This paper will appear in the Proceedings of the IEEE CompCon conference, March 5-8, 1995, San Francisco.

The Scotch Parallel Storage Systems

Garth A. Gibson, Daniel Stodolsky, Fay W. Chang, William V. Courtright II, Chris G. Demetriou, Eka Ginting, Mark Holland, Qingming Ma, LeAnn Neal, R. Hugo Patterson, Jiawen Su, Rachad Youssef, Jim Zelenka

Parallel Data Lab, School of Computer Science, Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
URL: http://www.cs.cmu.edu:8001/Web/Groups/PDL

Abstract

To meet the bandwidth needs of modern computer systems, parallel storage systems are evolving beyond RAID levels 1 through 5. The Parallel Data Lab at Carnegie Mellon University has constructed three Scotch parallel storage testbeds to explore and evaluate five directions in RAID evolution: first, the development of new RAID architectures to reduce the cost/performance penalty of maintaining redundant data; second, an extensible software framework for rapid prototyping of new architectures; third, mechanisms to reduce the complexity of and automate error-handling in RAID subsystems; fourth, a file system extension that allows serial programs to exploit parallel storage; and lastly, a parallel file system that extends the RAID advantages to distributed, parallel computing environments. This paper describes these five RAID evolutions and the testbeds in which they are being implemented and evaluated.

1 Introduction

As information systems become increasingly critical, the demand for high-capacity, high-performance, highly available, storage systems increases. The introduction of parallel processing, coupled with the unrelenting pace of microprocessor performance improvements, has converted many traditionally compute-constrained tasks to ones dominated by I/O. Redundant Arrays of Inexpensive Disks (RAID), as defined by Patterson, Gibson, and Katz [1], has emerged as the most promising technology for meeting these needs. Consequently, the market for RAID systems is undergoing rapid growth, exceeding three billion dollars in 1994 and expected to surpass 13 billion dollars by 1997 [2].

However, RAID storage is not without limitations. First, there are cost and performance penalties for maintaining a redundant encoding of stored data. Overcoming these penalties continues to spur the development of new variations of RAID architectures. Second, while the rapid invention of clever new architectures is important, it exacerbates the need for a high-fidelity framework for rapid development and evaluation of new designs. Third, the complexity of fault-tolerance is becoming more unmanageable with each new optimization incorporated. Fourth, even ignoring the implications of failures, many workloads generate I/O accesses with inadequate concurrency or sequentially to efficiently exploit parallel storage. Finally, RAID architectures directly attached to a host system bus are inherently not scalable.

In this paper we present research projects addressing each of these five challenges for parallel storage systems. We begin, in Section 2, with an overview of the experimental testbeds used to demonstrate and evaluate our research. Section 3 focuses on the first three limitations, all of which arise and can be addressed within directly attached RAID subsystems. It presents a variant of RAID level 5 that improves on-line failure recovery performance, an extensible framework for evaluating RAID architectures, and a methodology for structuring RAID control software that automates error handling. Section 4 presents informed prefetching and scalable, parallel file systems research that address the latter two limitations through application disclosure of future accesses and application coordination of parallel file system synchronization.

2 Scotch Experimental Testbeds

The Parallel Data Lab at Carnegie Mellon University contains three experimental "Scotch" testbeds for parallel storage research. In the sections that follow we describe the research that is being evaluated in each testbed.

The first Scotch testbed, Scotch-1, no longer in use, was primarily used for the prefetching file systems research described in Section 4. As shown in Figure 1, Scotch-1 is composed of a 25 MHz Decstation 5000/200 with a turbochannel system bus (100 MB/s) running the Mach 3.0 operating system. It is equipped with two SCSI buses and four 300 MB IBM 0661 "Lightning" drives.

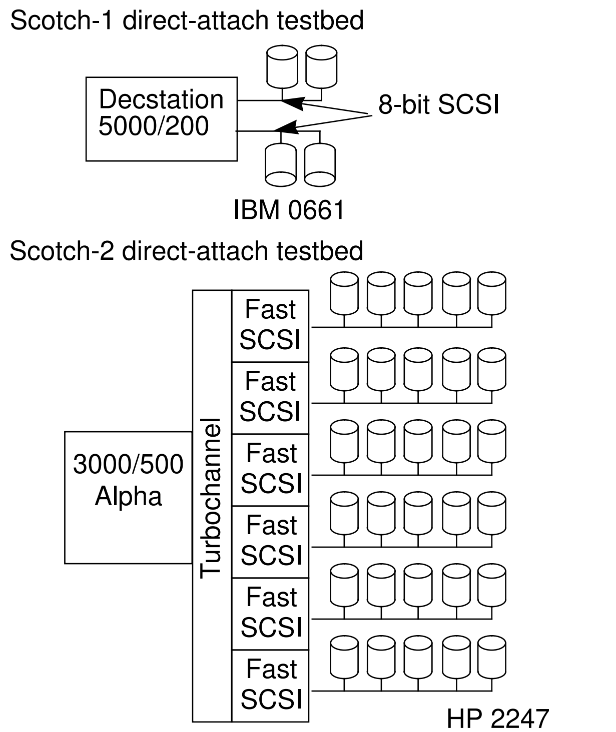

Figure 1: The Scotch direct-attach testbeds.

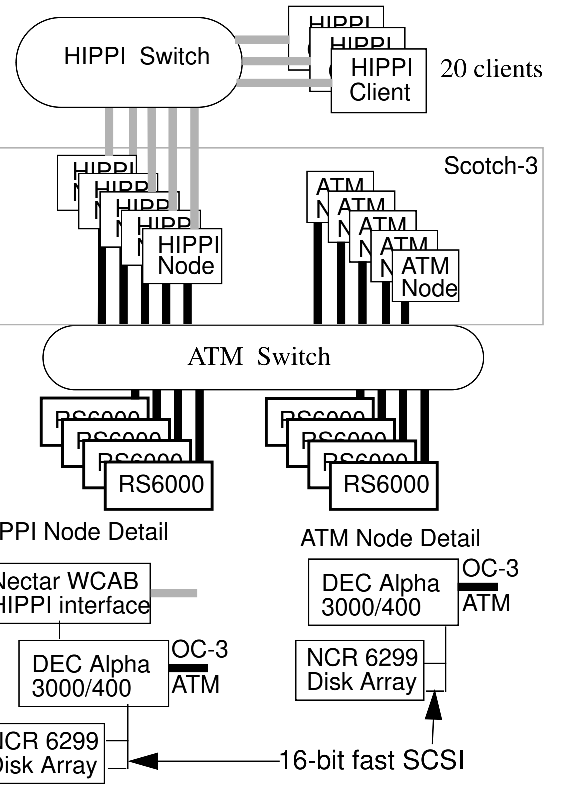

Figure 2: Scotch-3 network-storage testbed.

The second Scotch testbed, Scotch-2, is a larger and faster version of Scotch-1 used for the RAID architecture and implementation research described in Section 3 and for second generation prefetching file system experiments. As Figure 1 shows, Scotch-2 is composed of a 150-Mhz DEC 3000/500 (Alpha) workstation running the OSF/1 operating system and equipped with six fast SCSI bus controllers. Each bus has five HP 2247 drives, giving the total system a capacity of 30 GB.

The third testbed, Scotch-3, is the storage component in a heterogenous multicomputer composed of 38 workstations, 30 DEC 3000 (Alpha) and 8 IBM RS6000 (PowerPC), distributed over switched-HIPPI and OC3 ATM networks. This multicomputer is used for parallel application, parallel programming tool, and multicomputer operating system experiments in addition to the parallel file system research described in Section 4. As shown in Figure 2, Scotch-3 is composed of ten DEC 3000 (Alpha) workstations with turbochannel system buses. Each workstation contains one fast, wide, differential SCSI adapter connected to both controllers of an AT&T (NCR) 6299 disk array. All workstations are interconnected by OC3 (155 Mbit/s) links to a FORE ASX-200 ATM switch complex and five of the workstations are also connected by HIPPI (800 Mbit/s) links to a NSC PS-32 HIPPI switch complex. All storage is available to any node through the Scotch parallel file system and the appropriate routing.

3 Storage Subsystem Research

A crucial factor in the acceptance of RAID has been the ability of storage subsystem providers to provide the RAID advantages of performance, capacity and reliability through existing storage subsystem interfaces such as the SCSI bus and the IBM channel interface. In this section we present research that can be applied without nullifying this advantage. For the sake of brevity, we describe only parity declustering, our most mature RAID architecture. Additional architectures and the work of others is described in a broad survey of RAID research by Chen, Lee, Gibson, Katz, and Patterson [3].

3.1 Architecture Example: Parity Declustering

Fault tolerance and high concurrency make RAID level 5 an attractive storage architecture for transaction processing environments. However, RAID level 5 disk arrays typically experience a 60-80% load increase in the presence of a failed drive. This severe performance degradation limits the applicability of RAID level 5 disk arrays to systems that must be highly available. Further, this failure-mode performance degradation may lead implementors to restrict the fault-free user workload to 50% of the satu-

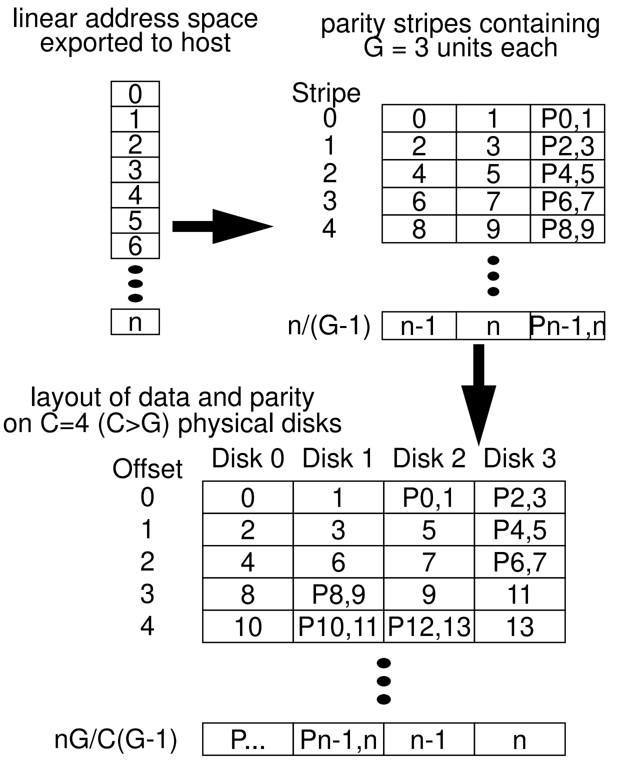

Figure 3: Parity declustered mapping.

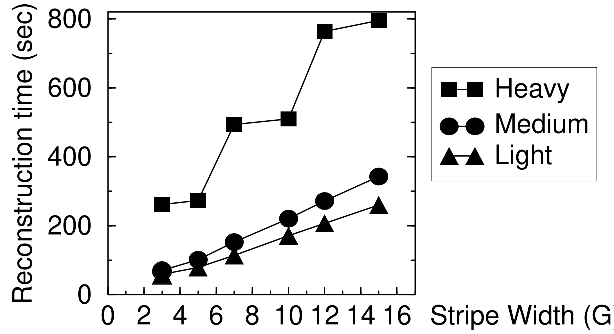

Figure 4: Declustering vs. reconstruction time.

rated load, to avoid overload during on-line failure recovery.

Parity declustering is a variant of RAID level 5 that reduces the performance degradation of on-line failure recovery [4]. The key idea behind parity declustering is that a parity unit protects fewer than *N*-1 data units, where *N* is the number of disks in the array. To achieve this, parity declustering introduces a second layer of mapping between the RAID address spaces and the physical disks (Figure 3) .

We have implemented parity declustering in the Scotch-2 parallel storage testbed. Figure 4 shows the time measured for the reconstruction of the first 200 MB of a disk in a 15-disk declustered array under three workload intensities [5]. As the width of the logical array decreases, both the amount of I/O and computation required for reconstruction drops, allowing reconstruction time to approach the minimum possible — the time to sequentially write 200 MB to the replacement drive.

3.2 Rapid Prototyping with RAIDframe

Design and evaluation of novel RAID architectures such as parity declustering is typically done by custom, design-specific simulation. To achieve more compelling evaluation of competitive or interacting storage architectures, more designs need to be given concrete implementation. However, concrete implementations are often prohibitively expensive and time-consuming to develop. To level the playing field and enrich the design environment, we are developing a portable, extensible framework, RAIDframe, applicable to both simulation and implementation of novel RAID designs. RAIDframe is currently operational as both a simulator and a user-level software array controller that accesses disks via the UNIX raw-device interface. We use its implementation in the Scotch-2 parallel storage testbed where the measurements of parity declustering reported in Figure 4 were collected.

RAIDframe's key feature is the separation of mapping, operation semantics, concurrency control, and error handling, illustrated in Figure 5. Central to the design of RAIDframe is the use of directed acyclic graphs (DAGs) as a flexible, extensible representation of the semantics of an architecture's operations. Figure 6 exemplifies the DAGs RAIDframe uses to specify its operations. Based on our experience with RAID architectures, these DAGs capture the dependencies, primitives, and optimizations that are the essential differences between RAID architectures.

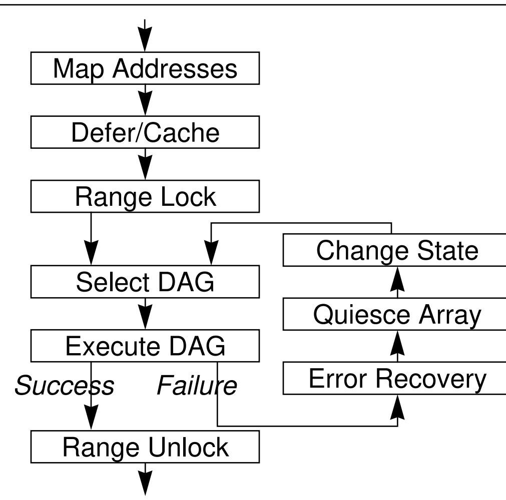

Figure 5: The structure of RAIDframe.

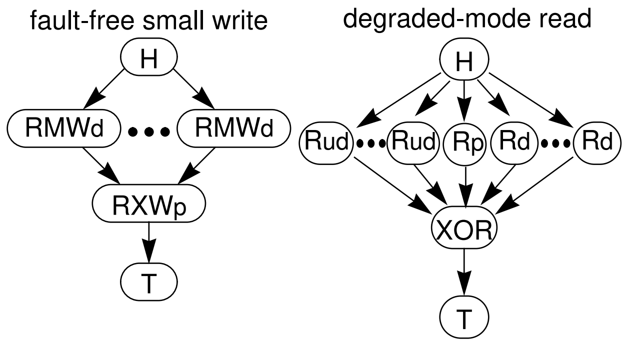

Figure 6: RAIDframe I/O templates.

Nodes labelled “H” and “T” are the header and terminator nodes of the DAG. “R” and “W” nodes invoke disk reads and writes. “XOR” nodes implement an XOR computation over a set of input buffers. An “RMW” node causes the contents of a disk unit to be read into a buffer, and then immediately overwritten with the contents of a second buffer. An “RXW” node causes a disk unit to be read and then immediately overwritten with the XOR of the unit’s contents and other buffers. The subscript “d” identifies a read or write of data that is specifically addressed by the user operation, “ud” identifies data in the stripe that is not being directly addressed by the operation, and “p” identifies parity.

Our first extension of RAID functionality in RAIDframe was the addition of double-failure correction (a P+Q encoding) [5],[6]. Further extensions are underway.

3.3 Automated Error Recovery

Error handling is one of the major sources of complexity in the implementation of a RAID controller [7]. In a non-fault-tolerant system, many errors are handled by discarding all operations in progress and reporting the error for host software to handle. The increasingly complex algorithms which optimize error-free performance in RAIDs have led to an explosion in the size of the state space that must be navigated by error-handling code. Further compounding the problem, RAID implementations often add state-specific performance optimizations to the error-recovery code in a misguided attempt to build a faster RAID. Our approach, consistent with the automated DAG execution in RAIDframe, is to emphasize a separated, mechanized, simple, and robust error-handling system that does not degrade the performance of error-free operation.

In a manner similar to transaction systems, our approach simplifies recovery by eliminating the need for interpretation of incomplete state transitions exposed when an operation fails. However, unlike transaction systems, we do not journal state changes to a log, thereby avoiding the error-free performance penalty associated with logging.

When a DAG fails, we discard it from the system, returning any resources which it may have acquired. After the state of the array has been updated to reflect the fault which caused the error, we initiate a compensating DAG which completes the requested operation.This compensating DAG uses neither data read or computed by the initial method.

The approach is mechanized in RAIDframe by defining a cleanup node for each node of a DAG. A cleanup node releases the resources acquired by its associated node. When an error is detected during forward execution of the graph, we begin working backward through the graph, executing cleanup nodes. When the header node is reached, all resources for the graph have been returned and a compensating method may be initiated. This process is illustrated in Figure 7.

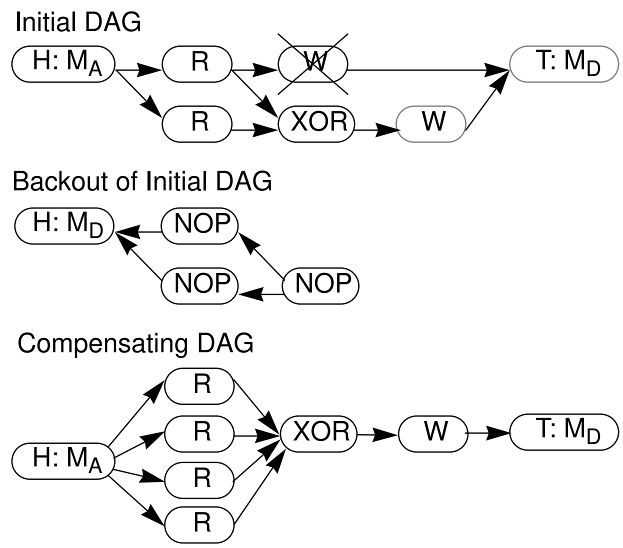

Figure 7: Backward Execution on Error.

In this example, a RAID level 5 small write fails because its data write encounters an error. The XOR of the old data and parity, occurring simultaneously with the failure, is allowed to complete, but the consequential write of new parity is blocked (dashed bubble). When the XOR completes, the DAG is aborted by walking backward through the executed nodes, releasing allocated resources (M_A is memory allocate and M_D is memory deallocate).

After backout of the failed DAG, the system determines that the data write failed because a data drive has failed. A compensating DAG, in this case, a reconstruct-write, is then used to complete the write.

The goal of this approach is to define a minimal set of constraints on the design of error-free DAGs that while allowing a compensating method to not depend on the state of the original DAG at which failure occurred.

4 File Systems Research

In contrast to the RAID subsystem research reported in the previous section, this section reports research embedded in file systems controlling parallel storage.

4.1 Transparent Informed Prefetching

When a workload has many concurrent accesses or consists of huge transfers, parallel storage systems can be immediately employed to achieve increased I/O performance. Unfortunately, many workloads serially issue small or medium-sized I/O requests, presenting little I/O parallelism. For write-intensive workloads, write-behind can be used to batch and parallelize this sequential request stream. However, for read-intensive workloads, the comparable technique, sequential read-ahead, becomes more expensive and less efficient as more parallelism is sought.

Fortunately, many read-intensive applications know in advance the sequence of I/O requests they will make. If applications disclose this advance knowledge, the file system can convert the application's serial request stream into a set of parallel data prefetch accesses.

The performance benefits of exploiting advance knowledge are threefold. First, by exposing parallelism not found in the demand request stream, I/O throughput is increased and application response time decreased. Second, resource decisions, notably buffer-cache management, can be improved by foreknowledge. Third, deep prefetching yields deep disk queues that allow disk scheduling to improve access throughput.

Transparent Informed Prefetching (TIP) is a system we have developed to exploit access-pattern information for read-intensive workloads [8]. Applications are annotated to generate hints that disclose future accesses. The application passes these hints to the buffer cache manager through the file system interface, which then issues prefetch accesses that efficiently utilize the parallel storage system and available system memory.

The TIP system provides applications with portable I/O optimizations. Applications express hints in terms of the existing demand-access interface and thus obtain cross-layer optimizations in a manner consistent with the software engineering principle of modularity. Furthermore, because applications can provide hints without knowing the details of the underlying system configuration, they obtain performance optimizations portable to any machine incorporating a TIP system.

Slice Rendered		Time (Seconds)			
		1 disk	2 disks	3 disks	4 disks
Y-Z 722 blks	no TIP	5.21	5.25	5.17	5.18
	with TIP	5.12	4.27	4.32	4.36
	speedup	1.02	1.23	1.20	1.19
X-Z 722 blks	no TIP	5.86	6.07	6.17	6.36
	with TIP	5.84	4.36	4.43	4.43
	speedup	1.00	1.39	1.39	1.43
X-Y 361 blks	no TIP	8.16	8.40	8.16	8.23
	with TIP	7.86	3.49	2.56	2.23
	speedup	1.04	2.41	3.19	3.69

Figure 8: Visualizing 3-D dataset slices with TIP.

TIP has been implemented in the Scotch-1 direct-attach storage testbed and measured for compilation, text search, and visualization applications [8]. Figure 8 shows the our experience with the 3-D scientific data visualization package, XDataSlice. Originally an in-core rendering tool, we modified XDataSlice to handle datasets too large for memory, in this case 112 MB, by staging data directly from blocked disk files. This blocking is asymmetric, so the X-Y plane contains half as many disk blocks as the other two, to balance approximately the single-disk, non-TIP response time for rendering a slice in each of the Y-Z, X-Z, and X-Y planes. Measurements were taken for each plane both with and without TIP when the dataset was striped over 1, 2, 3, and 4 disks. Speedup is the ratio of the time to fetch a slice's data without TIP to the comparable time with TIP.

Figure 8 shows that XDataSlice cannot exploit a disk array without TIP and that with only one disk, XDataSlice is so I/O-bound that TIP is unable to overlap much computation with I/O. With as little as two disks, however, TIP provides speedups of 1.2 to 2.4, saturating Scotch-1's CPU for the Y-Z and X-Z planes. The X-Y plane continues to benefit from increased disk parallelism, saturating the CPU at four disks with a speedup of 3.7.

While the results of applying TIP in Scotch-1 are promising, this testbed is too slow and small to evaluate many I/O-bound applications. We are in the process of constructing a second implementation of TIP in the Scotch-2 direct-attach storage testbed with emphasis on exploiting application disclosure to make informed cache-management decisions.

4.2 Parallel File Systems

The data sharing needs of network-interconnected workstations are usually provided by a *distributed file system,* in which an individual file is stored on a single server, and the access bandwidth of a single file is limited to that of a single server. Multiple clients simultaneously writing a single file is rare, and is either unsupported or supported with relatively poor performance ([9],[10]). While there may be multiple storage devices in this environment, they are not managed as a parallel storage system.

As the speed of individual client workstations increases, their bandwidth needs cannot be satisfied by a distributed file system. However, their data sharing needs may be met by a *distributed file system with parallel storage*, in which individual files are striped over many storage nodes. This allows a file to be read or written at high bandwidth by a single client ([11],[12]). While simultaneous write access by several clients in these environments remains an unanticipated occurrence, their storage is managed as a unit and may be endowed with RAID functionality.

In many environments, these fast client machines are used for time-consuming computations such as VLSI simulation, weather simulation, and rational drug design [13], whose datasets are often massive (10 MB - 100 GB). With the wide availability of high-level parallel programming tools, such as PVM, high performance FORTRAN, and distributed shared memory (DSM), there is a growing trend to implement each of these applications as a parallel task running on many workstations ([14],[15],[16],[17]). We call a network of workstations used in parallel a *multicomputer* [18].

The multicomputer environment provides new challenges for a distributed file system. The bandwidth and storage capacity requirements are similar to that of a supercomputing environment, but multiple clients concurrently writing a single file are now commonplace. The sharing, fault-tolerance, and scaling challenges of a multicomputer environment are being by the development of *parallel file systems* [19].

We are developing the Scotch Parallel File System (SPFS) for the multicomputer environment shown in Figure 2. It supports concurrent-read and -write sharing within a parallel application and provides scalable bandwidth and customizable availability by striping over independent servers on a file-by-file basis.

SPFS client processes interface directly with SPFS servers through a portable library and the environment's high-performance reliable packet protocol. The SPFS client library includes protocols that coordinate SPFS servers and to provide a single file system image.

SPFS servers are stateless with respect to each other. The pieces of a parallel file that are managed by one server are exported by that server as a single file with the same name as the parallel file. For efficiency, SPFS servers access their file in large blocks through the UNIX raw-device interface. SPFS servers each export a flat namespace, and file access and allocation controls.

SPFS exploits application disclosure of access patterns by integrating informed prefetching. Both SPFS clients and servers use this access pattern knowledge to aggressively prefetch data and defer writes, leading to efficient utilization of servers, network links, and storage devices, and masking the high latencies of networks and disks. SPFS servers additionally utilize informed cache management on manage server memory resources.

SPFS provides redundancy on a per-file basis. This allows applications to choose the level of fault-protection, and the associated overhead cost, on a per-file basis. Because miscomputation of the redundant data encoding only corrupts data the application could already destroy, the per-file redundancy may be computed by SPFS clients on behalf of the application without compromising SPFS integrity. Also, at the application's discretion, redundancy computations can be selectively disabled and enabled to minimize the performance cost of short bursts of rapid changes. This idea, the deferred computation of parity, is called a *paritypoint* by Cormen and Kotz in their requirements for out-of-core algorithms [20].

SPFS is intended to complement rather than replace parallel programming tools such as PVM or DSM by providing high-bandwidth file storage. We expect the generic synchronization needs of applications to be meet by mechanisms provided by these tools. Therefore, SPFS does not provide synchronization primitives such as barriers or locks. However, because SPFS does anticipate file sharing within a parallel application and because it aggressively defers and prefetches, SPFS implements a form of *weakly consistent shared memory* [21].

SPFS exports two primitives, *propagate* and *expunge*, to provide weakly-consistent sharing. Sometime after writing a portion of a shared file, an SPFS client must explicitly propagate that portion to make sure it is visible to other SPFS clients. A sequence of writes without an intervening propagate allows the SPFS client library to coalesce and delay writes. Similarly, an application must explicitly expunge a portion of a shared file to guarantee that its subsequent reads will return the data that has been more recently propagated (exposed) by other clients. A sequence of reads without an intervening expunge allows the SPFS client library to return locally cached data, improving performance.

```
/* Sequentially consistent file system*/

file_handle fh;
int my_start = 2000 * process_number();

loop forever
  fs_read(fh,...);      | reads may span
  computation           | entire file and overlap
  fs_read(fh,...)       |
  computation
  ...
  BARRIER;
  /* write a disjoint section */
  fs_write(fh,...);     | writes restricted to
  computation           | my_start ... my_start+2000
  fs_write(fh,...);     |
  computation
  ....
  BARRIER;
endloop
```

```
/* Weak consistency for SPFS */

spfs_file_handle sfh;
int my_start = 2000 * process_number();

loop forever
  spfs_read(sfh,...);   | reads may span
  computation           | entire file and overlap
  spfs_read(sfh,...);   |
  computation
  ...
  BARRIER;
  /* write a disjoint section */
  spfs_write(sfh,...);  | writes restricted to
  computation           | my_start ... my_start+2000
  spfs_write(sfh,...);  |
  computation
  ....
  spfs_propagate(sfh,my_start,2000);
  BARRIER;
  spfs_expunge(sfh,entire_file);
endloop
```

Figure 9: Data parallelism using SPFS.

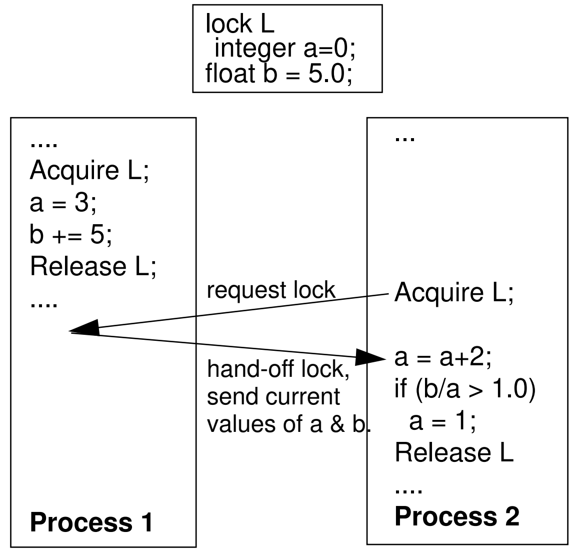

Figure 10: Entry-consistent shared memory.

Entry-consistent programs associate every shared data with a multiple-reader/single-writer lock. Only during a critical section does an entry-consistent system guarantee that a process will obtain valid data. Consequently, entry-consistent programs must communicate shared data only when a lock is acquired, allowing the data transfer to be piggybacked on lock acquisition. This figure shows a fragment of an entry-consistent program. Variables *a* and *b* are guarded by lock *L*. When Process 2 wants to acquire *L*, it determines that Process 1 was the last holder of *L* and requests the lock. Since Process 1 has already released the lock, it immediately responds, sending both the lock and the new values of *a* and *b*. In particular, changing (writing) *a* and *b* does not require the writing process to either broadcast the new value or invalidate other processor's copies of the variable.

Although SPFS does not participate in an application's locking of ranges of a shared file, it offers *expunge* and *propagate* primitives to achieve the consistency provided by acquire and release, respectively.

Figure 9 shows an example of a sequentially consistent single-program multiple-data application modified to allow SPFS to optimize aggressively. After a phase in which all processes read arbitrary sections of the file, each process writes a private section of the file. A barrier naturally occurs between each phase to avoid read/write data hazards. To achieve the proper synchronization in SPFS, the barrier after the write phase is preceded by a propagate (to make the written data visible) and succeeded by an expunge (to discard stale data before entering the read phase).

SPFS's sharing model is close to a DSM model called *entry consistency* [15], illustrated in Figure 10. Expunge and propagate in SPFS are analogous to acquire and release in entry consistency, respectively, but lack the synchronization semantics.

While the largest part of SPFS's implementation is in progress, an early and incomplete version is operational on the Scotch-3 testbed to facilitate application development.

5 Conclusions

The demand for high performance and highly reliable secondary storage systems continues to grow unabated. The Parallel Data Lab at CMU has constructed the Scotch parallel storage systems as testbeds for the development of advanced parallel storage subsystems and file systems for parallel storage.

To advance parallel storage subsystems, we are developing new RAID architectures, an extensible framework for rapidly prototyping RAID architectures, and coding methodologies for simplifying error handling in RAID controllers. RAIDframe, our extensible framework, is operational in Scotch-2 testbed, has demonstrated fast, online reconstruction for RAID levels 5 and 6, and is structured for automatic error handling.

Towards file systems for parallel storage, we are developing prefetching and cache management strategies based on application disclosure and a fault-tolerant network-based parallel file system to support I/O-intensive parallel applications. TIP, our informed prefetching system, has demonstrated a factor of up to 3.7 reduction in execution time for out-of-core visualization on a four-disk array, and is being extended to perform informed cache management. SPFS, our Scotch parallel file system, exploits client-side file management to provide scalability, weak consistency, and per-file configurable availability.

We encourage interested parties to poll our web page, URL http:://www.cs.cmu.edu:8001/Web/Groups/PDL, for further information including the status of these projects and the availability of code. The PDL can also be contacted by electronic mail as *pdl@cs.cmu.edu*.

6 Acknowledgments

The work of the Parallel Data Lab has been supported by many organizations. Equipment has been directly donated by Hewlett-Packard, Digital Equipment Corporation, International Business Machines, NCR (AT&T/GIS) Microelectronics, and Seagate Technology. Scholarship and other support has been provided by Data General, IBM, and NCR. PDL research is also supported by the National Science Foundation through the Data Storage Systems Center, a NSF engineering research center, under grant number ECD-8907068 and the Advanced Research Projects Agency under contract number DABT63-93-C-0054.

7 References

[1] Patterson, D., Gibson, G., Katz, R. "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. ACM Conf. on Management of Data*, 1988, pp. 109-116.

[2] DISK/TREND, Inc. 1994. *1994 DISK/TREND Report: Disk Drive Arrays.* 1925 Landings Drive, Mountain View, Calif., SUM-3.

[3] Chen, P. M., Lee, E. K, Gibson, G. A., Katz, R. H., Patterson, D. A."RAID: High-Performance, Reliable Secondary Storage," *ACM Computing Surveys*, 26(2):145-185, 1994.

[4] Holland, M., Gibson, G. "Parity Declustering for Continuous Operation in Redundant Disk Arrays," *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* pp. 23-25, 1992.

[5] Holland, M.C, Stodolsky, D., Gibson, G. "Parity Declustering and Declustered P+Q in RAIDframe," School of Computer Science, Carnegie Mellon University, Tech Report. Under preparation.

[6] Storage Technology Corporation. *Iceberg 9200 Storage System: Introduction*, STK Part Number 307406101, Storage Technology Corporation, Corporate Technical Publications, 2270 South 88th Street, Louisville, CO 80028

[7] Courtright, W. V. II, Gibson, G. "Backward Error Recovery in Redundant Disk Arrays," *Proc. 1994 Computer Measurement Group Conf.*, pp. 63-74, 1994

[8] Patterson, R. H., Gibson, G. "Exposing I/O Concurrency with Informed Prefetching," *Proc. 3rd Int. Conf. on Parallel & Distributed Information Systems*, pp. 7-16, 1994.

[9] Howard, J. H., Kazar, M. L, et. al. "Scale and Performance in a Distributed File System," *ACM Trans. on Computer Systems*, 6(1):51-81, 1988.

[10] Satyanarayanan, M, Kistler, J. J., Kumar, et. al., "Coda: a Highly available File System for a Distributed Workstation Environment," *IEEE Trans. on Computers*, 39(4): 447-459, 1990.

[11] Hartman, J.H, Ousterhout, J.K. "The Zebra Striped Network File," *Proc. 14th ACM Symp. on Operating Systems Principles*, pp. 29-43, 1994.

[12] Cabrera, L., Long, D. D. E. "Swift: Using Distributed Disk Striping to Provide High I/O Data Rates," *Computing Systems*, 4(4):405-436, 1991

[13] del Rosario, J.M. Choudhary, A.N. "High-performance I/O for Massively Parallel Computers: Problems and Prospects," *Computer*, 27(3):59-68, 1994.

[14] Geist, A., Beguelin, A., Dongarra, J., et. al., *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994, ISBN 0-262-57108-0

[15] Bershad, B. N., Zekauskas, M. J., Sawdon, W. A. "The Midway Distributed Shared Memory System," *Proc. 1993 IEEE Compcon Conf.*, pp. 528-537, 1993.

[16] Zosel, M. E. "High Performance FORTRAN: An Overview," *Proc. 1993 IEEE Compcon Conf.*, pp. 132-6, 1993.

[17] Carter, J. B., Bennett, J., K., Zwaenepoel, W. "Implementation and Performance of Munin," *Proc. 13th ACM Symp. on Operating Systems Principles*, pp. 152-164, 1991.

[18] Kung, H.T., Sansom, R., Schlick, S., et. al., "Network-based Multicomputers: an Emerging Parallel Architecture," *Supercomputing'91*, pp. 664-673, 1991.

[19] Corbett, P.F, Feitelson, D.G. "Design and Implementation of the VESTA Parallel File System," *Proc. Scalable High-Performance Computing Conf.*, pp. 63-70, 1994.

[20] Cormen, T. H., Kotz, D., "Integrating Theory and Practice in Parallel File Systems," *Proc. DAGS/PC Symp.*, pp. 64-74, 1993.

[21] Adve, S. V. Hill, M. D. "A Unified Formalization of Four Shared-Memory Models," *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613-624, 1993.