Proceedings of the 3rd Petascale Data Storage Workshop held in conjunction with Supercomputing '08,
November 17, 2008, Austin, TX.

Fast log-based concurrent writing of checkpoints

Milo Polte*, Jiri Simsa*, Wittawat Tantisiriroj*, Garth Gibson*, Shobhit Dayal*,
Mikhail Chainani*, Dilip Kumar Uppugandla*
*School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, 15213

Abstract

This report describes how a file system level log-based technique can improve the write performance of many-to-one write checkpoint workload typical for high performance computations. It is shown that a simple log-based organization can provide for substantial improvements in the write performance while retaining the convenience of a single flat file abstraction. The improvement of the write performance comes at the cost of degraded read performance however. Techniques to alleviate the read performance penalty, such as file reconstruction on the first read, are discussed.

I. Introduction

Parallel lesystems generally need to support concurrent write sharing by many compute nodes on the same le at the same time. Perhaps the most common concurrent write sharing access pattern is the parallel application checkpoint. A checkpoint is the image of all the state of a (parallel) application stored on disk. Parallel applications do this because they run on such large computers that the time between application crashes is quite small, 8-48 hours in the biggest computers, and because these applications quite often run for days and sometimes weeks in order to complete their task. So checkpoints are taken periodically so that after each application failure, the entire set of all clients in the parallel application abort then reload from the last checkpoint and start again [1] [2].

While it is certainly possible for a parallel checkpoint to be composed of one le per processor in the parallel computer, all clients writing to a single file may be more desirable to make it easier to load the checkpoint on a different configuration of nodes, to avoid the congestion of creating thousands of files at once, and for ease of management. Unfortunately, if each client does not write to the single file sequentially (say, numerous clients creating a single HDF5 file [3]), the series of small writes interspersed with seeks may cause performance to drop off dramatically.

In this report, we investigation a log based approach to handle this important and complex access pattern. Suppose the le system represented the writing each client is doing as a log of write commands stored in a sequentially written log le, similar to the log structured filesystem [4]. If this logging is done at the filesystem layer, it can hide the on disk representation under the abstraction of a single flat file without requiring application awareness. Of course, such an encoding requires either long read accesses, reconstruction of the le on the rst read, or an approach in between. However, this may be acceptable for the checkpointing workload, where write speed is paramount and most files are never read back.

II. Design

Design goals of our solution are the following. First and foremost, we want to optimize performance for concurrent and random access writes. Second, maximize stability and reliability by minimizing complexity of implementation. Finally, we would like our implementation to provide for reasonable performance of a read operation.

A. Write Operation

In concurrent and random access writes, disk latency (both seek and rotational) is the main reason for poor performance. To achieve the first goal, this design attempts to reduce disk latency by always writing data sequentially in the file logical address space. We restructure the regular file as a log file to optimize writes. When a write request comes, instead of seeking to the requested offset and then writing data, each write request is encoded into a chunk (as depicted in Fig. 1), which includes the original offset and the size of the data followed by data. This chuck is always appended to the end of file (as depicted in Fig. 1).

Fig. 1: File Format and Chunk Format

File:	Chunk-1	Chunk-2	Chunk-3	···	Chunk-N
Chunk:	offset (8 bytes)	size (8 bytes)	data (size bytes)		

For example, the command `pwrite(file, buffer[256], 64, 1024)` will be encoded as the chunk depicted in Fig. 2.

Fig. 2: Chunk example

1024	64	buffer[256]..buffer[319] (64 bytes)

B. Read Operation

In order to satisfy both second and third requirement, the implementation for the read operation must be simple yet optimized for performance. In our implementation, each read request requires a traversal of all chunk headers that are distributed across the entire file. If the data within any chunk falls within the requested area, the data will be read into a buffer in the same order they have been written to the file. This style of organising the chunk headers and reading them will deteriorate the read performance. In section IV we discuss several other approaches to improve read performance.

III. Evaluation

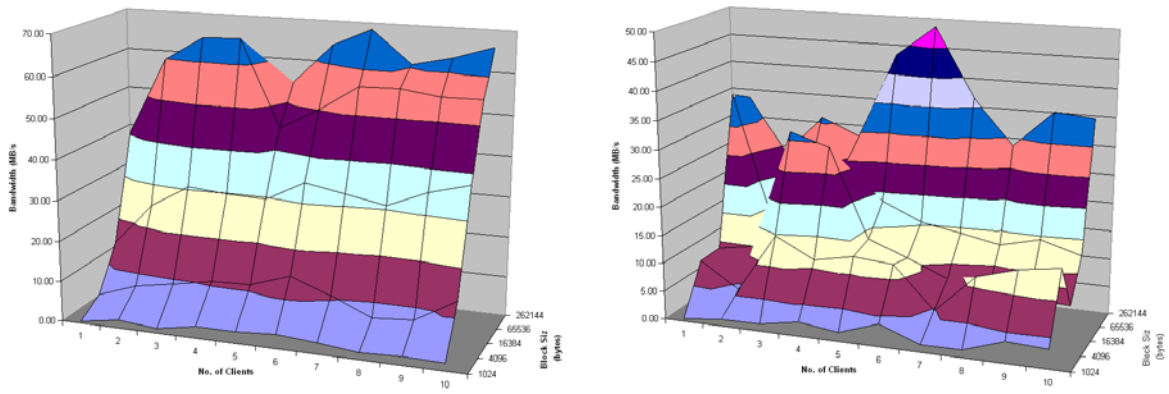
A. Experimental Setup

For this experiment, we modified the Parallel Virtual File System version 2 [5] to implement our design. PVFS2 is an open source parallel filesystem, for high performance computing developed at Argonne National Laboratory. To test our performance, we used the MPI-IO Test benchmark developed at Los Alamos National Laboratory [6] running on top of the MPICH2 MPI library [7].

To achieve our design goals, the write path of PVFS2 was modified to open its internal storage files in append mode and every write wrapped with an appropriate header. The changes read path reflected the changes made to the write path. Every read scanned the chunk headers of the file for ranges of the requested data.

We evaluated the performance of our implementation on a single server with a SATA disk used for PVFS2 storage. We ran MPI-IO Test against the modified and the unmodified PVFS2 server varying the number of clients from 1 to 10 and the block size from 1K to 256K for each client. In each test, we kept the number of objects written by MPI-IO Test constant as 1024. We present below the results and its analysis.

Fig. 3: Write Bandwidth of modified (left) and unmodified (right) PVFS2 server



B. Write performance

The Fig. 3 and Fig. 4 compare the write performance of the modified PVFS2 to that of the original PVFS2. As expected for large block sizes 8K - 256K we observe improved performance. The performance increases both with the block size and the number of clients. This trend continues until the disk bandwidth is saturated. After this point, there is a small drop in performance when the number of clients are increased, as the server gets loaded with more concurrent requests than the disk can handle, and the contention on locks and other components of the local file system begin to degrade the performance. For small writes our performance is sometimes comparable, but mostly slightly worse than that of the original PVFS2. This happens for two block sizes—1K and 4K—described in detail below.

Block size 1K: At this block size, as the number of clients increase, the writes from multiple clients to an unmodified server, appear almost as small, random write requests. Unfortunately we are not able to leverage the advantage of converting random writes to sequential since each write size is too small. We don't transfer large chunks to disk, and wait a full rotation before the next small write can be serviced (though some of these may get coalesced by the disk IO scheduler). Also, our modified PVFS2 performance is further degraded by the need to reorganize the incoming write to include a header and a memory copy. For small writes these overheads are significant and lower our performance in comparison to the original server.

Block size 4K: In addition to the disadvantages of the 1K block size write, we have write alignment issues with 4K blocks. Our PVFS2 server runs on top of a 4K block ext2 filesystem. Under a 4K write pattern, unmodified PVFS2 makes aligned 4K writes. Our modified implementation, however, adds a header to this block and misaligns the request.

Fig. 4: Overall write performance

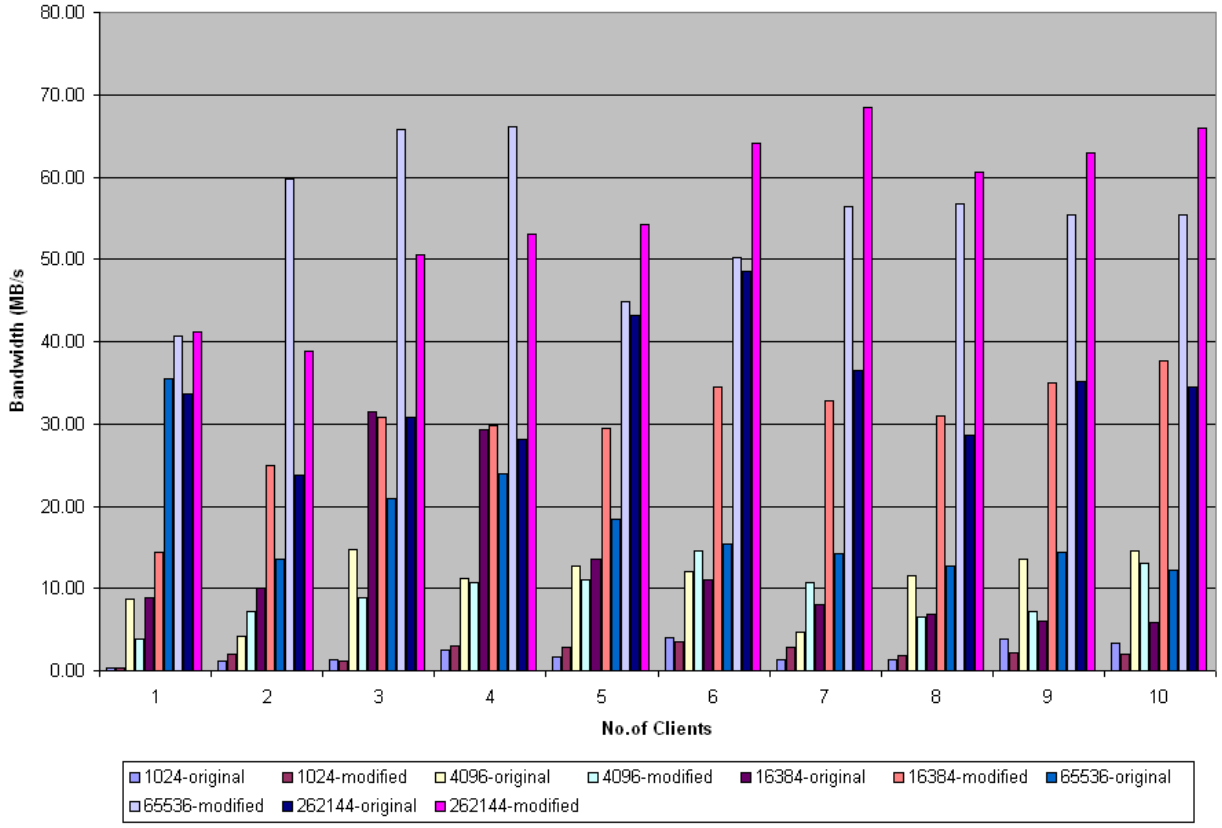
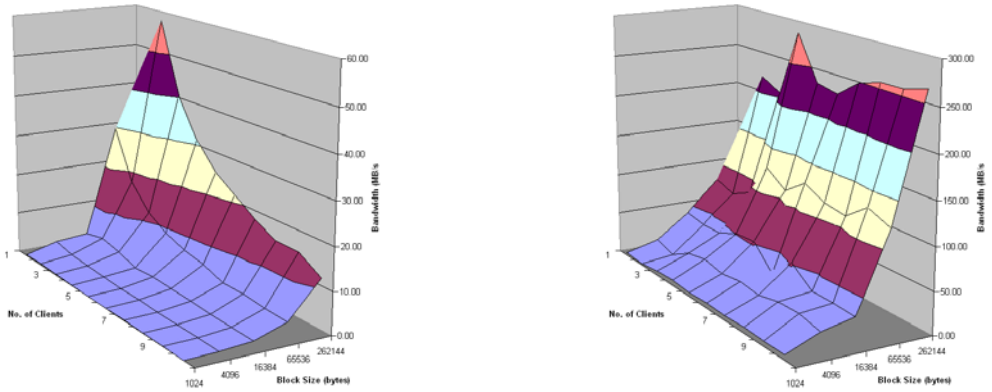


Fig. 5: Read Bandwidth of modified (left) and original (right) PVFS2 server

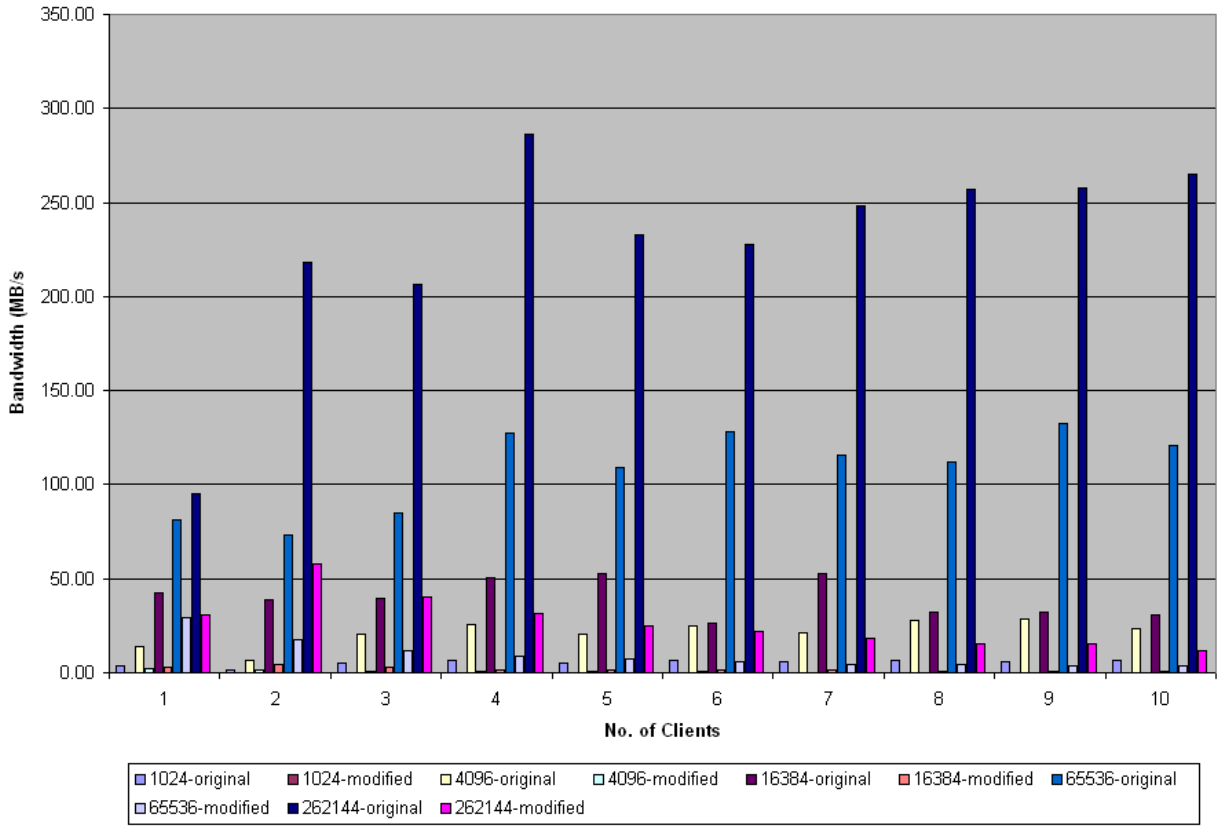


C. Read performance

The Fig. 5 and Fig. 6 compare the read performance of the modified PVFS2 to that of the original PVFS2. As expected our read performance is lower than that of the original PVFS2, especially at small block size. Furthermore, given our current design, the performance decreases with increasing number of clients as each client's read involves a scan of the entire file,

thrashing the cache when the file does not fit in the cache. Overall our read performance is poor across all configurations (1–10 clients, 1K–4K blocks) when compared to that of the original PVFS2. We discuss ways to improve our read performance in section IV.

Fig. 6: Overall read performance



IV. Future work

In the current implementation every read will scan the entire file for header information in order to find the latest write at the requested offset. Below we suggest some improvements for the read path.

A. Reversed Read

Instead of reading the chunk headers from the beginning to the end, a read operation would read the headers from the last chunk to the first. Essentially, this means replacing headers with footers so that backwards navigation is possible. Once we find a set of log entries covering our request we do not need to search anymore since we have the most recent data. This approach also solves the problem of reader starvation.

B. Decoupling data and metadata

We may associate a new file for each data file, that stores all the log headers. This file will be smaller and can be read in both directions to further speed up the read. One design issue with this approach is time of updating of this metafile. It may be done by a special thread that runs periodically or when the system is idle.

C. Flattening the file on the first read

The first read to a log structured file can process the entire contents storing it as a single chunk replacing the original file. This might hurt the performance of the first read, but will speed up the performance of subsequent reads.

V. Conclusion

As HEC applications move further into the Petascale, the speed of checkpointing operations will become increasingly important for fault tolerance. In this report, we have shown how simple mechanisms at the file system layer can improve the performance of the many-to-one checkpoint writing pattern, while still providing the convenience of a single flat file interface. There remain, however, research questions concerning how to access file efficiently by multiple readers during reconstruction.

Acknowledgment

The authors would like to thank to Mike Kassickm and Tomer Shiran for class project collaboration that help lay the foundations for this paper.

References

[1] "High End Computing Revitalization Task Force (HECRTF), Inter Agency Working Group (HECIWG) File Systems and I/O Research Workshop HECIWG," 2006. [Online]. Available: http://institute.lanl.gov/hec-fsio/docs/HECIWG-FSIO-FY06-Workshop-Document-FINAL-FINAL.pdf

[2] E. N. Elnozahy and J. S. Plank, "Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 2, pp. 97–108, 2004.

[3] S. Saini, D. Talcott, H. Yeung, G. Myers, and R. Ciotti, "A Scalability Study of SGI Clustered XFS Using HDF Based AMR Application," Terascale Systems Group, NASA Ames Research Center, Tech. Rep., 2006.

[4] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-structured File System," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, 1992.

[5] "Parallel Virtual File System, Version 2." [Online]. Available: http://www.pvfs.org/

[6] "Los Alamos National Labs MPI-IO Test." [Online]. Available: http://public.lanl.gov/jnunez/benchmarks/mpiiotest.htm

[7] "MPICH2: High-performance and Widely Portable MPI." [Online]. Available: http://www.mcs.anl.gov/research/projects/mpich2/