-879, August 1994.

[23] K. Pattabiraman, C. Vick, and A. Wood, "Modeling coordinated checkpointing for large-scale supercomputers", in Proc. of the 2005 International Conference on Dependable Systems and Networks (DSN'05) , pages 812-821, Washington, DC, 2005. IEEE Computer Society.

[24] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A software-based multicast/reduction network for scalable tools", in Proc. of SC2003: High Performance Networking and Computing, Pheonix, AZ, November 2003.

[25] R. Subramaniyan, R. S. Studham, and E. Grobelny, "Optimization of checkpointing-related VO for high-performance parallel and distributed computing", in Proc. of The International Conference on Parallel and Distributed Processing Techniques and Applications, pages 937-943, 2006.

[26] N. H. Vaidya, "Impact of checkpoint latency on overhead ratio of a checkpointing scheme", IEEE Transactions on Computers, 46(8):942-947, 1997.

[27] L. M. Silva and J. G. Silva, "An experimental study about diskless checkpointing", in Proc. of the 24th EUROMICRO Conference vasteras, Sweden, August 1998.

[28] J. W. Young, "A first order approximation to the optimum checkpoint interval", Communications of the ACM, 1974.

[29] The Software Persistent Memory Project (SoftPM), URL: http://dsrl.cs.fiu.edu/projects/softpm/start.

[30] FUSE: File System in User Space, URL: http://fuse.sourceforge.net/.

[31] GPFS library, URL: http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=%2Fcom.ibm.cluster.gpfs32.basicadm.doc%2Fbl1adm_preallc.html.