# Design Considerations for Scalable Parallel File Systems

HERMANN HELLWAGNER

*ZFE STSN 1, Corporate Research and Development, Siemens AG, D-81730 Munich, Germany*

**This paper addresses the problem of providing high-performance disk I/O in massively parallel computers. Resolving the fundamental I/O bottleneck in parallel architectures involves both hardware and software issues. We review previous work on disk arrays and I/O architectures aimed at providing highly parallel disk I/O subsystems. We then focus on the requirements and design of parallel file systems (PFSs) which are responsible to make the parallelism offered by the hardware and a declustered file organization available to application programs. We present the design strategy and key concepts of a general-purpose file system for a parallel computer with scalable distributed shared memory. The principal objectives of the PFS are to fully exploit the parallelism inherent among and within file accesses, and to provide scalable I/O performance. The machine model underlying the design is described, with an emphasis on the innovative architectural features supporting scalability of the shared memory. Starting from a classification of various scenarios of concurrent I/O requests, the features of the PFS design essential for achieving the goals are described and justified. It is argued that the inter- and intra-request parallelism of the I/O load can indeed be effectively exploited and supported by the parallel system resources. Scalability of I/O performance and of the PFS software can be ensured by avoiding serial bottlenecks through the use of the powerful architectural features.**

*Received June 1993; revised October 1993*

## 1. INTRODUCTION

Over the past decade, processor speeds have increased dramatically and main memory sizes have kept pace or have grown at an even faster rate. Storage capacity of secondary memory, i.e. magnetic disks, has grown proportionally with primary memory. In contrast, the performance of conventional disks has improved only modestly during the same period (Katz *et al.*, 1989). These trends are likely to continue in the foreseeable future. The consequence is an increase of the fundamental CPU-I/O performance gap.

Today's parallel systems with hundreds of high-speed processors and large main memory (or memories) offer tremendous raw computing power that will hit the TFLOPS level within the next few years. However, if I/O performance cannot be balanced with computing capabilities, more and more parallel applications will become I/O bound and will not be able to take full advantage of that power. A classical rule of thumb states that a computer system should provide 1 Mbit/s of I/O bandwidth per MIPS of CPU performance (Hennessy and Patterson, 1990). Recent measurements, however, indicate that I/O demands are higher, close to 1 Mbyte/s per CPU MIPS (Akella and Siewiorek, 1991). It is obvious that a conventional disk and file system cannot supply data to the processors at these rates. Exploiting parallelism for I/O in hardware and software is regarded as the only means to overcome the I/O bottleneck in a parallel computer system.

Despite its importance, parallel disk I/O has been devoted adequate research only recently. Work has been performed on three levels:

- Device level (disk arrays).
- Architecture level (organization of disk I/O subsystems).
- Software level (parallel file systems).

In this paper, we will briefly review previous work on the hardware and architecture levels, and then focus on the software level, i.e. parallel file system (PFS) requirements and design.

A PFS is responsible for making the parallelism and raw bandwidth of a distributed I/O subsystem available to applications. It must exploit parallel I/O resources (and other machine resources) to permit or generate parallel I/O transfers. These should translate into high throughput rates and/or low I/O latencies encountered by applications, and result in a balance of computation and I/O and high performance eventually.

Thus, exploiting the parallelism offered by the architecture, on the one hand, and the parallelism available (explicitly or implicitly) in I/O requests, on the other hand, is one of the principal objectives of a PFS. Equally important and closely related to this goal is ensuring scalability of I/O performance. This requires the PFS software to be written in a scalable way, without serial bottlenecks.

These objectives, providing *parallelism* of I/O transfers and ensuring *scalability* of file system software and I/O performance, represent the cornerstones of a PFS design which is the subject of this paper. In addition, the PFS should be suitable for *general-purpose use*. It therefore does not rely on potentially specialized I/O operations,

but aims at exploiting available parallelism in a way transparent to the user.

The machine model underlying the PFS design is a general-purpose, scalable, distributed shared memory (DSM) parallel computer similar to that being described in another paper of this special issue (Abolhassan *et al.*, 1993). The main thrust of machines of this type is that they provide truly scalable shared memory by various innovative architectural means, namely memory randomization, combining of memory accesses in the interconnection network in order to avoid 'hot spots' and multi-threaded processors that efficiently support effective memory latency hiding. Given such features, the PFS serves as a test case for their utility and the programmability of such a machine.

Starting from the I/O load patterns anticipated for such a machine, the major design principles that aim at providing the desired parallelism and scalability properties of the PFS are derived. It turns out that the scalable DSM offers the best opportunities to ensure these features.

The basic design considerations as presented here leave several issues open for further investigations. Hence, we conclude by giving an outlook on further (partly ongoing) work on design refinement, prototype implementation to provide a 'proof of concept' and simulations to evaluate to what degree the objectives have been met.

## 2. RELATED WORK

Parallelism on the path to and from disk devices must be provided and supported on the three levels identified above. In this section, we will briefly review previous work and progress made in these areas. A recent special issue of the *Journal of Parallel and Distributed Computing* (nos 1/2, January/February 1993) provides a broad overview of the field of parallel I/O systems.

### 2.1. Device level

The most promising way to overcome the bandwidth limitation of conventional magnetic disks appears to be 'device parallelization', i.e. to group together multiple physical disks and make them appear as a single logical disk. With such *disk arrays*, single or multiple data blocks (up to entire files) are divided into portions of equal size and spread over successive physical disks of the array. The advantage of disk arrays is that the aggregate bandwidth of multiple disks can be utilized to service single large I/O requests or support multiple independent small requests *in parallel*. As a consequence, the average latency of I/O requests is reduced.

That approach of partitioning data onto multiple disks under the control of a single array controller has become known as *disk striping* (Salem and Garcia-Molina, 1986); often, the term *interleaving* is used equivalently. Operating disk arrays in a synchronous or asynchronous manner has been proposed and studied in terms of its performance effects (Kim, 1986; Kim and Tantawi, 1991). The basic concepts, a taxonomy and the benefits of disk arrays are well described in Katz *et al.* (1989); also, comprehensive descriptions of other techniques for improving performance of conventional disk systems are given.

The key conclusion of that work is that high I/O data rates are best achieved by using a large number of conventional small and cheap disks. Employing multiple disks in parallel, however, substantially increases the probability of disk array failure. Therefore, disk array organizations have been developed that attempt to provide increased fault tolerance by storing redundant information, while maintaining high throughput capabilities and storage capacity. The RAID scheme (*Redundant Arrays of Inexpensive Disks*) has been proposed to encompass different disk array structures with different levels of performance and fault tolerance properties (Patterson *et al.*, 1988; Katz *et al.*, 1989).

RAID systems are now being delivered by most manufacturers. Each of the RAID levels and systems represents a different trade-off between I/O bandwidth, effective storage capacity, and reliability. For the purpose of providing a highly parallel, general-purpose I/O subsystem, a RAID Level 5 system will probably offer the best price/performance ratio (Katz *et al.*, 1989). A disk array of this type therefore appears to be suited as the basic building block of the machine model introduced in Section 3.

### 2.2. Architecture level

The architecture of disk I/O subsystems has not been studied as extensively as disk array structures and, as a consequence, many issues are still open. Most parallel I/O systems built or proposed to date for parallel computers consist of multiple I/O processors, each with one or more disks attached to it. The I/O nodes and the attached devices can be addressed independently, establishing multiple separate I/O units.

In a 'traditional' I/O system, all I/O units are treated in isolation. That is, each file is located (as a whole) on one I/O unit only, which allows for parallel transfers of multiple files, but not to and from a single file. The latter type of parallelism is enabled by a technique termed *file declustering* which breaks up a file into a number of portions and maps them onto different I/O units (e.g. in a round-robin fashion).

Examples of commercial systems of this type are the I/O subsystems of the Intel iPSC/2, iPSC/860 and Paragon computers (Pratt *et al.*, 1989; Intel, 1992) and of the nCUBE machines (del Rosario, 1992). Experimental hypercube systems with concurrent I/O structures have been proposed as well (Flynn and Hadimioglu, 1988; Witkowski *et al.*, 1988; Reddy and Banerjee, 1990).

The advantage of this arrangement of parallel independent I/O units over disk arrays is superior scalability,

at least from a hardware point of view. I/O nodes with disks can readily be added to a system whereas adding disks to an array places an additional burden on the controller and eventually makes it a bottleneck.

Obviously, both these approaches, parallel independent I/O units (file declustering) and disk arrays (data striping), can be combined. Questions that arise are what combination of declustering and striping yields best performance and whether a general guideline for the trade-off in the amount of declustering and striping can be given at all. Studies to answer questions like these have been reported in (Reddy and Banerjee, 1989, 1990), but clear recommendations for I/O subsystem design cannot be derived from the evaluation results.

A general finding probably is that I/O architectures with combined parallelism, i.e. those that use high degrees of both declustering and striping, provide better performance for scientific applications than organizations with lower parallelism. However, other loads such as the transaction workload of those studies appear to benefit more from other structures. Further study is certainly required to derive more useful configuration guidelines.

For general-purpose use, a practical compromise appears to be to use disk arrays of moderate size in the I/O units such that the array controller is not prone to become a bottleneck and array reliability is acceptable, and to determine the degree of declustering (number of parallel I/O units) according to the anticipated I/O traffic of the system. From a scalability point of view, some form of declustering is needed anyway (Reddy and Banerjee, 1989). The Paragon I/O system (Intel, 1992) appears to have been designed along this rationale (incorporating multiple I/O nodes and RAID technology).

However, determining the appropriate number of I/O nodes in relation to the number of compute nodes of a parallel system is a further open problem. Frequently, systems with ratios of I/O nodes to compute nodes of 1:1 to 1:10 are studied or proposed, with non-scientific load (multi-user file system/transaction load) appearing to benefit more from a higher number of I/O nodes (Reddy and Banerjee, 1989; Eckardt, 1991). Interestingly, a ratio of $1:\log p$ is proposed for general-purpose file I/O on hypercube systems, where $p$ is the number of processors (Ghosh *et al.*, 1993). This ratio and the corresponding scalability factor of $p/\log p$ for the I/O subsystem are argued to imply a balanced system. The general belief is, however, that I/O resources are to scale in proportion with the number of processing elements.

### 2.3. Software level

Given a parallel I/O subsystem, the challenge is to make its extensive raw disk bandwidth available for parallel programs. Parallel file system software is needed that maps file I/O requests to the parallel I/O system effectively and efficiently. The PFS must exploit parallelism inherent in the I/O load among and within I/O requests to achieve high overall and individual data transfer rate and minimize I/O latencies.

In order to meet these requirements, a PFS must itself be implemented as a highly parallel program. It should be easy to use, to free the programmer from the burden of dealing with low-level details of partitioning and declustering files to the parallel I/O devices to obtain good performance. In addition, a file system should be scalable in the sense that, when I/O subsystem resources are scaled up, the I/O performance delivered to application programs should scale up proportionally.

Several parallel file systems have been proposed and implemented, predominantly on distributed-memory systems. Commercial systems have become available for iPSC/2 and iPSC/860 and nCUBE/2 hypercube machines.

The well-known Intel *Concurrent File System (CFS)*, for example, declusters files over I/O nodes (Pierce, 1989). On each I/O node, a disk process executes to service read and write requests directed to that node; it controls local file allocation and file structure maintenance, and maintains a buffer cache that holds blocks from locally attached disks. On the compute nodes, read or write operations issued by clients are split up into separate requests for file blocks by CFS runtime library routines. Using file structure and distribution information cached locally, the library sends these requests directly to the appropriate disk nodes. This software structure allows much parallelism of the I/O architecture to be exploited.

However, as pointed out in Choudhary (1993), existing systems are limited due to the requirement on users to perform low-level management of file accesses to obtain high performance. Also, scalability appears to be limited in some systems, e.g. the Intel CFS and the experimental file system *Bridge* (Dibble *et al.*, 1988; Dibble and Scott, 1989), which include centralized server components.

A major issue in existing file systems, in particular for distributed-memory machines, is mapping the distribution of data in main memories and the distribution of data on secondary storage devices to each other, such that truly parallel I/O transfers can take place. Users ought to be relieved from this data mapping requirement as far as possible.

An interesting contribution in that respect in nCUBE's new I/O system software (DeBenedictis and del Rosario, 1992; del Rosario, 1992). A programmer must define or use a function to map data from an application-specific distribution to an intermediate representation only. This is taken by the PFS software to obtain a highly declustered file structure. The combined mapping then defines the data flow between primary and secondary storage. This two-level approach is designed to allow convenient definition of file I/O in a distributed-memory system (without the demand on the user to have to deal with

the I/O architecture), enable parallel I/O transfers, and ensure performance scalability.

It must be noted here that, with shared memory as being available in the machine model underlying our PFS design, data mapping is no longer a critical issue; it suffices to specify the address and length of a main memory buffer where file data have to be transferred to or from.

Other work has investigated special PFS functionality and interfaces. Special I/O modes, file organization concepts, and file system operations have been introduced (Witkowski, 1988; Asbury and Scott, 1989; Crockett, 1989; Pierce, 1989; Kotz, 1992), but a conclusive and generally useful view of file system functionality has not emerged. In addition, many of the schemes are tailored towards scientific applications which are dominated by various forms of sequential access patterns.

Finally, caching in PFSs, in particular prefetching issues and write-back policies, have been studied. All these techniques represent problems and opportunities specific to multiprocessors. Trade-offs and results have been presented in Kotz and Ellis (1990, 1993).

In summary, a number of issues of PFS software design are still open. Our PFS design addresses the problems of ensuring scalability and exploiting parallelism in a transparent way.

## 3. PARALLEL MACHINE MODEL

### 3.1. Architectural overview

The parallel machine assumed for our PFS design is a scalable architecture designed for general-purpose use. In the development of massively parallel architectures, a trend is currently observed towards systems providing a single global physical address space. Such a shared memory is generally held to be a better target for convenient parallel programming, automatic parallelization, load balancing, debugging, and portability of parallel software. A *scalable* shared memory system can only be built as a DSM (Hellwagner, 1990; Nitzberg and Lo, 1991) and efficiency requirements demand that the DSM be implemented in hardware.

The architecture underlying the PFS design is, therefore, a parallel MIMD system with DSM which should scale into the range of massive parallelism, i.e. with thousands of processing elements (PEs). The architecture is depicted in Figure 1.

Each PE is a standard MIMD node with its own independent CPU, local memory and a portion of the global memory (a memory unit or module) attached. These PEs are (virtually) fully connected via a global interconnection network. In our model, this network can be viewed as a black box with no specified physical structure. It should, however, meet certain requirements on the universal scalability of throughput and delay with system size, which are satisfied by network structures with $O(\log p)$ diameter and $O(p \cdot \log p)$ link capacity together with a routing scheme which properly

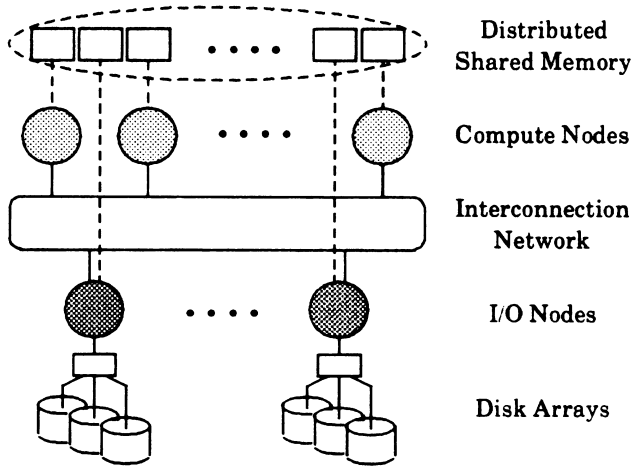


**FIGURE 1.** Parallel machine model.

distributes all kinds of traffic across the entire network. Network structures and routing schemes which meet these conditions have been presented in Klein (1991); a paper in this special issue discusses such interconnect technology as well (Thompson, 1993). The network is supposed to provide both types of services, direct message-passing between processes located on different nodes and remote memory accesses demanded by the DSM subsystem.

### 3.2. Disk I/O subsystem

Among the PEs, we distinguish between compute nodes and I/O nodes (or disk nodes), the latter constituting the I/O subsystem. Attached to each I/O node are one or more disks, preferably organized as a redundant disk array, e.g. according to RAID Level 5. The disk array controller has DMA capabilities to transfer disk blocks to and from the I/O node's local memory. The disk array is viewed by the I/O processor as a single storage device with known capacity and access characteristics.

In this paper, we do not assume or require a specific number of I/O nodes or a specific ratio of I/O nodes to compute nodes. In any case, even if the number of disk nodes is an order of magnitude lower than the number of compute nodes, the I/O subsystem comprises hundreds of nodes and probably thousands of disks. We do, however, assume, as an essential requirement for scalable performance, that the number of disk nodes grows proportionally with overall system size.

The distributed nature of both the I/O subsystem and the shared memory (DSM) offer the chance to perform highly parallel data transfers between primary and secondary memory. From a hardware point of view, there need not be any bottleneck in the data transfer paths. It is up to the PFS to fully exploit this opportunity, to the benefit of parallel programs.

### 3.3. Scalable DSM

Implementation of a scalable shared memory has long been deemed infeasible. However, the DSM concept

offers new opportunities to achieve this goal. Yet, a major impediment to scalability remains even for a DSM system, i.e. increasing communications latencies for remote accesses when system size grows.

There are two principal approaches to deal with this problem:

- *Latency reduction* which attempts to minimize the number of remote accesses and, hence, of average memory access times through caching. Its effectivity heavily depends on the degree of locality exhibited by parallel applications, and on the efficiency of coherence maintenance protocols.
- *Latency hiding* which aims at exploiting the latency associated with a remote memory access in that the issuing processor switches over to execute another process. This approach is effective only if sufficient parallelism (in terms of number of processes) is made available by the software, and the hardware supports efficient process switching.

The first approach is incorporated in systems like the KSR1 (Rothnie, 1992), the DASH machine (Leonski *et al.*, 1992), the Data Diffusion Machine (Hagersten *et al.*, 1992) and the Scalable Coherent Interface (SCI) Standard (IEEE, 1992).

The second approach is of particular interest because it has a thorough theoretical background. Results of (Valiant, 1990a,b) constructively show that shared memory can be emulated in a system with distributed memory units in a scalable way, i.e. with asymptotically constant efficiency, using latency hiding and excess parallelism (O(log $p$) processes per node).

The global memory system we assume is therefore based on this second approach. It incorporates a number of innovative architectural means designed to overcome the remaining threats to scalability. These features are depicted in Figure 2 and described in the following. (Note that the 'dance-hall' configuration of Figure 2 gives a logical view of the memory system. Physically, the memory units comprising the DSM are associated with processors, as shown in Figure 1.)

- *Memory randomization* (address hashing, Randomized Shared Memory (RSM)). If the address space were mapped to memory modules in a regular fashion (e.g. consecutively), regular (e.g. consecutive) access patterns could overload individual memory units and lead to systematic blocking in network switches leading to those modules. Thus, many accesses would encounter increased latency penalties. As a means to obtain an equal loading of all memory modules and distribute memory access traffic uniformly in the network, address hashing has been proposed. This technique pseudo-randomly distributes global memory addresses and, at runtime, global data accesses throughout available memory units (Valiant, 1990a). Recent investigations of the practical efficiency of this RSM scheme have shown encouraging results (Hellwagner, 1992).

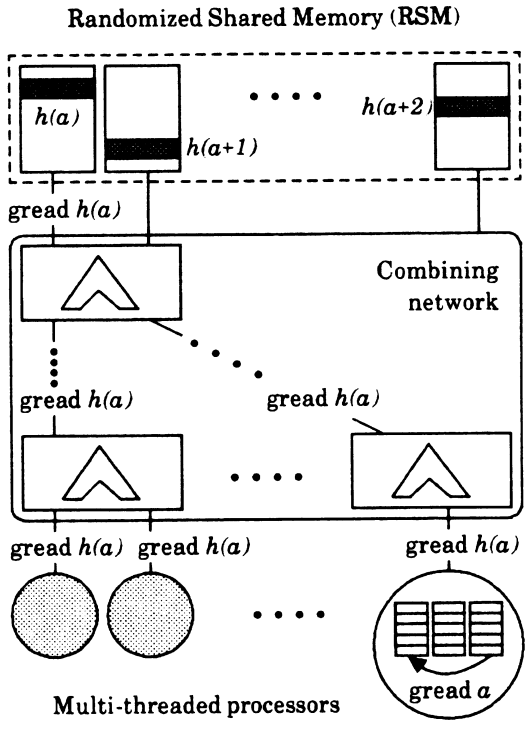


**FIGURE 2.** Scalable DSM implementation.

- *Memory request combining.* Even with RSM, high access latencies can still result from many processes concurrently accessing the same memory location, e.g. a synchronization variable. Such 'hot spots' are avoided in that accesses heading to the same memory location are combined in the interconnect (Gottlieb *et al.*, 1983).
- Fetch&add *memory transactions.* Many algorithms, in particular synchronization algorithms, require or benefit from atomic read-modify-write memory operations. In our memory system, an indivisible fetch&add memory transaction (Gottlieb *et al.*, 1983) is assumed. Concurrent fetch&add operations can be combined in the network.
- *Multi-threaded processors.* As stated above, the processor must support efficient process switching for purposes of effective latency hiding. We assume multi-threaded CPUs that have a number of hardware contexts (register sets, also termed 'virtual processors') which can be switched without delays, i.e. within a cycle. In addition, the contexts can be swapped in from (out to) local memory within a few processor cycles. Finally, such a processor also supports efficient creation and deletion of processes, e.g. directly in its instruction set.

For optimization purposes, it is advantageous to permit explicit control of the allocation of certain shared memory regions to specific nodes (localized pages), and our concepts make use of these features. Similar func-

tionality is provided in the Tera computer system (Alverson *et al.*, 1990).

While such a system may appear 'exotic', it is not unrealistic or unimplementable. Rather, several projects are underway to develop and implement scalable DSM machines or components thereof according to these principles. The Inmos Chameleon project and the PRAM being developed at Universität des Saarlandes (Abolhassan *et al.*, 1991, 1994) encompass all the architectural features listed above and correspond most closely to our machine model. A commercial endeavour using memory randomization is the Tera computer (Alverson *et al.*, 1990). Earlier systems based on such principles are the NYU Ultracomputer (Gottlieb *et al.*, 1984), the Fluent machine (Ranade *et al.*, 1988) and the BBN Monarch computer design (Rettberg *et al.*, 1990).

As indicated earlier, it is decisive for the success and efficiency of machines with RSM that sufficient parallelism is available to realize effective latency hiding. In this sense, our PFS design serves as a case study of whether this kind of system software can adequately contribute to the required degree of parallelism. (Note that other contributions come from other system software and from application processes.)

## 4. I/O LOAD MODEL

In addition to the architecture model, a basic understanding of the patterns of I/O requests a general-purpose PFS will be faced with, must be developed. The I/O load model from which the basic design principles of our PFS have been derived, is described in this section.

### 4.1. Situation

As pointed out in (Choudhary, 1993), the I/O requirements of parallel applications differ widely. Well-known workloads are transaction processing which is characterized by a large number of random I/O accesses of small size, and scientific applications which require fewer accesses, each of larger size and often involving some form of sequential pattern (French *et al.*, 1993). Image processing programs perform even fewer accesses, but of very large size. Multimedia applications potentially require enormous bandwidth and storage capacity when accessing video data. Other types of I/O requirements result from support for virtual memory, checkpointing, and storage of data for off-line visualization; still others from decision support systems.

In contrast to uniprocessors and distributed systems, file access patterns in parallel systems are far from well-understood. This situation is attributed in Kotz and Ellis (1990) to the deficiencies of existing PFSs which have discouraged typical parallel file access patterns to develop. Further, no studies have been carried out to collect data on file usage of existing parallel applications (or they have not been published).

Hence, one has to resort to building models of parallel file I/O patterns based on experience and knowledge or intuitive understanding of parallel applications. Such models have been developed (see e.g. Kotz and Ellis, 1990, 1993; French, 1993; Ghosh *et al.*, 1993), but they either concentrate on scientific workload (sequential accesses) or are derived from distributed-memory systems only.

The model of I/O loads for our general-purpose parallel DSM machine and PFS needs to be more general. We classify I/O requests of parallel applications according to the degree of cooperation and data sharing of the constituent processes.

### 4.2. I/O load scenarios

In the following, three general classes of I/O load are introduced. It must be pointed out that the scenarios presented here describe prototypes for important concurrency situations and not the I/O behaviour of real applications which will rather resemble some (dynamically changing) combination of these simple cases. Nevertheless, the load classes represent a useful framework to point out the most important problems occurring in concurrent file I/O and to indicate potential solutions to these problems, as will be done in Section 6.

#### *4.2.1. Independent clients (Figure 3a)*

The simplest concurrent request scenario is one where many client processes issue their own independent sequences of PFS requests. This scenario is typical for distributed systems or for multiprocessor systems with multiple independent applications executing concurrently. Thus, it includes concurrent requests of all types (i.e. directory as well as data transfer operations), possibly on the same set of files, but without systematic sharing and contention on individual files. Some random sharing may still occur, however, and must be properly coordinated by the PFS. For performance and scalability reasons, the *inter-request parallelism* represented by multiple independent requests must be exploited effectively and efficiently by the PFS.

#### *4.2.2. Cooperating clients with transfers from/to disjoint memory regions (Figure 3b)*

A more specific task for a PFS is the situation where the client processes cooperate systematically and probably in a synchronous way, using a common data set managed by the PFS. The resulting request scenario consists of many concurrent transfer requests for disjoint or overlapping parts of the same file, which often occur nearly synchronously in certain program phases. This load type is typical for distributed-memory machines, and the I/O load models referred to earlier (e.g. Kotz and Ellis, 1993; French *et al.*, 1993; Ghosh *et al.*, 1993), can be subsumed into this class. Request patterns of this type result, for instance, from data-parallel programs where the participating processes execute the same program on separate subsets of the data. The PFS should again support *inter-request parallelism* and, in addition,

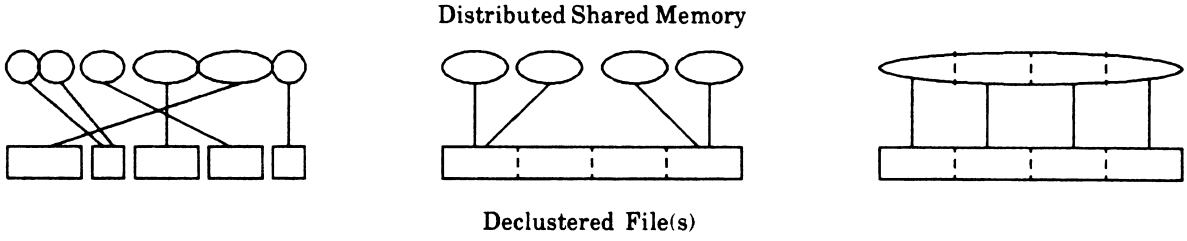


**FIGURE 3.** Classes of parallel I/O loads. (a) Independent clients. (b) Cooperating clients. (c) Single client (large transfers).

allow scalable parallel access to the same file or files, with serialization in case of sharing and contention reduced to what is semantically indispensable.

*4.2.3. Single clients with transfers from/to global memory (Figure 3c)*

With global memory available, a convenient way of doing file I/O is to have a single distinguished process (e.g. a master) request a large data transfer on behalf of all the other processes of the application program (e.g. the slaves). The shared main memory (DSM) data structure from (to) which the data are transferred is produced (consumed) collectively by the application processes, e.g. in a data-parallel way. Note that while I/O in this case is logically centralized, it is not implied that all data must physically pass through the node issuing the I/O request. The challenge for a PFS here is the internal parallelization of the single transfer operation, i.e. exploiting *intra-request parallelism*, in order to support scalable system performance.

### 4.3. Scaling behaviour of I/O loads

In order to eventually determine whether scalability of I/O performance has in fact been achieved, the scaling behaviour of I/O loads must be modelled and the corresponding desired scaling behaviour of I/O performance criteria must be determined. Table 1 presents the basic scaling characteristics for the three load scenarios and the evaluation criteria. It is assumed for this scaling model that overall system size, e.g. number of nodes, main memory capacity, network bandwidth and disk I/O subsystem resources, scale by a factor of $s$. Clearly, variations of this basic scaling behaviour can be considered. However, this model suffices to demonstrate our understanding of scalable I/O performance.

Two entries of Table 1 have to be noted. First, the individual data transfer rate for each request in the cooperating clients load is supposed to remain constant. However, with access conflicts increasing systematically both in number and size, this goal may have to be revised. What can be requested then is that the individual data transfer rate decreases proportionally with the degree of conflicts. Second, note that the individual transfer rate in the single client load is supposed to increase by the overall scaling factor $s$. This requires the PFS to fully exploit intra-request parallelism represented by large transfers.

## 5. PFS FUNCTIONALITY AND STRUCTURE

The functionality and interface of the proposed PFS have been designed to adhere to an approved standard, POSIX.1 (IEEE, 1990), as closely as possible. This standard specifies a conventional read/write file system interface with UNIX-style sharing semantics.

There are certainly more convenient and elegant ways to deal with file I/O, for instance by means of memory-mapped files. However, the POSIX standard is widely adopted and familiar to most users, and has been adopted for almost all PFSs. More importantly, the use of *explicit* requests for file system services (read, write and other operations) greatly simplifies exploiting the inter- and intra-request parallelism inherent in the I/O load. In other words, the conventional interface provides a suitable basis to promote parallelism in file accesses.

As discussed earlier, quite a wide spectrum of specific I/O concepts, operations and modes to support parallel

**TABLE 1.** Basic scaling characteristics of load scenarios and I/O performance criteria when overall system size scales with a factor of $s$

| | *Independent clients* | *Cooperating clients* | *Single client* |
|---|---|---|---|
| Number of clients | $s$-fold | $s$-fold | constant |
| Request rate per client | constant | constant | constant |
| Transfer size per request | constant | constant | $s$-fold |
| Number of files | $s$-fold | constant | constant |
| Size of files | constant | $s$-fold | $s$-fold |
| Access conflicts on shared files and directories | random | systematic, increasing | systematic, constant |
| Response time for single request | constant | constant | constant |
| Individual data transfer rate | constant | constant | $s$-fold |
| Aggregate data transfer rate | $s$-fold | $s$-fold | $s$-fold |

processing have been proposed. However, no consensus on what functionality a PFS should provide, has emerged. For our PFS design, we have refrained from incorporating sophisticated, yet potentially specialized functionality. Rather, we have made the attempt to make the basic mechanisms (standard I/O calls) efficient and flexible to use. This should enable a software layer above the PFS, e.g. a run-time system, to provide richer, application-oriented functionality without great effort.

The POSIX.1 standard has been specified for a uniprocessor environment where concurrent operations, but not really parallel ones occur. Hence the existing standard I/O operations had to be extended appropriately for parallel processing.

The most notable extension relates to the *sharing semantics*, i.e. the rules governing concurrent or parallel accesses to shared files and directories. The PFS enforces *multiple readers/single writer (concurrent read/exclusive write, CREW)* behaviour of such accesses. More specifically, these semantics allow read and write operations to proceed concurrently or simultaneously under the following constraints:

- A write operation blocks read operations, and vice versa, if the scope of the accesses overlaps (i.e. a conflict exists).
- Write operations are atomic with respect to each other and to read operations.
- Conflicting accesses are serialized into arbitrary order.

The CREW semantics apply to accesses to various types of data: directories, PFS management data structures, and files. That means, that in contrast to conventional UNIX systems, for example, multiple simultaneous non-conflicting operations on a single shared file are supported. These semantics are strict and intuitively appealing, similar to UNIX semantics. They demarcate the scope of parallelism in I/O load that the PFS should be able to exploit.

Figure 4 illustrates the basic functional and data layers of the PFS. In Section 6, we will concentrate on the central management level which is the part critical for scalability and for exploiting the parallelism inherent in the I/O load and architecture. It comprises several levels

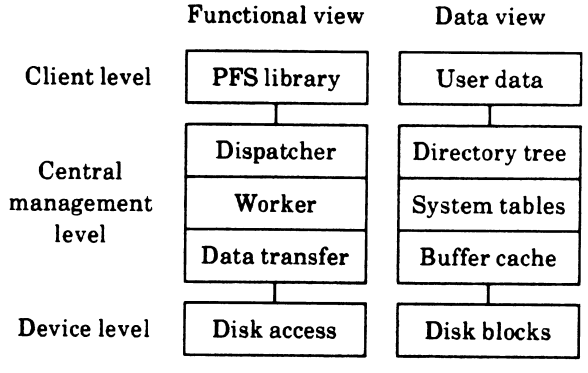


**FIGURE 4.** File system layers.

of server processes and manages the PFS metadata and the buffer cache. The PFS library is linked to the client applications and mediates their I/O requests to the server processes. The client level as well as the device level, which includes the device drivers, are not further dealt with in this paper.

## 6. KEY DESIGN PRINCIPLES OF THE PFS

This section introduces the essential features of the PFS that have been designed to map the parallelism inherent in file and directory I/O to the parallel system described in Section 3. The scope of parallelism is determined by the CREW sharing semantics given in Section 5. Exploiting as much parallelism as semantically possible is the key building block of providing scalable I/O performance. The PFS design concepts are, therefore, introduced and justified primarily in terms of how they promote parallelism of data transfers or how they avoid serial bottlenecks in the PFS software.

We derive these concepts by considering the load scenarios of Section 4 separately. For each load profile, the potential impediments to parallel operation are identified and the appropriate concepts to avoid serial bottlenecks are introduced. This is not to say that each concept introduced is effective for the corresponding load profile only. Rather, structuring the presentation in this manner serves to show how the proposed features quite naturally follow from the potential threats to and opportunities of parallel I/O operation. In practice, just as any real I/O load will be some combination of the 'canonical' load profiles, only the combination of the file system features will allow the I/O load and architectural parallelism to be effectively matched.

### 6.1. Parallelism support for independent requests

Under the independent clients scenario, the file system load consists of many unrelated directory and file accesses. There is no systematic concurrent sharing of files, and conflicting file accesses are assumed to occur at random only. Thus, from a semantic point of view, file accesses may proceed largely concurrently or truly parallel. The challenge for the file system to promote parallel operations under these circumstances is 2-fold:

- Multiple requests must be serviced in parallel. This clearly precludes a centralized server architecture.
- If processed in parallel, multiple requests produce high demand and potential contention on the file system metadata, i.e. the directory tree and internal data structures. Serialization of accesses to the metadata must be avoided as far as possible, under the CREW semantical constraint.

The concepts to overcome the potential threats to parallelism and scalability are outlined subsequently.

### 6.1.1. *Distributed, multi-threaded server*

The basic idea to handle client requests in parallel is to create a light-weight server process (thread) for each request. The concept is illustrated in Figure 5 by means of an example.

The file system library, on behalf of an application process that issues the `open()` call, sends a request message to a PFS *dispatcher* process. This, in turn, spawns a *worker* thread that takes over responsibility to service the request and, in that example, performs pathname translation, sets up or modifies appropriate system table entries, and eventually returns a file descriptor and an error indicator to the client. The dispatcher process is not burdened with that request and is ready to receive further requests and spawn more server threads meanwhile.

Many dispatcher processes are distributed throughout the system such that concurrent client requests can be served truly in parallel. An obvious strategy would be to have one dispatcher process per compute node, but other configurations are equally possible. The worker threads need not necessarily be allocated on the same nodes as their clients. In the first place, performance considerations may suggest so, but load balancing may be equally important. In an RSM-based parallel system, in particular, every node should have a sufficient number of processes to execute such that high memory access latencies can be hidden effectively (excess parallelism). Random placement of file system worker threads, for example, may be employed to support this goal. This issue will be revisited later in this section.

Notice that the multi-threaded server approach contributes to meet the excess parallelism requirement. The prerequisite for this concept to be efficient is that thread creation and switching is cheap, but, as argued earlier, this has to be the case in an RSM-based architecture anyway to make it reasonably efficient.

### 6.1.2. *Management data structures in DSM*

To enable the distributed server threads to operate on a common database, the file system metadata are located in DSM.

The metadata essentially consist of three levels of tables similar to UNIX file system data structures which are termed user file descriptor table, open file description table, and *i*-node table in Tanenbaum (1992). The purpose of the tables in our file system is similar, but there are differences in several details. The single most notable difference is how disk block addresses are stored. This issue is however not relevant to the degree of parallelism achievable. Suffice it to say here that our design allows files up to the terabyte range to be maintained, and that the block addressing scheme is symmetrical. Other differences relate to the locking scheme discussed below.

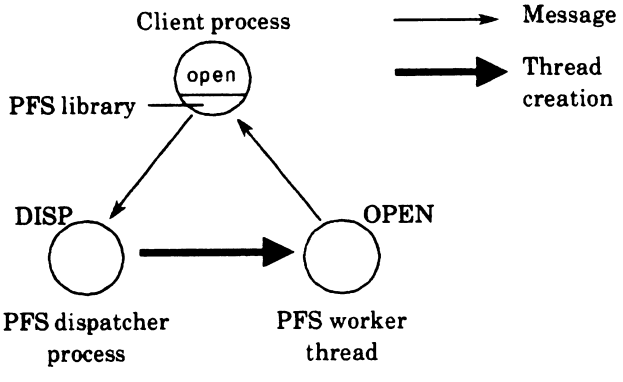


**FIGURE 5.** Creation of server threads.

Logically, the directory tree belongs to the metadata as well, although there are no separate main memory data structures to hold directory information. The directory is stored as a single file on the parallel disks and fetched into memory (in blocks) through the buffer cache mechanism. It is globally accessible in the DSM for the file system worker threads just as the system tables are.

The RSM architecture is most effective for holding such central data structures. While being logically centralized and globally accessible, the metadata are physically distributed in fine grain over the entire system. Thus, the memory and corresponding network load generated by metadata accesses of server threads is smoothed out in the system; the available system resources are utilized in parallel. These properties hold for other DSM schemes as well, but to a lesser extent.

Since RSM is accessible from every node with approximately uniform latency, there is no need to replicate the metadata. Only a single copy of the central data structures is maintained. Thus, coherency problems do not arise in our architecture.

### 6.1.3. *Fine-grained, two-level locking of metadata*

To guarantee consistency of shared data structures in a concurrent or parallel system, temporary locking of critical data (mutual exclusion synchronization) is required. This should be kept to a minimum to enforce serialization only when semantically indispensable.

To achieve this goal, we employ a refined locking scheme characterized by two properties:

- It is *fine-grained* in that only small, but self-contained portions of metadata are protected, e.g. specific directories, and individual directory or system table entries.
- It comprises *two lock types, shared* and *exclusive* locks, which are employed to enforce CREW semantics on common data.

Clearly, as compared with conventional (single-type) locking as used in UNIX file systems, this locking technique requires increased design and implementation effort, and special care to avoid deadlocks and provide fairness conditions, for example. The benefit is, however, that more parallel directory searches, pathname translation, table lookup, and table walks can be performed.

#### 6.1.4. *Scalable* fetch&add*-based readers/writers coordination*

Implementing the two-level (CREW) locking scheme in a scalable and efficient way is an essential prerequisite for the PFS to meet its goals.

Research on such synchronization methods, usually termed readers/writers (R/W) coordination, has been pursued for several years now and brought about sophisticated bottleneck-free R/W coordination algorithms (Gottlieb *et al.*, 1983; Freudenthal and Gottlieb, 1991; Mellor-Crummey and Scott, 1991). There is one drawback to these schemes, however: they are restricted to use busy waiting when processes have to be blocked. Busy waiting wastes machine resources ('virtual processors', network and memory module bandwidth) and, hence, is not a good solution for blocking processes when waiting time is long. This may be the case in the PFS when a protected section involves disk accesses, e.g. for reading in a directory.

Therefore, we have designed a new scalable and flexible R/W synchronization algorithm that improves over previous ones. The algorithm is given in Figure 6 in terms of the entry and exit protocols of reader and writer processes accessing a shared database under the CREW regime.

The key idea of the algorithm is to use a shared counter cnt which keeps track of the current status of contention at the shared database, i.e. the number of readers and writers waiting or accessing the data. Both these numbers can be maintained in a *single* counter in that the counter is incremented by one for each reader and by the value HUGE for each writer. If cnt is a 64-bit quantity and $\text{HUGE} = 2^{32}$, then up to $(2^{32}-1)$ readers and $(2^{32}-1)$ writers may contend for the database. The first activity in the entry (exit) protocols of readers and writers is to atomically increment (decrement) the counter by one or HUGE, respectively, using fetch&add. Thus, with a *single* atomic memory transaction, a process announces (withdraws) its interest, and learns about the interests of other processes. The status returned is used to determine the further protocol actions to be taken. In the conflict-free case, the solution is optimally efficient, requiring one global memory access only to enter or exit from the critical region. In case of contention, the semaphores r and w, accessed by the familiar semaphore operations P() and V(), become effective in controlling and coordinating access to the shared database.

Scalability of this R/W solution is ensured through the use of the fetch&add memory primitive and the combining feature of the target architecture's interconnection network. Concurrent fetch&add accesses to the same memory location are combined 'on-the-fly' in the network, resulting in a single compound request being propagated, and the replies are split up on their way back. The requesters are returned results and the memory contents are changed just as if the requests were executed in some serial order. The combining facility fosters both parallelism and scalability in that such concurrent memory accesses are serviced truly in parallel rather than serially.

The algorithm is flexible in the sense that the semaphore operations P() and V() may be implemented employing either waiting discipline, busy waiting or queueing, without changes to the overall structure of the algorithm. This choice allows the algorithm to be readily adapted to the various instances of the R/W problem in the PFS. Scalable, fetch&add-based implementations of P() and V(), for both busy waiting and queueing, are easily derived from techniques given in Almasi and Gottlieb (1989), e.g. scalable parallel queue management.

### 6.2. Parallelism support for cooperative accesses

As stated in Section 4, the cooperating clients load will produce concurrent and parallel accesses to the same file(s). The load on each shared file will, in general, contain both conflict-free accesses and random and/or systematic access conflicts.

To handle such situations efficiently, the file system should provide the following two features:

- It permits as many parallel accesses on file data as semantically safe.
- It enforces as much coordination (i.e. serialization) as semantically required.

Traditional UNIX file systems are very restrictive in this respect. At any given time, only a single process is permitted access to a file, which is achieved through locking the file's *i*-node. The following feature is destined to establish the above properties in our file system and, thus, to provide substantial improvement over UNIX.

#### 6.2.1. *Record locking on file data*

In the context of a universal, POSIX-style file system, a record is defined as an *arbitrary portion* of a file. Thus, a record may range from a single byte to the entire file.

Associated with each file in our file system design is a table called *record table*. On each access, the file system checks whether the desired file portion is accessible without conflict. If so, the accessing thread may proceed, and the byte range and type of the access are entered into the record table. Again, two types of locks are employed, shared and exclusive locks, for CREW semantics (another instance of the R/W problem in the PFS). When an access is finished, the corresponding entry is removed from the record table.

If an access is detected to be in conflict with one or more others, as is the case for the write accesses $W_2$ and $W_4$ in Figure 7, the accessing thread is blocked until the specified record has been released by the opponent(s). Several strategies for activation of blocked threads can be considered, such as checking activation on every release of a record or at periodic or random time intervals.

Clearly, an efficient implementation of record locking

```
r: sema                          /* Semaphore to grant read access in case of conflicting accesses */
w: sema                          /* Semaphore to grant write access in case of conflicting accesses */

cnt: unsigned integer            /* 64-bit counter to indicate number of readers and writers; */
                                 /* reader increments it by one, writer increments it by HUGE = 2**32 */
nar: integer                     /* Number of active readers; of interest iff writer(s) is (are) */
                                 /* (going to be) blocked and readers have to drain out */

init_sema (r,0)
init_sema (w,0)
cnt := 0
nar := 0

process reader

    /***** Reader's entry protocol: request read access *****/
    if (fetch_and_add(cnt,1) >= HUGE)        /* Writer(s) active or waiting ? */
        P (r)                                /* Wait for writer to grant access */
    .......   /* Read the database */

    /***** Reader's exit protocol: release read access */
    if (fetch_and_add(cnt,-1) > HUGE)        /* Writer(s) waiting ? */
        repeat until (nar>0)                 /* Wait till number of active readers is set by writer */
        if (fetch_and_add(nar,-1)=1)         /* Am I last active reader in this group ? */
            V (w)                            /* Grant access to one waiting writer */

process writer

  oldcnt, nwr: integer                       /* Variables local to each process */

    /***** Writer's entry protocol: request write access *****/
    oldcnt := fetch_and_add(cnt,HUGE)        /* Request access and check R/W status */
    if (oldcnt > 0)                          /* Reader(s) active ? */
        if (oldcnt < HUGE)                   /* Am I first writer to be blocked ? */
            nar := oldcnt                    /* I am first writer; set number of active readers */
        P (w)                                /* Wait for last active reader to grant access */
    .......       /* Write the database */

 /***** Writer's exit protocol: release write access *****/
    oldcnt := fetch_and_add(cnt,-HUGE)       /* Release exclusive access */
    nwr := MOD(oldcnt,HUGE)                  /* Determine number of currently waiting readers */
    if (nwr > 0)
        if (oldcnt > 2*HUGE)                 /* More writer(s) waiting ? */
            nar := nwr                       /* Yes; set number of active readers for next read round */
        par i:=0 for nwr
          V (r)                              /* In parallel, grant access to nwr readers */
    else
      if (oldcnt > HUGE)                     /* Writer(s) waiting ? */
            V (w)                            /* Grant access to next writer */
```

**FIGURE 6.** Scalable R/W coordination algorithm.

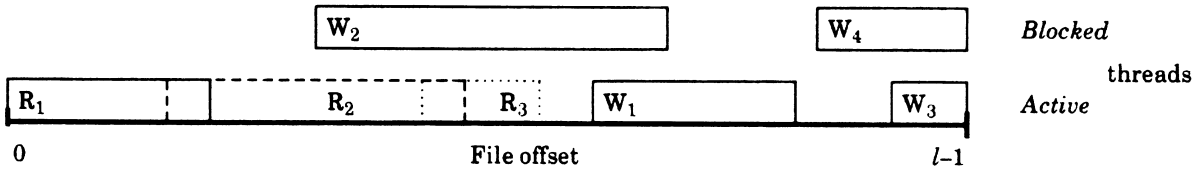


**FIGURE 7.** Conflict-free and conflicting concurrent accesses on file data.

is crucial for file system performance. It is certainly difficult to implement, but it is expected that the parallelism achievable will well outweigh the increased overhead of the record locking protocol.

### 6.3. Parallelism exploitation from large transfers

The third scenario assumes large I/O transfers to and from global memory. A single access to a large file portion does not entail conflicts, but might result in all data being passed through a single node if not forced to exploit the parallel hardware resources. The challenge for the PFS is, therefore, to map the intra-request parallelism inherent in large accesses to the declustered file organization and parallel data transfer paths of the machine.

The following two concepts are proposed for that purpose. The first one, decomposition, is particularly tailored to meet the challenge of large transfers, whereas the second one, the buffer cache, is effective for the other load patterns as well.

#### *6.3.1. Parallel decomposition of large transfers*

The intra-request parallelism of large transfer operations is exploited in that a single access is decomposed into multiple parallel accesses each of which is serviced by a dedicated *data transfer* or *copy* thread. This is depicted in Figure 8 where the data transfer threads are denoted by COPY_RD. These threads are spawned by the worker thread (READ, in this case) and copy data from the disk system (or cache) to the user space. It is proposed to assign transfers of constant size to each of the copy threads, and to have their number vary with the transfer size.

Clearly, the first prerequisite for parallel transfer is that the data are available in parallel from the disk system or cache, i.e. that the file being accessed is declustered over several or all I/O nodes. The file block allocation mechanism has to ensure this property when the file is created or extended.

Second, the copy threads have to be distributed all over the system as well, such that the data transfers can be initiated and controlled in parallel and the memory load and network traffic is equally distributed throughout the system. In principle, the copy threads can be scheduled on compute nodes or disk nodes, based on a deterministic (e.g. round-robin) or random allocation strategy. The important point here is that the process management system supports creation and placement of a possibly very large number of threads in an efficient (parallel) and scalable manner.

This concept provides a number of advantages:

- Utilization of parallel transfer paths, i.e. parallel operation on all architectural levels.
- Support of excess parallelism and load balancing in that multiple processes are generated and distributed for a single I/O operation (given it is sufficiently large).
- Support of scalability through increasing transfer rate for growing system and transfer sizes, induced by an increasing number of copy threads.

#### *6.3.2. Buffer cache in DSM*

For performance reasons, a buffer cache is added to the file system which avoids disk accesses for frequently used disk blocks. Since the file system administers the disk blocks of all I/O processors (IOPs) globally (files are declustered over *all* IOPs), the cache has to be globally accessible as well. Local caches in private IOP memories cannot be used; rather, the cache blocks have to be placed in DSM. This has the advantage that only one instance per cache block has to be maintained which avoids coherency problems. The arguments and benefits are analogous to those for placing file metadata in DSM.

Cache management is performed by the data transfer processes. They search the cache for the requested disk blocks and, in case of a cache hit, transfer data directly between cache and client address space. If a data transfer thread encounters a cache miss, however, it initiates a disk access in that it spawns a *disk access* thread (denoted RD in Figure 8) on the very IOP the requested disk blocks are allocated upon.

The obvious way to organize the cache in DSM is to rely on the standard address randomization scheme to distribute cache blocks and cache administration data structures throughout the entire system. This yields a truly global cache that can be accessed from any node and can hold disk blocks from any IOP in the system in an arbitrary, dynamically changing pattern. This can be effective for relieving an IOP from heavy load, for example, in that (many) more blocks of that IOP are stored in the cache than of other IOPs. There is a significant drawback, however, in that data copying from a disk to the cache (or vice versa) has to take place in two steps: first from the disk controller to a local disk

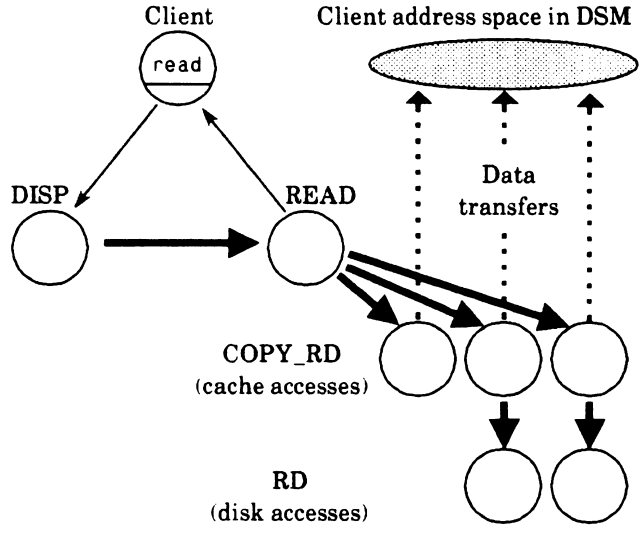


**FIGURE 8.** Decomposition of large data transfers.

buffer (DMA transfer), and second, from this buffer to the distributed cache block (randomization).

The facility of localizing pages on specific nodes, as indicated in Section 3, allows an alternative scheme (and potential optimization) to be considered. The cache can be organized as a collection of independent, local IOP caches, each of which holds blocks from the attached disks only. The local disk blocks and local cache administration data, while still being globally visible and accessible, are allocated in pages localized to the IOP. That is, they are not distributed, and each page is stored contiguously in memory. The main advantage is that disk data can be copied into the local cache area *directly*. This approach avoids the second step of the global cache organization (i.e. additional data copying over the network) and will thus be more efficient in moving data between disks and caches.

Because of this obvious performance advantage and our expectation that 'hot' disks and IOPs should occur rarely in a system storing files in declustered fashion, we currently prefer the organization with many separate IOP caches. One must, however, bear in mind that there is still a trade-off between the efficiency of this organization and the superior caching behaviour of the global cache. The issue will have to be resolved by quantitative evaluation studies.

## 7. CONCLUSION AND FURTHER WORK

We have presented the design strategy and the key concepts of a parallel file system supporting standard POSIX-style file I/O with maximum concurrency in a universal parallel computer equipped with scalable DSM (RSM). Such machines do not yet exist, but are expected for the near future. Besides, many of the proposed concepts will also be effective for DSM schemes with less strict scalability requirements.

The major thrust of our PFS is consistent utilization of the parallelism offered by the underlying architecture and avoidance of serial bottlenecks in the PFS software to ensure scalable I/O performance.

The design principles presented in this paper obviously do not describe a complete PFS design. However, as we have tried to make clear using qualitative reasoning, they are essential prerequisites for the PFS to meet its scalability and performance objectives.

Research on the PFS continues along three lines:

- Design refinement.
- Prototype implementation.
- Quantitative evaluation (simulation).

*Design refinement* in essence involves two activities. First, scalable algorithms and concurrent data structures for the central management level of the PFS (*cf.* Figure 4) are being conceived. An important example are data structures for record locking and corresponding algorithms, as indicated in Section 6; another is detailed design of caching policies. An essential step in that direction has already been completed through the design of the scalable readers/writers coordination algorithm given in Figure 6. Second, decisions to 'dimension' certain data structures have to be taken. This involves determining quantities such as the size(s) of the individual transfer units comprising high-volume data transfers (*cf.* Figure 8) or the size(s) of local buffer caches. This must be supported by quantitative evaluations (modelling and simulation) outlined below.

A *prototype implementation* is underway on a small, bus-based shared-memory multiprocessor that comprises six compute nodes and six disk nodes. The implementation is to demonstrate that the proposed PFS concepts are indeed feasible and implementable with reasonable effort. An example of what can be learned from the prototype is whether the degree of parallelism (in terms of number of processes) required for RSM machines for effective latency hiding, can be generated or adequately supported by file system activities. A peculiar memory architecture makes the multiprocessor well-suited for implementing the PFS concepts: besides physically shared memory, non-coherent DSM with NUMA access characteristics is provided by the hardware. This allows localized shared pages to be defined and used, for example. The implementation uses a proprietary operating system kernel which supports message passing, efficient light-weight process creation and switching (thread management), and simple shared memory management.

Finally, *quantitative evaluations* are performed to serve three purposes. First, since scalability cannot be demonstrated by the prototype implementation, one has to resort to *simulation*. A model of the I/O load, the PFS, and the parallel machine is being developed and simulated to obtain quantitative results on I/O performance and its scalability behaviour. Indirectly, it can be established that the PFS software is free of serial bottlenecks, or impediments to scalability can be identified. Second, the results allow an estimate of the absolute performance of the machine's I/O subsystem to be derived. Third, in the course of developing the overall model, intermediate models are created and simulated in order to quantitatively support the detailed design decisions referred to above (e.g. 'dimensioning' aspects).

## ACKNOWLEDGEMENTS

The parallel file system design is the result of joint work with Horst Eckardt and Axel Klein. Both have also contributed to an earlier version of this paper. The work reported in this paper has been carried out within the cooperative ESPRIT project GPMIMD (General-Purpose MIMD Machines) and has partly been funded by the CEC under EP5404.

### References

Abolhassan, F., Keller, J. and Paul, W. J. (1991) On the cost-effectiveness of PRAMs. In *Proc. 3rd IEEE Symp. on Parallel and Distributed Processing*. IEEE CS Press, Los Alamitos.

Abolhassan, F., Drefenstedt, R., Keller, J., Paul, W. J. and Scheerer, D. (1993) On the physical design of PRAMs. *Comp. J.*, **36**.

Akella, J. and Siewiorek, D. P. (1991) Modeling and measurement of the impact of input/output on system performance. In *Proc. 18th Ann. Int. Symp. on Computer Architecture*. IEEE CS Press, Los Alamitos.

Almasi, G. S. and Gottlieb, A. (1989) *Highly Parallel Computing*. Benjamin/Cummings, Redwood City.

Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A. and Smith, B. (1990) The Tera computer system. In *Proc. 1990 Int. Conf. on Supercomputing*, pp. 1–6. ACM Press, New York.

Asbury, R. K. and Scott, D. S. (1989) Fortran I/O on the iPSC/2. In *Proc. 4th Conf. on Hypercube Concurrent Computers and Applications*, pp. 129–132.

Choudhary, A. (1993) Parallel I/O systems: Guest Editor's Introduction. *J. Parallel Distributed Comput.*

Crockett, T. W. (1989) File concepts for parallel I/O. In *Proc. Supercomputing 1989*, pp. 574–579. ACM Press, New York.

DeBenedictis, E. and del Rosario, J. M. (1992) nCUBE parallel I/O software. In *Proc. 11th Ann. IEEE Int. Phoenix Conf. on Computers and Communications (IPCCC'92)*, pp. 117–124. IEEE CS Press, Los Alamitos.

del Rosario, J. M. (1992) High performance parallel I/O on the nCUBE 2. *IEICE Transactions*. Institute of Electronics, Information and Communication Engineering, Japan.

Dibble, P. C., Scott, M. L. and Ellis, C. S. (1988) Bridge: a high-performance file system for parallel processors. In *Proc. 8th Int. Conf. on Distributed Computer Systems*, pp. 154–161. IEEE CS Press, Los Alamitos.

Dibble, P. C. and Scott, M. L. (1989) Beyond striping: the bridge multiprocessor file system. *Comp. Architecture News*, **17**, 32–39.

Eckardt, H. (1991) *Disk I/O Requirements and Distributed I/O Structures for Parallel Computers*. Internal Report SYS 3-BeG 022/91, Siemens, Munich.

Flynn, R. J. and Hadimioglu, H. (1988) A distributed hypercube file system. In *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications*, pp. 1375–1381. ACM Press, New York.

French, J. C., Pratt, T. W. and Das, M. (1993) Performance measurement of the concurrent file system of the Intel iPSC/2 hypercube. *J. Parallel Distributed Comput.*, **17**, pp. 115–121.

Freudenthal, E. and Gottlieb, A. (1991) Process coordination with fetch-and-increment. In *Proc. Conf. ASPLOS IV*, pp. 260–268. ACM Press, New York.

Ghosh, J., Goveas, K. D. and Draper, J. T. (1993) Performance evaluation of a parallel I/O subsystem for hypercube multicomputers. *J. Parallel Distributed Comput.*, **17**, pp. 90–106.

Gottlieb, A., Lubachevsky, B. D. and Rudolph, L. (1983) Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM TOPLAS*, **5**, 164–189.

Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L. and Snir, M. (1984) The NYU ultracomputer—designing an MIMD shared memory parallel computer. *IEEE Trans. Comp.*, **C-32**, 175–189.

Hagersten, E., Landin, A. and Haridi, S. (1992) DDM—a cache-only memory architecture. *COMPUTER*, **25**, 44–54.

Hellwagner, H. (1990) *A Survey of Virtually Shared Memory Schemes*. SFB Report No. 342/33/90 A, Technical University, Munich.

Hellwagner, H. (1992) On the Practical Efficiency of Randomized Shared Memory. In L. Bougé, M. Cosnard, Y. Robert, and D. Trystram (eds.), *Parallel Processing: Proc. CONPAR 92/VAPP V*, pp. 429–440. Springer-Verlag, Berlin.

Hennessy, J. L. and Patterson, D. A. (1990) *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo.

IEEE (1990) IEEE Std 1003.1–1990/International Standard ISO/IEC 9945–1: 1990. *Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]*. IEEE, New York.

IEEE (1992) IEEE Std 1596–1992. *SCI—Scalable Coherent Interface*. IEEE, New York.

Intel Corp. Supercomputer Systems Division (1992) *Paragon™ XP/S Product Information*. Intel, Beaverton.

Katz, R. H., Gibson, G. A. and Patterson, D. A. (1989) Disk system architectures for high performance computing. *Proc. IEEE*, **77**, 1842–1858.

Kim, M. Y. (1986) Synchronized disk interleaving. *IEEE Trans. Comp.*, **C-35**, 978–988.

Kim, M. Y. and Tantawi, A. N. (1991) Asynchronous disk interleaving: approximating access delays. *IEEE Trans. Comp.*, **C-40**, 801–810.

Klein, A. (1991) Interconnection networks for universal message-passing systems. In *Proc. ESPRITConf. '91*, pp. 336–351. CEC, Brussels.

Kotz, D. (1992) *Multiprocessor File System Interfaces*. Technical Report PCS-TR92–179, Dartmouth College, Hanover, NH.

Kotz, D. F. and Ellis, C. S. (1990) Prefetching in file systems for MIMD multiprocessors. *IEEE Trans. Parallel Distributed Systems*, **1**, 218–230.

Kotz, D. F. and Ellis, C. S. (1993) Caching and writeback policies in parallel file systems. *J. Parallel Distributed Comput.*, **17**, pp. 140–145.

Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Henessy, J., Horowitz, M. and Lam, M. S. (1992) The Stanford DASH multiprocessor. *Computer*, **25**, 63–79.

Mellor-Crummey, J. M. and Scott, M. L. (1991) Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proc. 3rd ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pp. 106–113. ACM Press, New York.

Nitzberg, B. and Lo, V. (1991) Distributed shared memory: a survey of issues and algorithms. *Computer*, **24**, 52–60.

Patterson, D. A., Gibson, G. and Katz, R. H. (1988) A case for redundant arrays of inexpensive disks (RAID). In *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 109–116. ACM Press, New York.

Pierce, P. (1989) A concurrent file system for a highly parallel mass storage subsystem. In *Proc. 4th Conf. on Hypercube Concurrent Computers and Applications*, pp. 155–160. ACM Press, New York.

Pratt, T. W., French, J. C., Dickens, P. M. and Janet Jr., S. A. (1989) A comparison of the architecture and performance of two parallel file systems. In *Proc. 4th Conf. on Hypercube Concurrent Computers and Applications*, pp. 161–166. ACM Press, New York.

Ranade, A. G., Bhatt, S. N. and Johnsson, S. L. (1988) The fluent abstract machine. In Allen, J. and Leighton, F. T. (eds.), *Advanced Research in VLSI (Proc. 5th MIT Conf.)*, pp. 71–93, MIT Press, Cambridge, MA.

Reddy, A. L. N. and Banerjee, P. (1989) An evaluation of multiple-disk I/O systems. *IEEE Trans. Comp.*, **C-38**, 1680–1690.

Reddy, A. L. N. and Banerjee, P. (1990) Design, analysis, and simulation of I/O architectures for hypercube multiprocessors. *IEEE Trans. Parallel Distributed Systems*, **1**, 140–151.

Rettberg, R. D., Crowther, W. R., Carvey, P. P. and Tomlinson, R. S. (1990) The Monarch parallel processor hardware design. *Computer*, **23**, 18–30.

Rothnie, J. (1992) Kendall square research introduction to the

KSR1. In Meuer, H.-W. (ed.), *Supercomputer '92*, pp. 104–114. Springer-Verlag, Berlin.
Salem, K. and Garcia-Molina, H. (1986) Disk striping. In *Proc. IEEE Data Engineering Conf.*, pp. 336–342. IEEE CS Press, Los Alamitos.
Tanenbaum, A. S. (1992) *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, NJ.
Thompson, P. (1993) Concurrent interconnect for parallel systems. *Comp. J.*, **36**.
Valiant, L. G. (1990a) General purpose parallel architectures. In van Leeuwen, J. (ed.), *Handbook of Theoretical Computer Science*. North-Holland, Amsterdam.
Valiant, L. G. (1990b) A bridging model for parallel computation. *Commun. ACM*, **33**, 103–111.
Witkowski, A., Chandrakumar, K. and Macchio, G. (1988) Concurrent I/O system for the hypercube multiprocessor. In *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications*, pp. 1398–1047. ACM Press, New York.