Enhancing Checkpoint Performance with Staging IO and SSD *

Xiangyong Ouyang Sonya Marcarelli
Dhabaleswar K. Panda
Department of Computer Science and Engineering
The Ohio State University
{ouyangx, smarcare, panda}@cse.ohio-state.edu

Abstract

With the ever-growing size of computer clusters and applications, system failures are becoming inevitable. Checkpointing, a strategy to ensure fault tolerance, has become imperative in such an environment. However existing mechanism of checkpoint writing to parallel file systems doesn't perform well with increasing job size.

Solid State Disk(SSD) is attracting more and more attention due to its technical merits such as good random access performance, low power consumption and shock resistance. However, how to apply SSDs into a parallel storage system to improve checkpoint writing still remains an open question.

In this paper we propose a new strategy to enhance checkpoint writing performance by aggregating checkpoint writing at client side, and utilizing staging IO on data servers. We also explore the potentials to substitute traditional hard disks with SSDs on data server to achieve better write bandwidth. Our strategy achieves up to 6.3 times higher write bandwidth than a popular parallel file system PVFS2 [6] with 8 client nodes and 4 data servers. In experiments with real applications using 64 application processes and 4 data servers, our strategy can accelerate checkpoint writing by up to 9.9 times compared to PVFS2.

1 Introduction

The trend in the High Performance Computing community over the past couple of years has been to use a large number of distributed processing elements, connected together using a high performance network interconnect. With the increasingly large scale clusters, a significant challenge is raised for a parallel storage system to quickly absorb huge amount of data from multiple data streams produced by large scale parallel applications.

As a computer cluster grows larger in its size, the Mean Time Between Failures (MTBF) for today's clusters has reduced from days to a couple of hours [17, 22]. As a result, it has become vital for such clusters to be equipped with fault tolerance capabilities. From an application's point of view, fault tolerance can be achieved by periodically saving its state to a persistent storage, so that in the event of a failure, the application can be restored to the most recent checkpoint[23, 16, 20].

Checkpointing generates simultaneous access to the storage system when multiple processes of a parallel job write their process images to a shared file system. The shared file system has to intermix these data streams and store them to backend storage. If not properly managed, the interference between many data streams at the data servers can easily result in pathologically poor write performance [12, 28, 19].

A lot of techniques have been proposed to improve the performance of concurrent write to shared storage media [13, 14, 15, 12]. However, we still observe very poor write bandwidth when checkpointing a parallel application to PVFS2 [6] parallel file system in our experiments. The under-utilization of storage write bandwidth at checkpoint writing can be ascribed to two reasons:

- The pure cost to write the data to storage.
- Multiple write streams interfere with each other at data servers, which causes variances in a data stream's writing completion time. This variance in turn delays the completion of writing for all parallel data streams as a whole.

Solid State Disk(SSD) is attracting more and more

*This research is supported in part by DOE grants DE-FC02-06ER25755 and DE-FC02-06ER25749, NSF Grants CNS-0403342 and CCF-0702675; grants from Intel, Sun MicroSystems, Cisco Systems, and Linux Networks; and equipment donations from Intel, AMD, Apple, IBM, Microway, PathScale, SilverStorm, Sun MicroSystems.

978-0-7695-4025-2/10 $26.00 © 2010 IEEE
DOI 10.1109/SNAPI.2010.10

attention due to its technical merits such as good random access performance, low power consumption and shock resistance. SSD provides higher read/write bandwidth than hard disks. Some research has been carried out to investigate the possibility of adopting SSD into general computing [18]. However, little work has been done to explore the possibility of applying SSD to a parallel storage system to improve checkpoint writing.

In order to improve checkpoint writing performance, we want to address several questions in this paper:

- What are the checkpoint writing patterns in a parallel storage system?
- What are the potentials to apply SSD into a parallel storage system to improve storage bandwidth?
- How to utilize the techniques such as Write Aggregation, Staging IO and SSD, to design a parallel storage to enhance checkpoint writing performance?

In this paper we propose a new strategy to use Write Aggregation [26, 27] and Staging IO [10] to effectively utilize storage system bandwidth to accelerate checkpoint writing. Our strategy aggregates multiple data streams from processes on a compute node into a single stream. This single data stream is then stripped through all data servers by Remote Direct Memory Access(RDMA). On data servers, data chunks are buffered in a small staging area. Meanwhile a set of IO threads asynchronously flush the data to persistent storage. This effectively overlaps the data receiving with data writing at data server side. SSDs are adopted on data servers to improve storage bandwidth. By aggregating multiple data streams into one stream at client side, and overlapping data receiving into staging area with data write at server side, our approach is able to improve the achieved write bandwidth by up to 6.3 times compared to PVFS2. In experiment with parallel applications using 64 processes, our approach can accelerate the checkpoint writing by upto 9.9 times when checkpointing a parallel application.

The rest of paper is organized as follows. In section 2, we describe the background of Checkpointing and Staging IO. In section 3, we analyze the profiling information collected for the NAS Parallel Benchmark to characterize checkpoint writing. In section 4, we present our detailed designs and discuss our design choices. In section 5, we conduct experiments to evaluate our designs and present results that indicate improvement. In section 6, we discuss the related work. Finally we provide our conclusion and state the direction of the research we intend to conduct in future.

2 Background

2.1 Application Level and System Level Checkpointing

There are two basic techniques for checkpointing. The first approach requires the application to write its own user data to a checkpoint. The advantages includes a smaller checkpoint size since not all data in an application need to be saved. But the downside is the complexity to develop and maintain a checkpoint component. The other approach is system-level checkpointing, where the checkpointing is implemented transparently to the application, usually through the kernel module such as Berkeley Lab Checkpoint/Restart(BLCR)[16]. Our work in this paper is based on system-level checkpointing.

2.2 InfiniBand

InfiniBand [3] is an open standard for next generation high speed interconnect. In addition to send/receive semantics, it also provides the memory semantics communication, called Remote Direct Memory Access(RDMA) for high performance interprocess communications. By directly accessing the contents of remote memory, RDMA can provide both high bandwidth and low latency. Both RDMA Read and RDMA Write are supported. Many parallel file systems like PVFS2 [6] and Lustre [4] have incorporated InfiniBand support to provide high-throughput low-latency communication channels. In this paper, we take the advantage of RDMA Read to improve the data transfer bandwidth between client nodes and data servers.

2.3 Checkpoint in MVAPICH2

MVAPICH2 is a MPI library with native support for InfiniBand and 10GigE/iWARP [5]. It supports application initiated and system initiated checkpointing [25, 24] using the BLCR Library for Checkpoint/Restart [21]. Checkpointing in MVAPICH2 involves the following 3 phases.

- Phase 1: Draining the communication channels of all pending messages and tearing down the communication endpoints on each process.

- Phase 2: Using the BLCR Library to independently request the checkpoint of every process that is part of the MPI job. The checkpoint is taken by BLCR in a blocking manner with the data being written to one file per process.
- Phase 3: Re-establishing the communication endpoints on every process.

The application continues its execution after the checkpoint is taken. Of the 3 phases, both Phase 1 and Phase 3 are relatively constant in terms of time cost for a given application size. Phase 2 usually exhibits large variations for different applications, and it constitutes the majority part of the time cost. In this paper, we will design a new strategy to improve write bandwidth in Phase 2 so as to accelerate a parallel checkpointing.

3 Characteristics of Checkpoint Writing to Parallel File system

In our previous work [26, 27], we have profiled checkpoint writing characteristics to local disk file system. In this section we extend our previous work to profile the checkpoint writing to a parallel file system. We run NAS parallel benchmarks LU and BT using the MVAPICH2 [5] with BLCR 0.8.0, and checkpoint the applications to PVFS2. We choose LU/BT Class C with 64 processes. The applications are run on a InfiniBand cluster. All nodes in the cluster are connected with DDR InfiniBand HCAs. Each node has 8 processor cores on 2 Intel Xeon 2.33 GHz Quadcore CPUs. The application runs on 8 nodes with one process per core. Each application process writes its checkpoint data to a separate file on a shared PVFS 2.8.1 file system. IB transport is chosen for this PVFS filesysetm. 4 data servers and used both as data server and metadata servers.

Table 1 lists some basic information of one checkpoint with NAS benchmark LU/BT of class C using 64 processes on 8 compute nodes. As indicated in section 2.3, checkpointing a parallel application consists of 3 phases. The cost of Phase 1 and Phase 3 is relatively constant given the application size. The cost of Phase 2 depends on both size of checkpoint data and storage system write bandwidth. In our experiments we find that phase 2 consumes the majority part of the checkpoint time. The first row in table 1 indicates the time cost in phase 2 to write the checkpoint data to a PVFS2 file system. From now on we only focus on Phase 2 in a checkpoint.

Table 1: Checkpoint Information on Phase 2(Class C, 64 processes, 8 processes per compute node)

	LU	BT
Time to Write One Checkpoint(seconds)	12.43	19.68
Checkpoint Data Size(MB) per Node	184.0	320.0
Number of VFS Write per Process	975	1057
Total Number of VFS Writes per Node	7800	8456

We profiled the checkpoint writing for these applications to understand the checkpoint file write patterns. Table 2 gives an example of the application LU of class C and 64 processes. It decomposes the checkpoint writing into different categories according to the size of writes. The first column is the size of write belonging to that category. The second column is percentage of writes within that range. The third column is percentage of data amount written by that type of write. The fourth column is percentage of time spent by VFS writes belonging to that category. The basic trends we observe here is similar to our previous observations in [26, 27].

Table 2: Checkpoint Writing Profile of LU.C.64

	% of Writes	% of Data	% of Time
0-64	50.86	0.04	0.12
64-256	0.61	0.00	0.00
256-1K	0.25	0.01	0.02
1K-4K	9.46	1.53	0.02
4K-16K	36.49	11.36	46.12
16K-64K	0.74	0.77	10.35
64K-256K	0.49	3.79	9.47
256K-512K	0.25	3.58	0.95
512K-1M	0.61	17.72	11.92
> 1M	0.25	61.21	21.06

(1) Most of the file writes only write small amount of data (smaller than 4KB per write). These small writes make up over 60% of all file writes, but they only write about 1.5% of total amount of data being dumped. It costs less than 0.2% of total time to perform these small writes. These small writes are primarily storing CPU registers, signal handler table, timers, open-file tables, process/group/session ids, and various other of BLCR data structures necessary to restore a process. Although large in numbers, all these small writes are buffered in VFS buffer cache. Therefore they only consumes a tiny fraction of all write time.

(2) There are a few large writes (greater then or equal to 512 KB per write). These writes constitute only about 0.8% of all writes, but they contribute about 79% of all data dumped. These writes consume about 34% of total write time.

(3) In between small and large writes are medium writes, which make up about 38% of all writes. They contribute about 20% of all data, but consume about 65% of all time. The medium and large writes actually store the virtual memory area (VMA) of a process. BLCR scans all VMAs of a process, and saves non-zero contiguous data pages to the checkpoint file. An application process usually has many VMAs. Many VMAs contain a handful contiguous pages that need to be dumped to file, which become a medium write. A few VMAs contain large block of contiguous pages to dump. They are the source of large writes.

During checkpoint writing, all application processes in a compute node perform a lot of VFS writes to the shared file system. What a data server sees is a lot of intermixed data streams coming from many processes. This inevitably causes frequent disk head seeks which can severely deteriorate the storage write bandwidth. In the following section, we propose a new design that can greatly improve checkpoint write bandwidth achieved by a parallel application.

4 Improving Checkpoint Write Performance with Aggregation and Staging I/O

In this section we present the Aggregated Staging I/O design that can greatly improve write bandwidth for checkpoint writing. The design consists of two components: the computer nodes side(client side) and the data server side(server side). As indicated in Figure 1, the shadowed boxes represent our new design on both client and server sides.

4.1 Write Aggregation on Client Side

In traditional checkpointing, each application process independently issues a sequence of VFS writes to separate checkpoint files. If not optimized, the interference of intermixed VFS write streams can severely degrade aggregated write bandwidth. Therefore we propose to insert an interposition layer between the application VFS writes and the actual data movement between client and data servers. This layer aggregates all VFS writes from the application, and takes care of moving data to data servers.

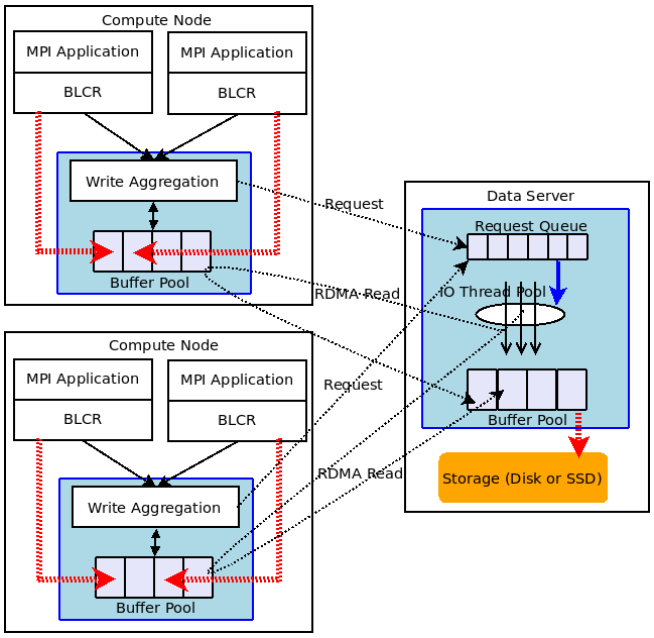

Figure 1: Checkpoint Write Staging

In our new strategy, the "Write Aggregation" module prepares a buffer pool by registering the buffer pool to the InifiniBand hardware. Each application process will request the "Write Aggregation" module to allocate a memory chunk from the buffer pool. As shown in Figure 1 on client side, when a VFS write is called by application process, instead of following the usual path to store data to a VFS buffer cache, the data is directly copied to the memory chunk associated to that process. When the chunk is full, the process requests another free chunk from the "Write Aggregation" module, and continues with checkpoint writing. Meanwhile for every filled chunks in the buffer pool, "Write Aggregation" module delivers a "RDMA-Read request" to data server. The data server then pulls the data to server side through RDMA Read. By aggregating many VFS writes into a buffer pool at client side, we expect to harvest significant improvement in file writing bandwidth at the cost of additional memory usage. A recent study [7] suggests that even large scale parallel jobs seldom use all available local memory. Therefore we feel it is reasonable to assume that residential memory is available in the client side. Our experiments indicate that even a mildly-sized buffer pool(64MB) can greatly improve the write bandwidth.

Although VFS write also stores the data into a VFS buffer cache by default, we believe our strategy outperforms the default VFS write data path for 3 reasons.

(1) Traditionally each application process has a data stream with a corresponding opened file. For a parallel file system, the cost to open/close many files can become significant at larger scale applications since the metadata server may become the bottleneck. Our strategy effectively coalesces multiple data streams into one with reduced metadata-related overhead.

(2) Parallel file systems usually flush client side data to servers at relatively small data size, which results in higher network round trip delay and lower bandwidth usage. Our strategy, on the contrary, can select proper chunk size to achieve better network bandwidth. Although one can specify larger flush size for the parallel file system, this may adversely affect other applications using the same parallel file system.

(3) Each VFS write system call involves cost at kernel/user context switch. An ever large overhead comes from the management of the complicated VFS buffer cache shared by many files. On the contrary, our strategy translate a VFS write to a swift memory copy.

Design alternatives exist to choose a proper buffer pool size and chunk size. Large buffer pool always help to improve the write bandwidth. Our experiments choose a mild size 64MB to be the buffer pool size. Once the chunk size is reasonably large(greater than 1MB), we find that the performance won't change much. So we will stick to 4MB chunk size in this paper.

4.2 Staging Area on Server Side

The data server maintains a queue to receive all requests from clients. Each request contains information such as: process ID, data size, offset of data in original checkpoint, buffer address to be used in RDMA-Read, remote key to be used in RDMA-Read. Once a request is enqueued, it's dispatched to a free IO thread out of the IO thread pool. This IO thread grabs a free chunk of memory in the local buffer pool, then issues a RDMA-Read operation to pull data from client memory to server memory. Once the data is present in the memory chunk, the IO thread appends this data chunk to local storage in a log-based file structure [8]. The persistent storage can be a disk file, or even a raw block device. The metadata about this chunk (process ID, data size, offset of data, physical offset in data server storage, etc.) is also saved for all IO threads. After the file write finishes, the IO thread sends a completion message back to the client. The client will then release the memory chunk to be used by other processes. Figure 2 illustrates this procedure.

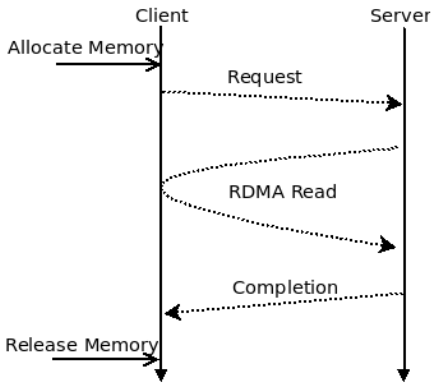

Figure 2: RDMA-Read Based Checkpoint Writing

In our design, we choose to let the client expose its memory to server, so server can perform RDMA-Read to pull the data. This approach owns the merit of better security over the alternative, where server exposes its memory to client and client performs RDMA-Write to push the data to server.

4.3 Reconstruct Checkpoint Files

Our design alters the structure of data stored in the persistent storage. At restart, it's necessary to reconstruct the data into the original checkpoint file format required by BLCR. When a data server writes the data chunks to persistent storage, the metadata for each data chunk (process ID, data size, offset of data) are cumulated and saved to an index file. At restart, a client node collects the index files from all data servers. These index files are parsed and the metadata of data chunks belonging to this client is condensed. Then the client can contact the data servers to retrieve corresponding data. These data chunks are then concatenated to rebuild the original checkpoint files.

5 Experimental Results

In this section, we conduct various experiments to evaluate the performance of our design. A 64 nodes RedHat Enterprise Linux 5 cluster is used in the evaluation. Each node has 8 processor cores on 2 Intel Xeon 2.33 GHz Quad-core CPUs. The nodes are connected with Mellanox MT25208 DDR InfiniBand HCAs. All our experiments are based on MVAPICH2 1.4 as the MPI library with BLCR 0.8.0.

The client side "Write Aggregation" design can be implemented as a stackable file system to intercept

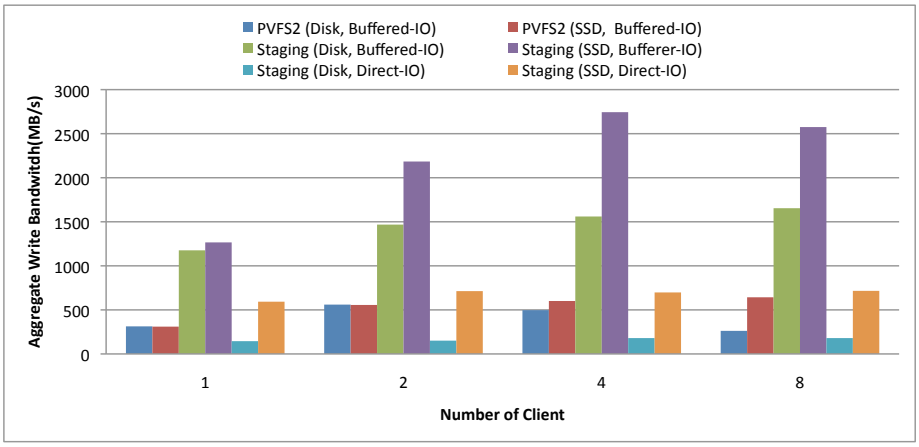

Figure 3: Aggregated Write Bandwidth with 4 Data Servers

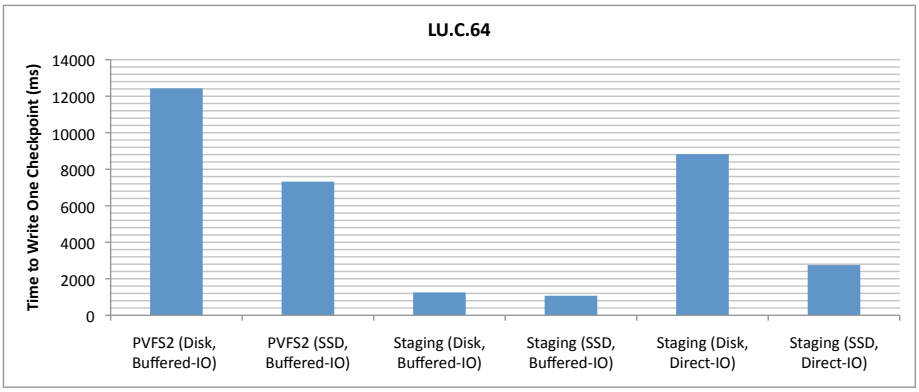

Figure 4: Checkpoint Write Time: LU.C.64

the VFS write system calls. In this experiment, we have modified BLCR kernel module to redirect VFS writes to the aggregation module. Client buffer pool is fixed to 64MB with 4MB chunk size. At the server side Staging Area design, 16 IO threads are used in the thread pool. The same buffer pool size(64MB) and chunk size(4MB) are used. Both hard disk and Solid State Disk(SSD) are tried as storage device at the server side. Table 3 gives the raw write bandwidth of these two kinds of storage devices.

Table 3: Device Raw Write Bandwidth (MB/second)

	Write	Read
Hard Disk	57	65
SSD(Intel X-25E 64GB)	179	202

5.1 Aggregated Write Bandwidth

We first run a synthetic benchmark to measure the aggregated write bandwidth achieved by our Aggregation Staging IO strategy. In this benchmark, 4 nodes act as data servers. All participating client nodes synchronize at a barrier, then start to write 1GB data at 4MB chunk size to the data servers. When all client nodes finish writing, they enter another barrier, after which the elapsed time T is measured. The aggregated write bandwidth is derived as $(TotalDataAmount)/T$. Both PVFS2 and Aggregated Staging IO strategy are evaluated as a comparison. Only buffered IO mode is tested for PVFS2, while both buffered IO and direct IO modes are evaluated with Aggregated Staging IO strategy. The data servers use hard disk and SSD as storage device at different runs. Figure 3 shows the aggregated write bandwidth using 1/2/4/8 client nodes.

PVFS2 with hard disk is used as a baseline for comparison. As client node number varies from 1/2/4/8, the aggregated write bandwidth is 313/560/497/262 MB/s respectively. It's clear that the interference between multiple data streams at data servers lead to a performance degradation when 8 client nodes are used. Since SSD has no mechanical seek latency as hard disk, the write stream interference should

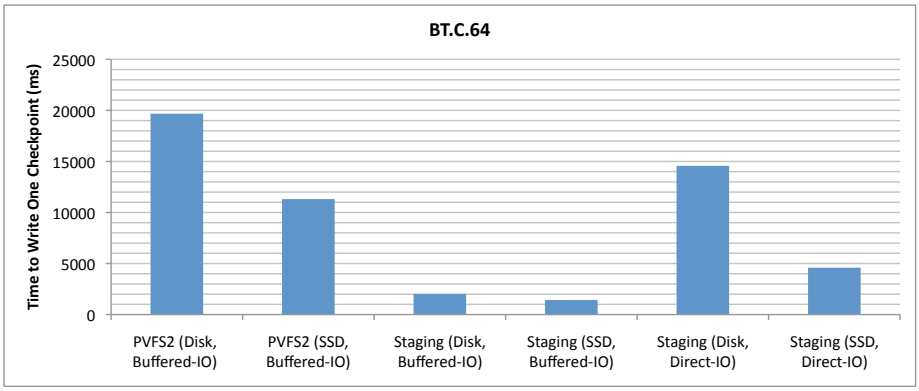

Figure 5: Checkpoint Write Time: BT.C.64

exhibit little impact on its write throughput. This is consolidated by our experiment to substitute disk with SSD. PVFS2+SSD achieves a aggregated write bandwidth of 310/556/601/643 MB/s, a big improvement over PVFS2+disk at 8 client nodes.

As a comparison we measured the performance of Aggregation Staging IO strategy. Staging IO with hard disk and buffered IO reaches aggregated write bandwidth of 1176/1468/1560/1654 MB/s when using 1/2/4/8 client nodes. Our strategy outperforms PVFS2+disk by 6.31 times at 8 client nodes. When replacing disk with SSD(Staging IO with SSD and buffered IO), we can achieve write bandwidth of 1266/2184/2744/2575 MB/s, 55% higher than Staging IO with disk at 8 clients. This improvement can be attributed to the better raw bandwidth of SSD over hard disks. If newer PCIe based SSD like Fusion-IO [2] is used, we will expect to gain more benefit in bandwidth. At 8 client nodes, Staging IO+SSD outperforms PVFS2+SSD by 4 times.

When direct IO is used instead of buffered IO, our staging IO with disk reaches write bandwidth of 145/151/180/180 MB/s. If using SSD as storage, the write bandwidth is 593/713/698/716 MB/s. Our strategy can saturate the 4 SSDs even with 2 client nodes.

5.2 Application Checkpoint Time

In this test we measure the time to checkpoint some parallel applications(LU and BT) from NAS parallel benchmark of class C and 64 processes. 8 compute nodes are used to run the application, and 4 storage nodes act as data servers. The benchmarks are compiled with MVAPICH2 1.4 Checkpoint/Restart and modified BLCR 0.8.0.

Figure 4 gives the time cost to write checkpoint data of application LU with different strategies. For PVFS2 with disk, 12.4 seconds are required to write the checkpoint data. PVFS2+SSD uses 7.3 seconds to complete the writing. If Staging IO with disk and buffered IO is used to write checkpoint data, the writing can be completed in 1.25 seconds, which is 9.9x improvement over PVFS2+disk. This time can be further driven down to 1.07 seconds if replacing disk with SSD. As with direct IO, Staging IO with disk can finish checkpoint writing in 8.83 seconds. Staging IO with SSD can finish writing in 2.75 seconds.

The similar trend can be observed for application BT in Figure 5. The checkpoint writing time can be reduced from 19.67 seconds(PVFS2+disk and buffered IO) to 2.02 seconds(Staging IO + disk and buffered IO), which represents a 9.7 times improvement. This time is driven down to 1.43 seconds by Staging IO with SSD and buffered IO.

6 Related Work

The overhead of checkpoint/restart on file IO has been studied by [17]. [26, 27] explore how to utilize write aggregation to improve checkpoint writing to a local file system. In this paper we extend the work to store checkpoint data to a shared parallel storage system. Milo etc. [8] proposes the use of log-based file structures at the server side to serialize all file writing requests for checkpoint. The parallel file system has to be altered to adopt this design. Stdchk [11] tries to scavenge spare storage resources from all participating nodes to form a dedicated storage space for checkpoint data. [9] developed a prototype distributed filesysetm to accelerate aggregated write bandwidth of checkpoint data. Our work differs from both in that we utilize aggregation on client side and staging IO on server side.

The work of PLFS [12] proposes to design a stackable file system on top of a parallel file system to remap the logical checkpoint data layout to a physi-

cal layout favorable to parallel filesysetm. The actual data storage is offloaded to the parallel filesysetm. On the contrary, our strategy doesn't rely on a parallel file system.

7 Conclusion and Future Work

In this paper we propose an Aggregation Staging IO to enhance write bandwidth of checkpoint data. We also evaluated the potential benefits of replacing hard disk with SSD in such a strategy. With Aggregation and Staging IO, our design is able to accelerate checkpoint writing significantly.

As part of the future work, we intend to conduct the experiments in larger scale. We plan to develop a stackable file system at client side based on FUSE [1] to implement the functionality of write aggregation. Additionally we want to compare our new strategy with other popular parallel file system such as Lustre [4].

References

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

[2] Fusion IO. http://www.fusionio.com/.

[3] InfiniBand Trade Association. http://www.infiniband.org/.

[4] Lustre Parallel Filesystem. http://wiki.lustre.org/.

[5] MPI over InfiniBand Project. In *http://nowlab.cse.ohio-state.edu/projects/mpi-iba/*.

[6] PVFS2. http://www.pvfs.org/.

[7] Application Requirements and Objectives for Petascale Systems . HPCwire, February 2008.

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

[9] Paul Nowoczynski, Nathan Stone, Jared Yanovich, Jason Sommerfield . Zest: Checkpoint Storage System for Large Supercomputers. In *PDSI 2008 workshop in conjunction with SC08* , Nov. 2008.

[10] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. Datastager: scalable data staging services for petascale applications. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, 2009.

[11] S. Al-Kiswany, M. Ripeanu, S.S. Vazhkudai, and A. Gharaibeh. stdchk: A checkpoint storage system for desktop grid computing. June 2008.

[12] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. Plfs: a checkpoint filesystem for parallel applications. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[13] Avery Ching, Alok Choudhary, Kenin Coloma, Wei-keng Liao, Robert Ross, and William Gropp. Noncontiguous i/o accesses through mpi-io. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, 2003.

[14] Avery Ching, Alok Choudhary, Wei keng Liao, Robert Ross, and William Gropp. Efficient structured data access in parallel file systems. *Cluster Computing, IEEE International Conference on*, 2003.

[15] Avery Ching, Alok Choudhary, Wei-Keng Liao, Robert Ross, and William Gropp. Evaluating structured i/o methods for parallel file systems. *Int. J. High Perform. Comput. Netw.*, 2.

[16] J. Duell. The design and implementation of berkeley lab's linux checkpoint/restart. Lawrence Berkeley National Laboratory,Paper LBNL-54941, April 2005.

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

[18] Sang-Won Lee and Bongki Moon. Design of flash-based dbms: an in-page logging approach. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007.

[19] Pin Lu and Kai Shen. Multi-layer event trace analysis for parallel i/o performance tuning. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 12, Washington, DC, USA, 2007. IEEE Computer Society.

[20] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. In *Technical Report UW-CS-TR-1346, University of Wisconsin-Madison, Computer Sciences Department*, April 1997.

[21] Paul H. Hargrove and Jason C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In *SciDAC*, 6 2006.

[22] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[23] J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[24] Q. Gao, W. Huang, M. Koop, and D. K. Panda. Group-based Coordinated Checkpointing for MPI: A Case Study on InfiniBand. In *Int'l Conference on Parallel Processing (ICPP)*, XiAn, China, 9 2007.

[25] 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.

[26] Xiangyong Ouyang, Karthik Gopalakrishnan and Dhabaleswar K. Panda. Accelerating checkpoint operation by node-level write aggregation on multicore systems. *ICPP 2009*, September 2009.

[27] Xiangyong Ouyang, Karthik Gopalakrishnan, Tejus Gangadharappa and Dhabaleswar K. Panda. Fast checkpointing by write aggregation with dynamic buffer and interleaving on multicore architecture. *HiPC 2009*, December 2009.

[28] Xuechen Zhang, Song Jiang, and Kei Davis. Making resonance a common case: A high-performance implementation of collective i/o on parallel file systems. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, 2009. IEEE Computer Society.