PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments*

Steven A. Moyer
V. S. Sunderam

Department of Mathematics and Computer Science
Emory University, Atlanta, GA 30322, U.S.A.
{moyer, vss}@mathcs.emory.edu

Abstract

PIOUS is a parallel file system architecture that provides cost-effective, scalable bandwidth in a network computing environment. PIOUS employs data declustering, to exploit the combined file I/O and buffer cache capacities of networked computing resources, and transaction-based concurrency control, to guarantee access consistency without explicit synchronization. This paper presents preliminary results from a prototype PIOUS implementation.

1 Introduction

Parallel programming environments that exploit networked computing resources offer a cost-effective alternative to traditional parallel machines. Environments such as PVM [14] and Linda [1], among others [15], enable parallel-distributed application development by providing mechanisms for interprocess communication, synchronization and concurrency control, fault tolerance, and process management. However, many parallel applications require or could benefit from a unified parallel I/O system that such environments generally lack.

PIOUS, the Parallel Input/OUtput System, is a parallel file system that provides process groups access to permanent storage in a network computing environment. PIOUS supports parallel application development by providing coordinated access to *parafile* file objects with sequential consistency semantics and a dynamically-selectable fault tolerance level. For cost-effective scalable bandwidth, PIOUS declusters files to aggregate the performance of networked resources.

This paper provides a brief overview of the PIOUS architecture and programming model and presents preliminary results from a prototype PIOUS implementation.

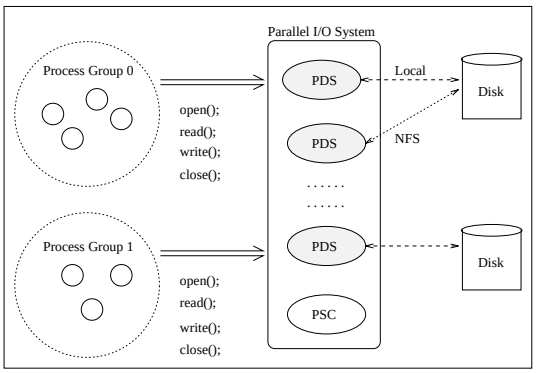

Figure 1: PIOUS software architecture

2 PIOUS Software Architecture

The PIOUS software architecture is depicted in Figure 1. PIOUS consists of a set of data servers, a service coordinator, and library routines linked with client processes. An underlying transport mechanism is assumed to carry messages between client processes and components of the PIOUS architecture. PIOUS data servers are assumed to access permanent storage via a native file system.

A PIOUS Data Server (PDS) resides on each machine over which files are declustered. Each PDS provides transaction-based access to the local files that

*Research supported by the National Science Foundation, under Award No. CCR-9118787, U. S. Department of Energy, under Grant No. DE-FG05-91ER25105, and the Office of Naval Research under grant N00014-93-1-0278.

represent a portion of a declustered file. PDS do not interpret the global structure of a file, however, thus separating file access mechanisms from policies. PDS are independent and hence do not communicate, enhancing system parallelism and scalability. Though ideally each PDS accesses a file system local to the machine on which it resides, two PDS may share a file system via a network file service.

A single PIOUS Service Coordinator (PSC) initiates activities within the system. For example, when a process opens a file the PSC is contacted to obtain file meta-data and to ensure that the requested file access semantics are consistent with those of other processes in the group. Note that the PSC only participates in major system events and not general file access, thus it does not represent a system bottleneck.

Finally, each PIOUS client process is linked with library routines that translate file operations into PDS/PSC service requests. Global file structure and access policies are implemented at this level, facilitating experimentation in user interface design without requiring re-implementation of the PDS or PSC.

The PIOUS architecture emphasizes asynchronous operation and scalable performance via the independence of parallel data servers, the interpretation of global file structure in client library routines, and the lack of a centralized manager for all but major events. Furthermore, the architecture is relatively free of assumptions concerning the underlying transport and storage subsystems. A more detailed treatment of the PIOUS architecture is presented in [9].

3 Transactions in PIOUS

Parallel I/O operations in PIOUS are performed within the context of transactions [2], transparently to the user, to provide sequential consistency [8] of access and tolerance of system failures. Referring to Figure 1, client library routines act as transaction managers for the data servers participating in a distributed transaction satisfying a user request.

The PIOUS architecture defines two distinct transaction types: *stable* and *volatile*. Both transaction types guarantee serializability of access. A stable transaction is a "traditional" transaction that employs logging and synchronous disk write operations to also guarantee that coherence is maintained in the event of a system failure. A volatile transaction is a "lightweight" transaction that does not guarantee fault tolerance. Providing two distinct transaction types allows user-selectable levels of fault tolerance to be implemented.

Specifying that PIOUS components interoperate via transaction-based protocols might seem to be an implementation detail rather than an architectural issue. However, employing transaction-based protocols allows elements of the PIOUS architecture to operate independently and obviates the need for special concurrency control and fault tolerance components.

4 PIOUS File Model and Interface

A PIOUS file structure and interface design are presented below. Only general PIOUS interface concepts are discussed; the current file system model is detailed in [9].

For the remainder of this text the term PIOUS refers to our particular implementation of the PIOUS architecture. Thus we discuss features of PIOUS not directly specified by the architectural description; e.g. file structure details.

4.1 Parafile Objects

Parafiles are implemented in PIOUS to facilitate parallel data mapping. Parafiles are logically single files composed of one or more physically disjoint segments. Each segment is composed of a linear sequence of zero or more bytes. The number of segments in a parafile is set at the time of creation and does not change for the life of the file.

Each parafile segment resides on a single PIOUS data server. If the number of parafile segments exceeds the maximum number of PDS for declustering the file, then segments are mapped to data servers in a round-robin fashion.

Figure 2 illustrates a parafile object with four segments declustered over two data servers. At each data server a directory contains file entries that represent parafile segments and associated meta-data; the directory path name is the logical file path name at the given server. Implementing parafiles on top of a conventional file system necessitates representing each parafile segment as a separate file.

Because parafiles are two-dimensional, array and tabular data map naturally. Thus developing parallel scientific, database, and sorting applications, among others, should be greatly simplified. The terms *file* and *parafile* are used interchangeably for the remainder of this discussion, as a parafile is the only file object implemented in PIOUS.

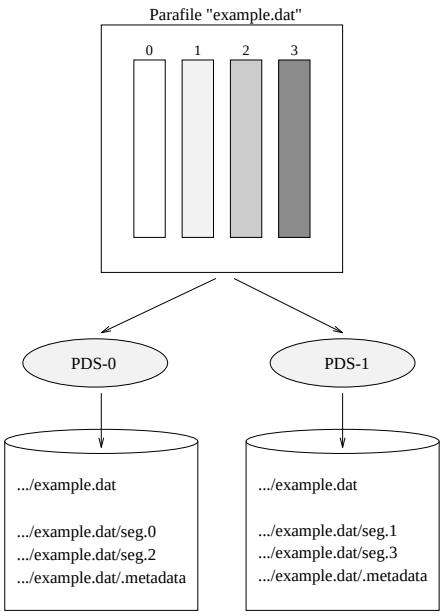

Figure 2: Parafile object

4.2 Parafile Access

The PIOUS interface provides a process group with three *views* of a parafile object: global, independent, and segmented. Each parafile view provides a process group with a distinct set of access semantics, as defined below:

global A file appears as a linear sequence of bytes; all processes in a group share a single file pointer; datum access and file pointer update are atomic.

independent A file appears as a linear sequence of bytes; each process in a group maintains a local file pointer; file access is atomic.

segmented The segmented structure of a parafile is exposed; each process accesses a segment via a local pointer; file access is atomic.

Under the views defined, parafiles are either accessed by segment or logically mapped to a linear sequence of bytes. A linear byte sequence is defined by ordering the bytes in a parafile by fixed size blocks taken round-robin from each segment. Note that a view only defines the way a parafile is accessed and does not alter the physical representation. Thus a given parafile can always be opened with any view. All processes in a group access a file with the same view.

Parallel file systems are relatively new and an area of active research. Currently no particular file structure or user interface has been embraced as being clearly superior; in fact, most commercial parallel file systems implement a traditional Unix[1] model. The set of file views currently supported in PIOUS is limited, though sufficient to implement a super-set of the common parallel file access patterns identified by Crockett [5]. We feel that until further experience is gained with the system, development of a more sophisticated interface is premature.

4.3 Consistency and Fault Tolerance

PIOUS provides all processes sequential consistency of access to files opened under any view. Between processes of different groups, some form of flexible file locking is being considered as well.

PIOUS supports parafile access in two dynamically-selectable fault tolerance modes: *volatile* and *stable*. Stable mode guarantees file consistency in the event of a system failure; volatile mode does not. Volatile mode is intended for non-critical applications that require high-performance. Applications requiring fault tolerance can access files in the lower-performance, but guaranteed consistent, stable mode.

The current PIOUS implementation does not provide applications with fault tolerance of media failures. Media fault tolerance is achieved by data replication on independent devices, either as multiple file copies [2] or as computed parity information [10]. Such techniques are independent of the file system architecture, provided that multiple independent storage devices can be accessed.

5 PIOUS Performance Study

Performance results for our prototype PIOUS implementation are presented below. The PIOUS prototype consists of fully functional data servers (PDS) and a limited set of library routines. The service coordinator (PSC) has not been implemented; however the PSC does not participate in general file access and thus does not impact our results.

The PIOUS prototype is implemented within the PVM [14] distributed computing environment. Standard PVM message passing services are utilized for all communication between components of the system. Data servers access files via the native file system interface.

[1] Unix is a registered trademark of X/Open Company Ltd.

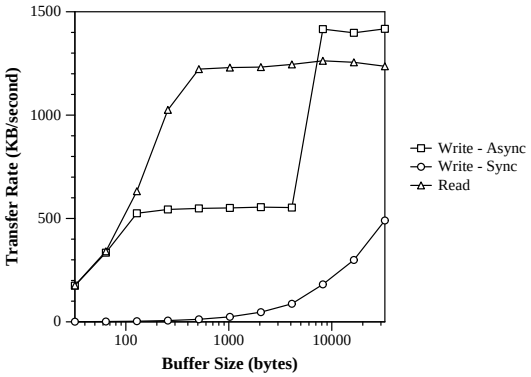

Figure 3: PDS local disk performance

5.1 Execution Environment

Performance measurements of the PIOUS prototype were taken in a network environment that consists of a set of Sun SS2 and IPC workstations connected via a 10Mb/second Ethernet. All machines are running the SunOS 5.3 operating system.

PIOUS data servers were placed on homogeneous Sun SS2 workstations each with 16MB of RAM and a Maxtor 7345S hard disk drive. Figure 3 depicts local disk drive performance, in terms of observed bandwidth, as a function of transfer buffer size for native read() and write() operating system calls; the term *transfer buffer* refers to the block of data involved in a read or write request. Write access performance is presented for both asynchronous (buffered I/O) and synchronous (non-buffered I/O) operations. File access is sequential on an otherwise quiescent system so that results represent best case performance; file size is 1MB for asynchronous access and 0.25MB for synchronous access.

5.2 PIOUS Software Overhead

Measuring PIOUS software overhead is important for assessing the viability of transactions as a concurrency control mechanism in network parallel file systems. To this end we have instrumented the PIOUS data server and measured the cost of servicing user read and write operations. Only PDS software overhead is measured; disk access, cache access, and data transport times are not included.

Table 1 depicts the PDS overhead for each component of a user read or write operation that accesses a file in volatile mode. Each user operation results in a number of read or write requests, followed by a single commit request, at each server containing a portion of the data referenced. Note that there are additional overheads associated with accessing a file in stable mode that are not presented here.

Operation	Overhead (microseconds)	
	Access Request	Commit Request
Read	293	133
Write	311	185

Table 1: PDS service overhead

Performance measurements indicate that for modest size data transfers, light-weight transactions can be implemented as an efficient concurrency control mechanism for parallel I/O operations. However it should be noted that the transaction mechanism increases message count in a network environment by introducing control messages not directly associated with requesting or supplying data; e.g. the commit message discussed above. Control messages are short, and hence consume little network bandwidth. Thus, while control messages may increase access latency for a given process, the aggregate data bandwidth of a process group should be only minimally effected.

5.3 File Access Benchmarks

Performance measurements are presented below for benchmark applications that implement two common parallel file access patterns identified by Crockett [5]: partitioned and self-scheduled. Partitioned access divides a linear file into contiguous blocks, with each block accessed sequentially by a different process. Self-scheduled access results when a linear file is accessed sequentially by a group of processes via a shared file pointer. Figure 4 illustrates both patterns for a hypothetical parallel computation consisting of four processes. In PIOUS, partitioned and self-scheduled access are implemented with the independent and global parafile views, respectively.

All benchmark applications access a parafile with a number of segments equal to the number of data servers. Thus the linear file mapping provided by the independent and global views results in a data distribution pattern that is equivalent to *disk striping* [12]. In disk striping, a linear sequence of fixed size data blocks, called *striping units*, are distributed in a round-robin fashion across some number of storage devices. A *data stripe* consists of the set of striping units distributed in a single round. This terminology is used in defining benchmark parameters.

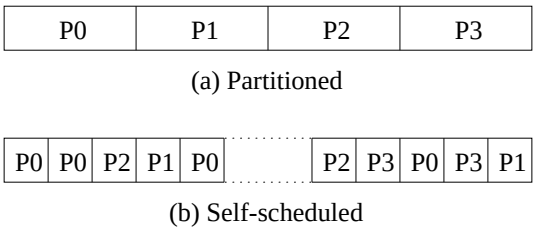

Figure 4: Partitioned and self-scheduled patterns

For each of the two access patterns discussed above, benchmark results are presented for systems of one to four data servers and one to twelve client processes. In all benchmarks, the transfer buffer size of each process request is 8000 bytes, the striping unit size is 4000 bytes, and accesses are 8000 byte aligned. Thus a single process can benefit from the parallelism of at most two PIOUS data servers. Files are accessed in both volatile and stable mode to demonstrate the costs associated with fault tolerance. Finally, to asses the benefits of caching in the PDS, file access is arranged such that all accesses in a given trial either cache-hit or cache-miss. Benchmark results are given in terms of the aggregate data rate obtained by all processes as perceived by an outside observer.

5.4 Partitioned Access Performance

Figure 5 depicts the performance obtained by a parallel application implementing a partitioned access pattern. For maximum performance the file is accessed in volatile mode.

Because of the relatively high latency of the PVM transport layer, requests from a single client do not saturate even a single data server. Thus for a fixed number of PDS, aggregate performance increases, up to some maximum, as more clients participate in the file access. Performance eventually begins to decrease somewhat as the number of clients is further increased due to increased contention for the Ethernet, an increased number of message interrupts at the server transport layer, and increased complexity of disk access patterns for trials that cache-miss.

Data caching at the PDS reduces service latency for accesses that cache-hit. As Figure 5 demonstrates, trials that cache-hit result in higher bandwidth than corresponding trials that cache-miss. Caching improves observed bandwidth significantly while performance is disk limited, and with diminishing effects as performance approaches that of the transport service.

The scalability of the PIOUS architecture is not

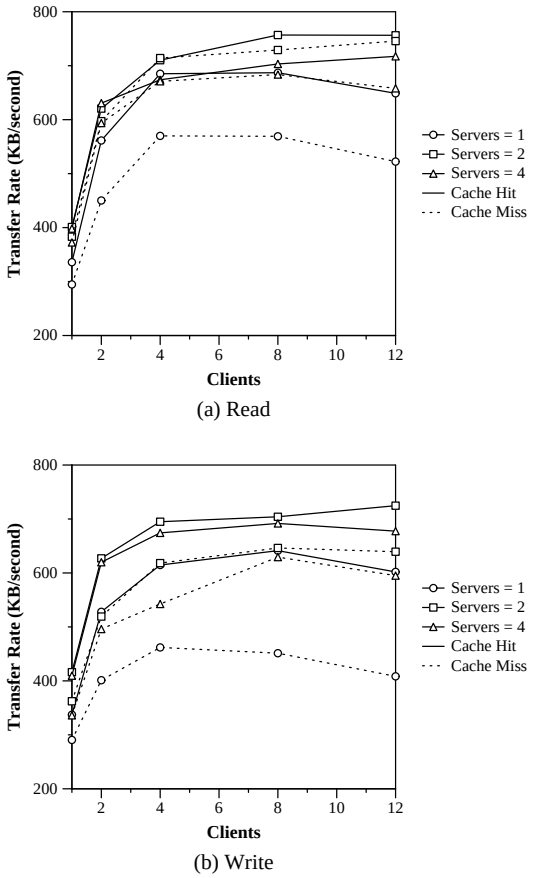

Figure 5: Partitioned access in volatile mode

well demonstrated in Figure 5 because the transport layer becomes the system bottleneck with two or more servers. In fact, two data servers often provide marginally better performance than four; we suspect this is due primarily to Ethernet contention.

Figure 6 depicts write performance obtained by a parallel application implementing a partitioned access pattern and accessing the file in stable mode. Stable mode provides fault tolerance to system failures at the cost of PDS performing logging and synchronous disk write operations. Read performance remains the same in stable and volatile mode and hence is not depicted.

Because synchronous write operations provide very low performance, as illustrated previously in Figure 3, and because logging guarantees at least one synchronous write per transaction, disk access becomes the system bottleneck in all trials of Figure 6 for a sufficient number of clients. Recall that with a transfer buffer size of 8000 bytes, a striping unit size of 4000 bytes, and an access offset that is a multiple of 8000

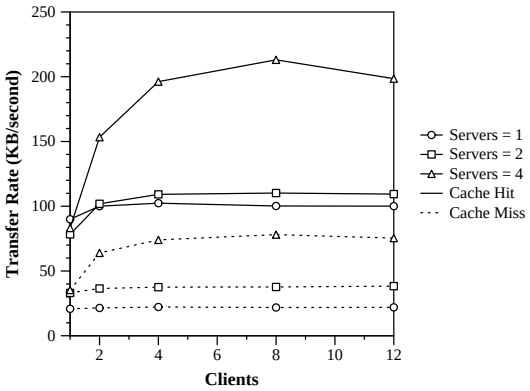

Figure 6: Partitioned write access in stable mode

bytes, each client can saturate at most two servers. For most trials, aggregate performance remains relatively constant for a fixed number of PDS so long as the ratio of clients to servers is at least one to two. Aggregate performance for all trials is less sensitive to a continued increase in the number of clients than the comparable case of volatile file access because lower data rates yield less Ethernet contention and a lower rate of message interrupts.

Data caching at the PDS is particularly effective for stable file access since the number of synchronous writes is halved, with a concomitant reduction in disk arm motion. For all trials, performance of accesses that cache-hit is more than double that of corresponding trials that cache-miss.

Figure 6 illustrates that the PIOUS architecture appears to scale well when disk performance is the system bottleneck and accesses do not require shared state, e.g. a global file pointer. Because the time to perform a synchronous write to disk is nearly constant for up to 8 kilobytes of data, performance scales linearly in the number of PDS that a single process can potentially saturate; two in this case. Thus in going from one PDS to two only a slight gain in performance is realized and results from additional concurrency in PDS-client communication. However, when the number of servers is increased from two to four, twice as many client accesses can proceed concurrently and hence performance doubles.

5.5 Self-Scheduled Access Performance

Figure 7 depicts the performance obtained by a parallel application implementing a self-scheduled access pattern and accessing the file in volatile mode. Recall that self-scheduled access implies that the file is

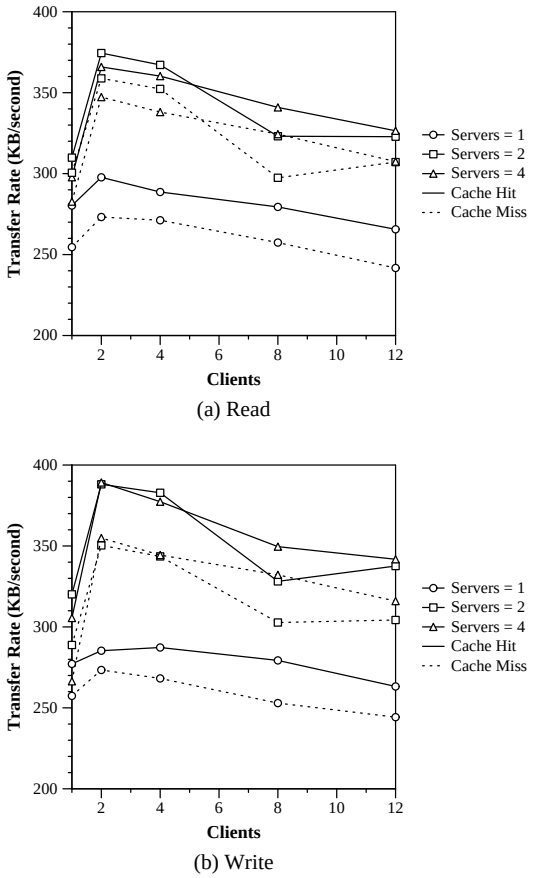

Figure 7: Self-scheduled access in volatile mode

accessed sequentially by a group of processes via a shared file pointer. In our PIOUS implementation, shared state is maintained in a PDS managed file.

Because the shared file pointer is updated as part of a single transaction implementing a user read/write request, access to the file is serialized[2]. Thus, for a fixed number of PDS, performance should remain relatively constant regardless of the number of clients. Figure 7 demonstrates an initial increase in performance as the number of clients is increased from one to two. This performance gain results from a one-time increase in PDS-client communication concurrency: a process blocked waiting to access the file pointer is sent the pointer value immediately after the process accessing the file completes that transaction. Once again performance begins to decrease somewhat as the

[2]Performance is increased significantly by releasing the shared pointer prior to data access. However the user must instruct the system to do so, as global access semantics can be violated under certain conditions.

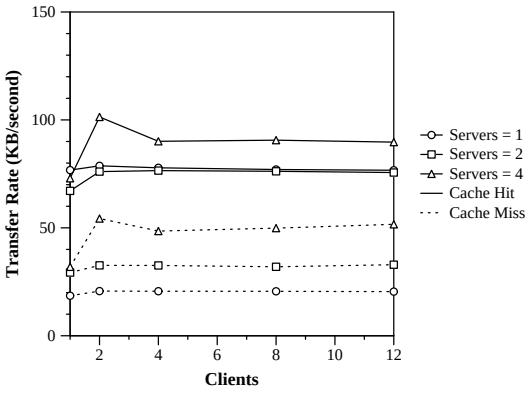

Figure 8: Self-scheduled write access in stable mode

number of clients is further increased due to increased contention for the Ethernet and an increased number of message interrupts at the server transport layer.

Naturally, self-scheduled access does not scale beyond the number of servers that a single client can saturate; two in this case as previously discussed. Thus, as Figure 7 demonstrates, additional servers beyond this number do not increase aggregate performance.

Figure 8 depicts write performance obtained by a parallel application implementing a self-scheduled access pattern and accessing the file in stable mode. Once again, read access performance remains the same in stable and volatile mode and hence is not depicted.

Performance curves for write operations retain the same general shape in stable mode as in volatile mode, albeit at a lower level due to the synchronous disk write and logging operations performed by the data servers. Again aggregate performance in stable mode is less sensitive to an increasing number of clients than an equivalent access pattern performed in volatile mode.

Note that Figure 8 depicts an increase in performance as the number of servers is increased from two to four. As discussed above, self-scheduled access should not scale in performance beyond the number of servers, two in this case, that a single client can saturate. Since the observed increase in performance can not be attributed to the PIOUS architecture, it must result from one of the underlying software systems; we suspect increased disk access efficiency resulting from smaller data and log files at each server as the number of servers is increased.

6 Related Work

The PIOUS architecture has roots in a number of network and parallel computer file systems. A brief overview of this related work is presented below.

Parallel machines commonly provide a parallel file system interface; familiar examples include the Intel CFS [11] and nCUBE Parallel I/O System [6]. Such systems provide parallel applications with varying degrees of file access coordination and control and often employ data declustering to increase aggregate performance. The IBM Vesta [4] multicomputer file system implements two-dimensional files and a unique stencil-based logical partitioning operator to coordinate parallel file access.

Swift [3] and Zebra [7] are research network file systems that implement data declustering across multiple servers with RAID-4/5 [10] fault tolerance of media failures. Zebra is unique in that it incorporates a log-structured file system that allows striping to be performed on a per client basis. However, Swift and Zebra are not parallel file systems as they lack support for parallel application development.

Express is a CUBIX-model [13] parallel computing environment that implements a parallel file system interface. However, in network environments, Express does not employ data declustering to aggregate the performance of distributed resources.

PIOUS extends the work of existing parallel file systems by incorporating transactions as a generalized concurrency control and fault tolerance mechanism. Consequently, PIOUS provides parallel applications with guaranteed consistency semantics, asynchronous file operations, and dynamically-selectable fault tolerance modes. Furthermore, to our knowledge PIOUS is the first network parallel file system to employ data declustering for scalable performance.

7 Conclusions

PIOUS is a parallel file system architecture for providing process groups access to permanent storage in a network computing environment. PIOUS is a true parallel I/O system that provides coordinated access to file objects with guaranteed consistency semantics. For performance PIOUS employs data declustering, to exploit the combined file I/O and buffer cache capacities of networked computing resources, and transaction-based concurrency control, to guarantee access consistency without explicit synchronization.

Measurements of a prototype PIOUS implementation indicate that, for modest size data transfers, the overhead introduced by the PIOUS software is minimal. This result validates our belief that lightweight transactions represent a viable concurrency control mechanism for network parallel file systems. Furthermore, by extending this mechanism to provide full transaction semantics, dynamically-selectable fault tolerance is easily provided.

For the high-performance volatile access mode, aggregate performance in our execution environment is limited by the latency and bandwidth of the transport mechanism for two or more data servers. Thus we are unable to measure scalability in volatile mode. For the fault tolerant stable access mode, aggregate performance is disk bandwidth limited. Measurements of stable mode access demonstrate a degree of scalability proportional to our limited resources.

The PIOUS prototype results are encouraging, demonstrating the potential of the PIOUS architecture for achieving cost-effective, scalable file system performance in a network computing environment.

8 Acknowledgements

The authors wish to thank the anonymous referees for their advice on improving this paper.

References

[1] M. Arango, D. Berndt, N. Carriero, D. Gelernter, and D. Gilmore. Adventures with network Linda. *Supercomputer Review*, 10(3):42–46, October 1990.

[2] Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, 1987.

[3] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4), Fall 1991.

[4] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Sandra J. Baylor. Parallel access to files in the Vesta file system. In *Proceedings of Supercomputing '93*, pages 472–481, November 1993.

[5] Thomas W. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

[6] Erik DeBenedictis and Juan Miguel del Rosario. nCUBE parallel I/O software. In *Eleventh Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC)*, pages 0117–0124, April 1992.

[7] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 29–43, December 1993.

[8] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[9] Steven A. Moyer and V. S. Sunderam. A parallel I/O system for high-performance distributed computing. In *Proceedings of the IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems*, April 1994. To appear.

[10] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[11] Paul Pierce. A concurrent file system for a highly parallel mass storage system. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160, 1989.

[12] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *IEEE 1986 Conference on Data Engineering*, pages 336–342, 1986.

[13] John Salmon. CUBIX: Programming hypercubes without programming hosts. In *Proceedings of the Second Conference on Hypercube Multiprocessors*, pages 3–9, 1987.

[14] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[15] L. H. Turcotte. A survey of software environments for exploiting networked computing resources. Technical report, Mississippi State University, February 1993.