PAULO ORLANDO REIS AFONSO LOPES

A SHARED-DISK PARALLEL CLUSTER FILE SYSTEM
("Um Sistema de Ficheiros Paralelos para Clusters de Disco Partilhado")

Dissertação apresentada para obtenção do
Grau de Doutor em Informática
Pela Universidade Nova de Lisboa,
Faculdade de Ciências e Tecnologia

LISBOA
2009

Agradecimentos

Estas poucas linhas são o corolário de uns bons anos de trabalho, polvilhados aqui e ali por alegrias (e não só) e momentos inesquecíveis; são as últimas a serem escritas, mas nem por isso menos importantes: de facto, são-no de tal forma, que constituem a "abertura" desta dissertação.

Em primeiro lugar, a minha gratidão a dois Professores do DI-FCT/UNL, que me convidaram para esta casa e são responsáveis pela completa mudança que se operou na minha vida – Profs. José Cardoso e Cunha e Pedro Medeiros; e a este último, em especial, um redobrar de agradecimentos por ter aceite a tarefa de me orientar ao longo destes anos. Também quero aproveitar para agradecer aos colegas que aceitaram distribuições assimétricas nas suas cargas de trabalho docente, permitindo que as minhas tarefas de docência se concentrassem e, nos outros períodos, me pudesse focar quase exclusivamente neste trabalho.

Duas empresas contribuíram decisivamente com os seus donativos para a realização deste trabalho, que se iniciou numa infra-estrutura doada pela Lusitania, Companhia de Seguros, S.A. e, prosseguiu numa outra obtida através de uma candidatura bem sucedida ao programa IBM Shared University Research (SUR). Os meus agradecimentos a ambas, e em especial a Teresa Moradias, António Jorge Matos e Luís Esteves, da Lusitania CS; e a Filipa Valente e Luís Diniz dos Santos, da IBM Portugal.

Foram vários os amigos e colegas a trabalhar em organismos e empresas que utilizam, ou são potenciais utilizadores dos paradigmas e/ou tecnologias aqui abordados, bem como outras que fazem destas o seu *core business*, e com os quais houve uma profícua troca de ideias e de experiências; destes gostaria de destacar, em especial Carlos Corredoura, da EMC Portugal.

E, por último, à Lili, ao Pedro e à Ana Sofia...

Resumo

O uso coordenado de múltiplos nós de computação (*clusters*) como plataforma para resolver, em ambientes de cálculo de elevado desempenho (HPC), problemas de grande exigência computacional, ou para oferecer, em ambientes de Sistemas de Informação (SI), serviços fiáveis e tempos de resposta adequados é hoje uma solução indiscutível, em termos de custo/benefício.

Os ambientes de HPC e SI são razoavelmente dissimilares, particularmente no que se refere a sistemas de ficheiros e as arquitecturas de armazenamento; em "ambientes HPC", favorece--se o uso de sistemas de ficheiros de elevado desempenho, em detrimento de outras características (não são, geralmente, compatíveis POSIX), e usam-se discos internos ou privados; em "ambientes SI", preferem-se soluções de alta disponibilidade suportadas em armazenamento externo e, quando tal se revela necessário, sistemas de ficheiros para discos partilhados (CFS), desde que compatíveis POSIX (mesmo sacrificando o desempenho).

O parallel Cluster File System (pCFS) é a nossa proposta para mudar este estado de coisas, usando o melhor de cada um: a fiabilidade dos CFSs e o excelente desempenho dos sistemas de ficheiros paralelos. Não se pretende conseguir máximos absolutos, mas tão somente uma compatibilidade total com a norma POSIX, versatilidade, e níveis de fiabilidade e desempenho suficientemente bons para uma utilização genérica – aplicações tradicionais e HPC, suporte de motores DBMS que armazenem dados em ficheiros, e *streaming* de vídeo. As ideias-chave para o pCFS são:

- *Caching* cooperativo, uma técnica usada em sistemas de ficheiros para discos distribuídos que, tanto quanto sabemos, nunca foi usada em CFSs em SAN ou sistemas de ficheiros paralelos. Resulta daqui que o pCFS pode usar todas as infra-estruturas (LAN e SAN) para aceder a dados.
- *Locking* de granularidade fina, que permite definir regiões disjuntas (ao nível do byte) num ficheiro podendo os processos, mesmo quando correm em nós distintos, nele ler e escrever em paralelo, à velocidade da infra-estrutura SAN (desde que não ocorram mudanças importantes na estrutura dos metadados).

Construímos um protótipo sobre o GFS (um CFS da Red Hat), modificando ligeiramente o módulo GFS, acrescentando-lhe dois módulos de sistema suplementares, e ainda um terceiro, de nível utilizador. No protótipo, o *locking* de grão fino está integralmente realizado e a *cache* global é mantida coerente com transferências de fragmentos de páginas realizadas sobre LAN.

Os testes efectuados para o caso de processos que correm em diferentes nós escrevendo sobre um mesmo ficheiro mostram que o pCFS tem um desempenho idêntico ao do Parallel Virtual File System (PVFS) e duas vezes superior ao do NFS, consumindo muito menos CPU que estes (cerca de 10 vezes); e que, quando comparado com o GFS, tem desempenhos que são 2 a 600 vezes superiores (para acessos de 4 MB e 4 KB, respectivamente) com idênticos consumos de CPU.

Abstract

Today, clusters are the de facto cost effective platform both for high performance computing (HPC) as well as IT environments. HPC and IT are quite different environments and differences include, among others, their choices on file systems and storage: HPC favours parallel file systems geared towards maximum I/O bandwidth, but which are not fully POSIX--compliant and were devised to run on top of (fault prone) partitioned storage; conversely, IT data centres favour both external disk arrays (to provide highly available storage) and POSIX compliant file systems, (either general purpose or shared-disk cluster file systems, CFSs). These specialised file systems do perform very well in their target environments provided that applications do not require some lateral features, e.g., no file locking on parallel file systems, and no high performance writes over cluster-wide shared files on CFSs. In brief, we can say that none of the above approaches solves the problem of providing high levels of reliability and performance to both worlds.

Our pCFS proposal makes a contribution to change this situation: the rationale is to take advantage on the best of both – the reliability of cluster file systems and the high performance of parallel file systems. We don't claim to provide the absolute best of each, but we aim at full POSIX compliance, a rich feature set, and levels of reliability and performance good enough for broad usage – e.g., traditional as well as HPC applications, support of clustered DBMS engines that may run over regular files, and video streaming. pCFS' main ideas include:

- Cooperative caching, a technique that has been used in file systems for distributed disks but, as far as we know, was never used either in SAN based cluster file systems or in parallel file systems. As a result, pCFS may use all infrastructures (LAN and SAN) to move data.
- Fine-grain locking, whereby processes running across distinct nodes may define non-overlapping byte-range regions in a file (instead of the whole file) and access them in parallel, reading and writing over those regions at the infrastructure's full speed (provided that no major metadata changes are required).

A prototype was built on top of GFS (a Red Hat shared disk CFS): GFS' kernel code was slightly modified, and two kernel modules and a user-level daemon were added. In the prototype, fine grain locking is fully implemented and a cluster-wide coherent cache is maintained through data (page fragments) movement over the LAN.

Our benchmarks for non-overlapping writers over a single file shared among processes running on different nodes show that pCFS' bandwidth is 2 times greater than NFS' while being comparable to that of the Parallel Virtual File System (PVFS), both requiring about 10 times more CPU. And pCFS' bandwidth also surpasses GFS' (600 times for small record sizes, e.g., 4 KB, decreasing down to 2 times for large record sizes, e.g., 4 MB), at about the same CPU usage.

Table of Contents

List of Figures

List of Tables

Part I:

Motivation and Background

In this Part we present the motivations for our work, along with a small introduction that covers the transition from the supercomputer architecture to clusters; we include a brief overview of applications that need high performance I/O, including scientific, database, and multimedia, and lay out the focus and major contributions we anticipate from our dissertation.

1 Introduction

1.1 The evolution of high performance computing architectures

In the last two decades, two factors were determinant in the architectural shift that made high performance computing (HPC) available to broader audiences: technological, in the form of high speed microprocessors and interconnects; and economical, as their inclusion in the commodity market of personal computers (PCs) resulted in even lower costs. Specialised supercomputer architectures have therefore been replaced by clusters[1] – originally built around piles of commodity PCs, then around small symmetrical shared-memory multi-processor (SMP) nodes made up from common off-the-shelf (COTS) parts, and recently with more "exotic" parts such as multicore processors and blade servers; today, large clusters can reach amazing raw performance figures, as we multiply a node's performance expressed, e.g., in floating-point operations per second (FLOPS), by the total number of nodes.

As clusters replaced Massive Parallel Processors (MPP) and other supercomputer architectures, efforts were carried out to simplify their installation, operation, administration, and everyday use; the aim was, quite understandably, to present a cluster as a single, although large-sized, computer. Efforts were pursued in the programming models arena too, aiming either to simplify programming, as proposed by the shared memory model adopters, or develop programs that extract the last ounce of performance, as advocated by the proponents of message passing programming model.

1.2 Bottlenecks: when reality crushes in

But despite the high performance figures we can get out from clusters (even when these figures are not merely raw values but ones derived from accepted benchmarks such as the LINPACK[2] [Don+01] benchmark), some HPC "real world" applications may run at a much slower speed than what it should be expected, given the system's rating. There may be several reasons why this may happen; to name a few, and drawing a parallel to what happens in the single node world, the application may be CPU bound, memory bound, or I/O bound.

We all know, just by mere observation of advertisements in the industry, the huge increases in processor and interconnect performance that occur, say, every year; so, dealing with a CPU bound application is simple, isn't it? We just need to add another CPU, or replace the current one with a faster model… well, it may help; or else, it may just highlight yet another bottleneck. Anyway, in a multiple node cluster, what should we do? Replace the CPUs in all nodes? Add in some more nodes? The first option is quite cumbersome, but the second is very

[1] For a comprehensive survey of computing architectures see Part II, section 4.
[2] Used to rate the world's top supercomputers in www.top500.org.

practical – although it may be inappropriate for some cases – e.g., when, to use more nodes, either the application itself or some of its parameters have to be changed.

Good observers will also notice that the high rate of improvement in CPUs does not hold in other technologies, e.g., disk, and/or memory; therefore, "slow application" behaviour can also result from memory latency or bandwidth problems, something that may not be so easy to fix as, say, adding more memory to each node when memory is scarce, and cannot for sure be fixed just by increasing the CPU clock.

1.3 The I/O bottlenecks

In this work we are first and foremost interested in systems that can be made to perform I/O at such a rate that it will not hinder the overall progression of the computation, i.e., systems that exhibit good I/O performance, and are scalable.

From a node's perspective, good I/O performance requires[3]: reasonably fast disk devices; a contention-free or slightly-contended I/O infrastructure to connect the devices to their bus adapters; a DMA-capable I/O controller plugged into a high bandwidth, low latency I/O bus; and finally, a good I/O stack – from the device driver to the file system layer.

But these are single-node perspectives and we are interested in clusters where more things should be considered; for example, should each node in the cluster have its own set of private disks, or share disks with other nodes? Should nodes be symmetrical, i.e., should they run the same set of services or, conversely, should some nodes perform one duty only, e.g., I/O storage/server, while others are I/O clients? And, finally, is the configuration (hardware, architecture, software, etc.) scalable, i.e., does resource addition such as nodes and/or disks result in more I/O bandwidth?

1.4 High availability

Today's cluster applications may use large numbers of nodes and run for days, or even months. In this scenario, failure of a component (CPU, memory, interconnect, disk, etc.) is a certainty, so steps must be taken to assure that the application state can be recovered and, as soon as the subsystem containing failed component has been either repaired or taken off, the computation can be restarted. From the I/O perspective this requires a highly available (HA) architecture, covering both hardware and software – e.g., file system. As a counterexample, an I/O architecture where nodes have their own, private, internal disks is not a good choice, as data is no longer accessible when a node fails, whereas an architecture where nodes access external array-based storage [Pat+89] may be able to offer some sort of service continuity.

[3] These aspects will be explored in greater detail in Part II.

2 I/O Intensive Applications

Today, I/O intensive applications are executed in clusters of all sizes; while today's most HPC clusters mimic supercomputers of yore and are usually organised into compute and I/O nodes (see sections 4.3 to 4.5), some are configured in a different way; anyway, nodes that perform I/O tasks do run a distributed file system – no matter how we call it: parallel, cluster, or just plainly distributed[1]. What do applications, running in those clusters, need from the I/O subsystem? How can the operating system (OS), file system, and storage subsystem satisfy their needs?

2.1 Who needs high-performance I/O?

It is common knowledge that most applications that access very large amounts of data need high-performance I/O, and we can find examples in very different fields such as scientific, database and multimedia.

Scientific applications cover domains such as astronomy (galaxy formation), chemistry (molecule synthesis), geophysics (climate, ocean), physics (fluid dynamics), high energy physics (particle accelerators), and medicine (tomography data mining). Common characteristics of "hard" scientific applications are [Nit+95]: they use multidimensional arrays, are not embarrassingly parallel, and are memory and/or CPU bound.

Database applications also benefit a great deal from high-performance I/O; well known examples include online transaction processing (OLTP) applications such as airline reservations and online shopping, online analytical processing (OLAP) applications such as business marketing, sales reporting, data warehousing, and data mining.

Multimedia applications such as video-on-demand require both high bandwidth, to cope with multiple data streams, and good quality of service (QoS), i.e., constant data rate from video servers.

2.2 Scientific applications

Broadly speaking, reasons to perform I/O in scientific applications can be grouped into two categories: *compulsory* and *out-of-core* [Cra+04, Sch+99]. Compulsory, *i.e.*, unavoidable I/O, includes data and parameter input, and output data. We label here as out-of-core all those operations that, while avoidable in principle, are nevertheless convenient to have; examples include checkpointing of partial results, both for debugging and to support application restart (because of failures or to try "something else" – e.g., a different set of parameters – on intermediate results); scratch files; and, finally, true out-of-core support where data overlaying is programmed/controlled explicitly by the application programmer because either

[1] We will propose, in Part IV, a reference model and a taxonomy that will enable us to establish a file system classification and highlight their important aspects.

he/she is using legacy code, or knows that performance will be better than if that same task is carried out by the kernel's virtual memory management.

When performing I/O to a disk file, several transformations may occur; the layout of a data structure, when in-memory, may be quite different from its in-file layout, and that one may also be distinct from the on-disk layout. Furthermore, the developer has to decide either for a data layout that will extract the best performance, or one that will be compatible with a sequential version of the application – quite useful for debugging, at the expense of performance; one may also choose to store it in a portable format, such as the Hierarchical Data Format, HDF [NCSA99] or the Network Common Data Form, NetCDF [Uni06]. User--level libraries provided with these packages do perform those transformations, but they are usually available only for POSIX compliant [IEEE04] file systems.

Therefore, to efficiently support a wide range of existing scientific applications and/or libraries (where some were developed to run on MPP platforms, others were tailored to large SMPs or vector supercomputers) in a cluster with a minimum of modifications to their source code, one must choose a file system that offers POSIX-compatibility (including both the API and the sharing semantics) while still providing high aggregated I/O performance.

2.3 Database applications

The evolution of high performance database servers followed an interesting path, from early shared *memory* architectures to shared *nothing* MPPs, and back to "shared something" in the form of today's clustered shared *disk* SMPs [Nor+96].

The shared disk approach was taken mainly because it successfully solves two major problems: tolerance to failures, and difficulty to find a good partitioning strategy to distribute data among the nodes (and their local disks). This is just another case of trading the theoretical peak power of the MPP approach for the apparently simpler, although theoretically less scalable, "cluster of SMPs" architecture; for simplicity, we mean that life is much easier for database designers/administrators and application programmers; as an example, database administrators have to decide about the placement of the physical database structures into logical disks, and these onto physical ones, taking into account data de-clustering, RAID levels, and multi-path[2] I/O; they don't need to worry on how to partition data among servers.

Still, for the DBMS engine implementer, the main difficulties for implementing a "Storage module" remain: the mapping of logical database structures, such as tables (in a RDBMS), indexes, hash tables, etc. into physical database structures, such as files in a filesystem, or blocks in *raw disks*[3] (also called raw devices) [Ndi+04].

[2] See Part IV, section 12.1.
[3] Microsoft SQL Server is a major DBMS that does not support raw disks.

To efficiently support a file-based DBMS, instead of a raw-disk based one, even general--purpose local file systems such as ext2 [Bov+05] or NTFS [Nai04] must provide high performance I/O; to run a file-based DBMS in a cluster with a minimum of modifications to the DBMS implementation, the supporting distributed file system must again offer high levels of aggregated I/O performance, while still providing POSIX-compliant file system features, such as "single-node equivalent semantics" (see 11.1.5 and [Sch+02]).

2.4 Multimedia applications

Multimedia environments are yet another example of the distributed client-server paradigm; a typical scenario has users at their terminals (TV, PC, PDA, mobile phone, etc.) choosing (in a process that may involve something as simple as browsing a list of choices, or as elaborate as querying a database) a "rich document", and accessing it (viewing and/or hearing, or even producing/modifying). From end to end, i.e., from the server down to the user's terminal, the whole infrastructure must concurrently support multiple data streams where synchronicity and isochronicity are of paramount importance [Ben+02].

From the I/O point of view, the "source" (servers and disks), which is our main subject of attention, has to cope with these requirements as they impose real-time constraints that must be met throughout all the file system layers, down to the bare disks. A common approach for today's multimedia servers is to have a clustered architecture where nodes have external SAN-attached disk arrays[4] (shared or not) and data is distributed across servers/disks according to some user/file system defined policy. One intuitively expects that the usual policies for reordering disk requests, such as the elevator algorithm [Bov+05], may not be adequate here and that having a richer file system API, one that enables us to communicate the above-mentioned requirements down to the file system layer, seems quite logic.

Finally, processing of streamed data may have to be carried out, *e.g.*, to adapt the stored frames resolution to the user's terminal, something that can be done a) at the server [Ben+02], trading CPU for a decrease in network bandwidth; or, b) at the user's terminal, if that is possible and desirable; or, c) in a middle tier of application-specific services/servers, an architecture we have never seen but seems a reasonable approach, and is probably an interesting research topic.

Thus, to effectively support rich media environments where multiple independent isochronous streams must be fed, a server (parallel) application must be able to specify its QoS requirements regarding sustained data delivery bandwidths for each stream. This seems possible even on top of a general purpose, POSIX-compliant file system, provided that some

[4] Internal disk configurations are not widely used because they constitute single points of failure; Storage Area Networks (SANs) are discussed on Part II, section 5.

minor modifications to the API are introduced, e.g., extending the range of options available to some calls such as `open()` and `ioctl()`.

2.5 High performance I/O for all: the case for shared disk storage

Disk arrays are today's ubiquitous storage bricks; for performance reasons as well as high availability they can be found from the smallest to the largest IT departments, hosting data bases, and in research centres, hosting very large data stores. They can be efficiently used both as shared storage, as in parallel database clusters and failover configurations (mail, file and web servers), and as privately attached storage (local disk "emulation") in non cluster--aware environments (legacy applications, video servers and scientific parallel I/O).

In this dissertation, we argue that a high performance, highly available, POSIX-compliant file system can be built for SAN-based clusters with shared storage disk arrays. Such a file system would be able to efficiently support all but perhaps the most demanding applications, from all problem domains, in small to medium sized clusters (up to a few hundred nodes).

3 Dissertation Focus

3.1 Problem statement

Today's supercomputer is the cluster which, to be conveniently used as a HPC platform, is usually configured in a way that emulates its predecessor's (the MPP) computing and I/O subsystems. But HPC clusters are only cost-effective when they are built from COTS parts, i.e., mainstream SMP server nodes. Unfortunately, the reliability of "the cluster" as a whole falls quickly when the number of nodes increases, and fault tolerant solutions must be used if one wants to provide the same service level supercomputer users are accustomed to. Solutions that withstand compute node failures rely on the ability of software – usually middleware – to perform cluster-wide checkpoint/restart of computations, while those to recover from I/O node failures are two-fold: when using internal disks, one could rely on the ability to cross-replicate data among nodes (a solution not used in HPC because it has an unacceptable overhead); or, instead, dispense altogether with internal disks and use external disk arrays and additional software to provide fault tolerance.

The external, array based solution used today in large HPC centres is, in fact, exactly the same approach that has been used for quite some time in business data centres to support highly available DBMS, mail, web and file servers, etc. However, there are some differences between those environments, including both the *file system*, and the *storage access model*. HPC-oriented file systems are geared towards maximum I/O bandwidth, use partitioned (also known as distributed) storage, and usually aren't fully POSIX-compliant; data centre environments favour high availability general purpose file systems, which, conversely, use

shared storage and are fully POSIX compliant – and to provide it, if necessary, they sacrifice performance.

In this dissertation we focus on the development of a prototype *cluster file system* that uses the shared storage approach and is fully POSIX compliant, while still being able to provide a high bandwidth, low latency access to reliable storage. The POSIX API is enhanced through the addition of new "option flags" to the `open()` call, although, in the future, new primitives may be added; both will allow the user to have a better control of the file system behaviour, and increase its performance.

3.2 Contributions

The foci of this dissertation are five-fold:

- To characterise the areas commonly known as "Parallel I/O" and "Parallel", "Cluster", or "Distributed" file systems and propose a set of rigorous definitions.
- To propose a reference model that encompasses all layers from the upper, data management services down to the device layer, and define a taxonomy for the "File System" layer.
- To propose a new architecture for a shared disk Cluster File System (CFS) that overcomes current parallel and cluster file systems inability to simultaneously provide full POSIX compliance and high performance.
- To develop a prototype (based on modifications to Red Hat's GFS[1]) for the proposed CFS, one that is fully POSIX compliant while still being able to provide a high bandwidth, low latency access to SAN-based reliable storage.
- To assess the prototype, comparing it against well established file systems running a synthetic benchmark.

3.3 Organization

This document is organised as follows:

Part I presents the motivations for our work, along with a small introduction that covers the transition from the supercomputer architecture to clusters; it also presents a brief overview of applications that need high performance I/O, including scientific, database, and multimedia, and lays out the focus and major contributions we anticipate from this work.

Part II presents a brief survey of computer, storage, and operating system architectures used when problems do not fit in a "single-box" anymore; we start with SMPs and then move to multi-node MPPs, non-uniform memory architectures (NUMAs) and clusters. In the storage section, we introduce storage devices (from disks to storage arrays) and interconnect

[1] See Part IV, section 15.3.1.2 for a thorough description and references.

architectures (from internal I/O busses to storage area networks). Finally, we briefly mention operating system choices for single and multi-node architectures.

Part III discusses fundamental concepts in file systems; we discuss the user-level views of file organization and access, sharing semantics and data consistency, and an array of techniques commonly used to enhance performance, such as data distribution and caching – starting from the perspective of the single-node computer and then moving to multiple node architectures. Each concept/technique is then illustrated with a "real world" file system.

Part IV starts by discussing I/O flow in modern architectures and operating systems, which allows us to extract precise definitions for *Parallel I/O* and *Parallel Disk Access*. Then a new Reference Model for Data Management Architectures (RM-DMA) is proposed, and taxonomies for the three topmost layers (File System, Object Storage, and Storage Access) are presented. A short evaluation of the model and accompanying taxonomy is carried out as a survey of some relevant, widely known, "parallel", "distributed", "client/server" and "cluster" file systems, I/O software stacks, and storage architectures.

Part V starts with a critique of traditional shared-disk cluster file systems, listing their features and benefits as well as limitations; while we specifically refer to Red Hat's GFS, remarks also apply to other CFSs. To validate whether initial ideas, e.g., using the LAN as a secondary path to move data among nodes, were sound, we have developed a pre-prototype and some preliminary tests were carried out. Results were very positive and led us to propose a new architecture for shared-disk CFSs, one that moves data sharing from the device to the file system cache while preserving POSIX semantics across cluster nodes; we call it the "parallel Cluster File System", pCFS.

Part VI is a prerequisite to understand the pCFS implementation: the first section discusses the architecture of the Linux VFS and how it is used to integrate specific file systems; then an overview of GFS internals is presented; and finally we describe, with some detail, how GFS implements locking and uses it to promote clusterwide coherency.

Part VII describes how we have implemented pCFS, through the addition of two kernel modules, a user-space daemon, and slight modifications to GFS code; the modified GFS code distributes information about clusterwide open files and active regions, and implements cache coherency without resorting to expensive disk flushing and cache invalidation operations.

Part VIII benchmarks pCFS against "plain" GFS and other well know file systems such as NFS and PVFS (where both the "regular" configuration, with internal disks, and the high available configuration, with disk volumes provided by a disk array, were benchmarked); these benchmarks go beyond the usual set of metrics and also account for CPU consumption.

Part IX assesses the benefits of pCFS – its use of an integrated approach to data movement, cooperative caching, and low latency cache coherence operations – and how they succeed in overcoming the I/O bottleneck. Finally, it introduces ideas for future work.

Part II:

A Survey of Computer, Storage, and Operating System Architectures

In this Part we present a brief survey of computer, storage, and operating system architectures that are used in those situations where the problems we want to address do not fit in a "single-box" anymore; we start with SMPs, then move to multi-node MPPs, non--uniform memory architectures (NUMAs) and clusters. In the storage section we introduce storage devices (from disks to storage arrays) and interconnect architectures (from internal I/O busses to storage area networks). Finally, we briefly mention operating system choices for single and multi-node architectures.

4 Parallel and Distributed Computing Architectures

4.1 Architectural archetypes

The topics we are going to cover now are introduced in a simple way by Fig. 4.1, as it charts architectures we're addressing on a two-dimensional grid built along two axes: the number of CPUs, and their "distance", measured in terms of memory access latencies.

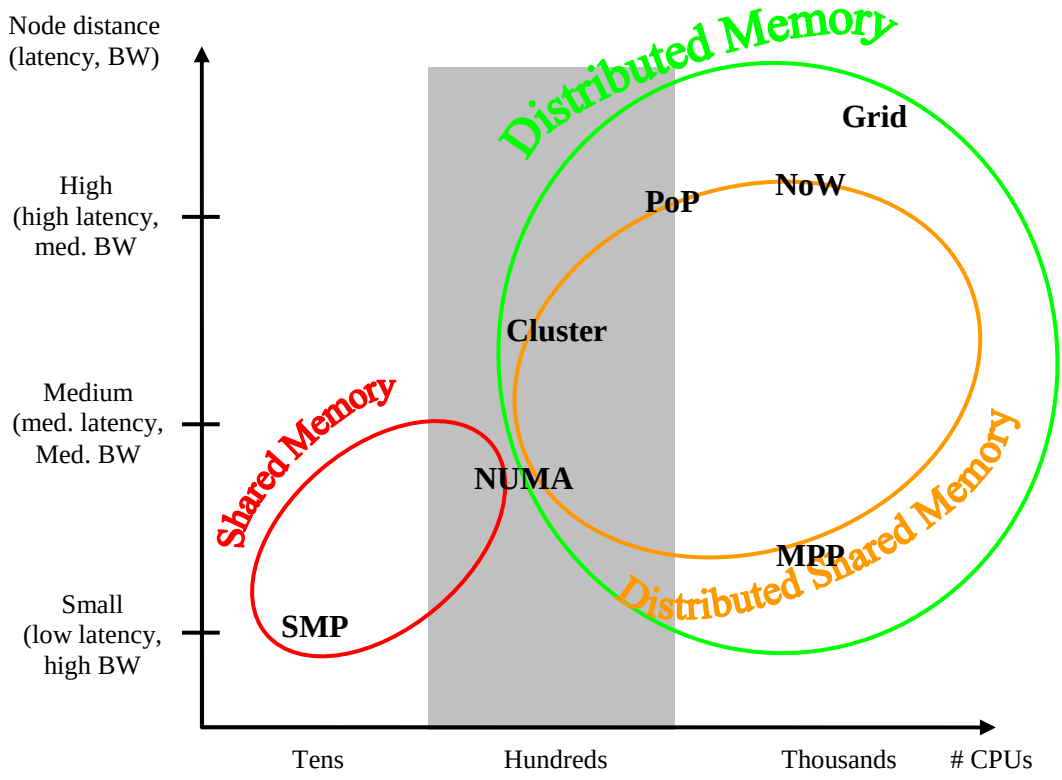


Figure 4.1 Architectural archetypes "at a glance" (hardware-biased view)

To illustrate the placement of an item in the chart let's look, for example, at a typical SMP: it has a few CPUs (so we place it in the "Tens" zone), sitting close to each other on a low latency, very high bandwidth interconnect (quite often, a shared bus) and therefore we place it close to the "Small" line. We then group ("encircle") similar architectures into families, according to the way CPUs access memory; for example, in *shared memory* architectures all CPU(s) can access the whole memory – and they are either of the Uniform Memory Architecture (UMA) variety, when all the CPUs can access all memory modules at the same "speed" (latency), or of the Non-Uniform Memory Architecture (NUMA) variety, when a CPU may access some memory modules at a faster "speed" (latency) than it may access others.

4.2 The shared memory multiprocessor

The symmetrical shared-memory multiprocessor (SMP[1]) is today's prevailing architecture for small size (up to 4 CPUs) and even medium size (up to 8 CPUs) COTS systems; it is so common that one can find a huge amount of literature, including textbooks, manufacturers *white papers* and computer magazine articles, and was scarcely a research topic. However, recent developments on multicore architectures have, once again, spurred research on SMPs.

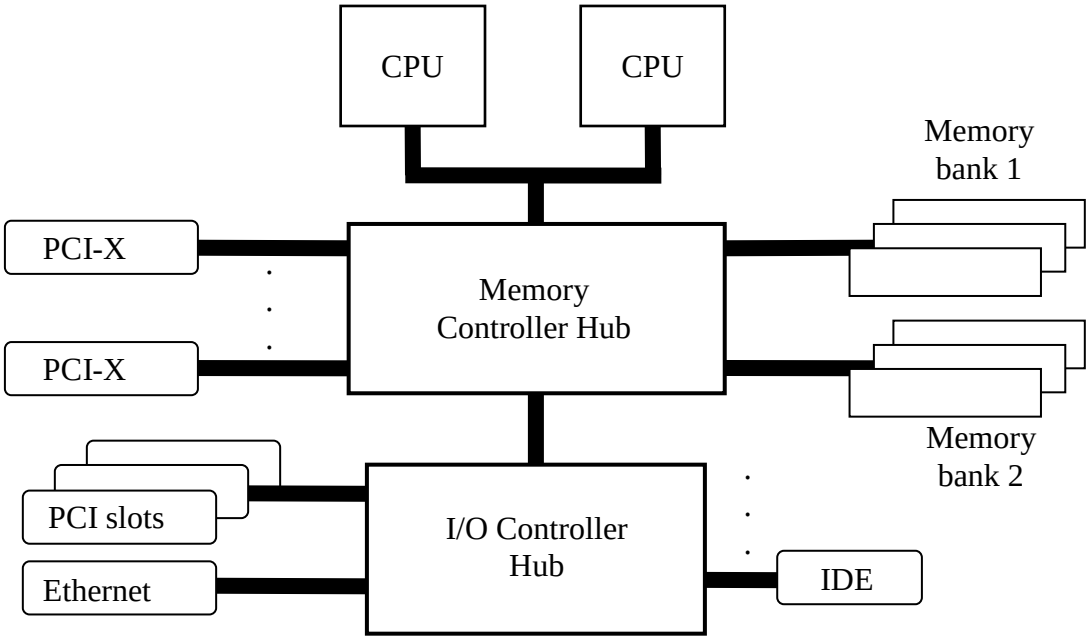


Figure 4.2 Architecture of a common off-the-shelf SMP server

Current Intel-based SMPs are of the uniform memory architecture (UMA) type, where the cost (latency) of accessing a memory location is the same for all CPUs, while AMD has been busy selling their Opteron-based SMPs [Jes05], a shared non-uniform memory access architecture (NUMA, to be detailed in section 4.4); understandably, NUMAs may need some operating system assistance, such as the Linux NUMA extensions [Dob03, Bli+04], in order to transparently achieve "optimum" application performance.

There are some very strong points in favour of the SMP architecture: it is very easy to program, as it implements the shared memory programming model; it is, at least with a properly designed memory subsystem, very easy to scale CPU performance if the number of CPUs is kept low (let's say less than a dozen); and, with properly designed memory and I/O subsystems, it is also relatively easy to scale I/O performance.

But this easiness is for low numbers; in fact, it is very difficult to simultaneously increase both the number of CPUs and the I/O bandwidth because a COTS SMP is designed around a

[1] We will, unless otherwise noted, use SMP to refer to shared-memory multiprocessors, a more generic architecture that includes the symmetrical shared-memory architecture as a subtype.

memory and I/O interconnect topology (usually one or more buses) which is not scalable. For cost reasons[2], this subsystem is built as PCB lanes onto the motherboard, and can not be "widened" to support a larger data path; another way to increase bandwidth is to increase the bus clock rate, but that's not an easy task: the electrical characteristics of the bus dictate the fastest clock rate it can sustain, and anything that's plugged in the bus only contributes to deteriorate its characteristics; so, if we succeed in increasing the clock rate in a specific motherboard's bus, it just means that it was being underutilized before.

Historically, the I/O subsystem has tried very hard to keep up with the CPU performance; the Peripheral Component Interconnect (PCI) bus, plugged into a "south bridge", has evolved from the original mid nineties 33 MHz, 32-bit wide PCI (at 133 MB/s) to a 66 MHz 64-bit wide bus found in mainstream products in late nineties (at 533 MB/s); now, PCI-X, with a bandwidth of circa 1 GB/s (with 2 and 4 GB/s almost ready to take off) can be found directly attached into a memory hub [PCI-X]. But high performance disk arrays on Fibre Channel at 800 MB/s per full-duplex port, and Gigabit Ethernet devices, at 100 MB/s, can still saturate it. A solution to this problem includes, among others, the latest generation of serial-based busses and interconnects: PCI Express (PCI-e) [Bha01], and Infiniband[3] (IB) [Pfi01].

4.3 The massive parallel processor

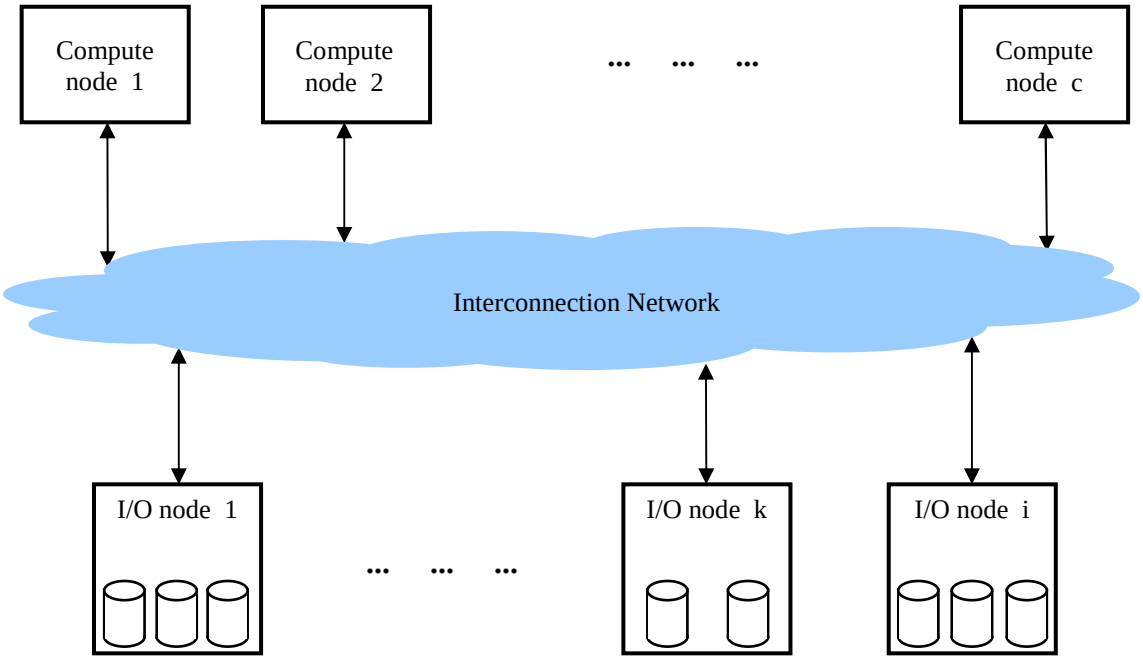


Figure 4.3 Architecture of an MPP

[2] To see what can be done in big, expensive shared memory architectures, see Sun's Starfire [Cha98].
[3] IB has yet to fulfill the promise of being an alternative to PCI [Pfi01]; tight control, by Intel, of the Front Side Bus (FSB) has deterred developers from "plugging" into the memory hub.

The Massive Parallel Processor (MPP) is another approach to increase raw computing power; the concept is based around a large number of nodes[4] sitting close to each other, linked together with an expensive, special-purpose, high bandwidth low latency interconnect – usually a non-bus topology (*e.g.*, mesh, hypercube, torus, etc.); depending on the architecture and/or topology of the interconnect, increasing the number of nodes can vary from reasonably easy to very hard; anyway, the available raw computing power will increase accordingly. The MPP is obviously well ("naturally") suited to support the message passing programming model, due to its distributed memory architecture.

The prevailing I/O architecture for MPPs is based on the use of a certain number of *I/O nodes* that either hold internal disks, or have a separated interconnection infrastructure linking them to external storage; these I/O nodes are, together with the compute nodes, attached to the general-purpose interconnection network [Ber+94].

The big advantage of the MPP architecture is its (theoretical) scalability: it is (at least with a properly designed interconnect) very easy to scale up raw performance, even in a large configuration (thousands of nodes), just by adding more compute nodes; and, similarly, to increase raw I/O performance, one may just add I/O nodes. On the down side, it not easy to program it, as message passing is the programming model of choice (some authors will strongly disagree with this statement). As a consequence a large amount of software originally developed for SMPs will not run in MPPs; to solve this problem, two approaches are therefore possible: porting the software, which can be a very expensive/time consuming endeavour for large products (*e.g.*, the port of a DBMS engine), or simulating a MPP-wide shared memory with appropriate middleware – this approach, called Virtual Shared Memory, VSM [Li+86], has been shown adequate for some applications.

A special point must be noted: we have been using the term *raw power*, which is the aggregated sum of the power (computational or other) of all the nodes; but one thing is to advertise the raw power, while another one is to be able to use it productively, to run applications. MPP applications are very sensitive to the layout of data distribution among nodes, as well as to the frequency of communication and amount of data exchanged between the nodes; if not properly done (which is no easy thing to do), performance will be much lower than what one could expect[5]. Another sensitive point is I/O: if computing nodes do not have direct access to storage, all I/O data must travel through the network interconnect, and this should be done in a way that does not interfere, *i.e.*, delay, application message exchange; one should strive for a segregation between the I/O data transfer messages and application

[4] A node is a package containing a complete system: CPU(s)-memory-I/O.

[5] For example, the performance of a well known proteing-folding application on a 2048-processor Blue Gene is about 4 ns/day, while it reaches 15 ps/day in a IBM 595 "big NUMA" with 944 CPUs.

communication, and, if possible, overlap them with computations; otherwise, we will be increasing the sequential term in Amdahl's law [Amd67, Gus88], and speedup will suffer.

Development of MPPs has been lingering for quite some time, losing for the much more cost-effective clusters, but recently IBM has been commissioned to develop the Blue Gene architecture, a "massive supercomputer" [Gar+05].

4.4 Distributed shared memory architectures and NUMAs

A distributed shared memory (DSM) architecture is a specialised distributed memory architecture[6] where it is possible for a node to use a separate interconnect (not the CPU-local memory interconnect) to access another node's memory; this remote memory access capability is provided by special hardware (which may, or may not, be complemented with OS-level software). DSM architectures were developed to overcome both "the" limitation – poor scalability – of bus based shared memory multiprocessors, and the low performance of software based virtual shared memory implementations while retaining their major strength: the shared memory programming model.

While the most general definition of Non-Uniform Memory Architecture (NUMA) encompasses all architectures where the latency of accessing distinct memory addresses may differ (*e.g.*, an omega network used for CPU/memory interconnection), it is used today mainly to refer to DSM architectures. NUMA research has focused mainly into three different architectures: the cache-coherent NUMA (cc-NUMA), the cache-only memory architecture (COMA), an implementation with coarse grained shared memory coherence, and the generic, non cache coherent, NUMA [Len+95]. Several cc-NUMA architectures were successful commercial designs in the past: the Kendall Square Research KSR-1 and KSR-2, the Convex (now HP) Exemplar, the Silicon Graphics Origin series, the Sequent (now IBM) NUMA-Q, and the Data General (now EMC) AViiON 20000. Silicon Graphics (SGI) is one of the companies still on the market with a (cc-)NUMA architecture, the Altix range of high performance computing systems (with up to a few thousand nodes); another is IBM with its large pSeries systems, *e.g.*, the 128 CPUs p575.

The advantage of a cc-NUMA over the NUMA is in the hardware-assisted coherence between local and remote (also called far) memory; it increases performance and makes the development of the operating system much easier, so both user-level software (applications) and "middleware" (such as DBMS engines) can run unmodified, although they may need tuning if we want to extract adequate performance.

[6] Notice that we restrict the more generalised use of the terminology, as does [Len+95]: we define DSM architectures as hardware-enhanced, thus eschewing software-only solutions, which we call VSM.

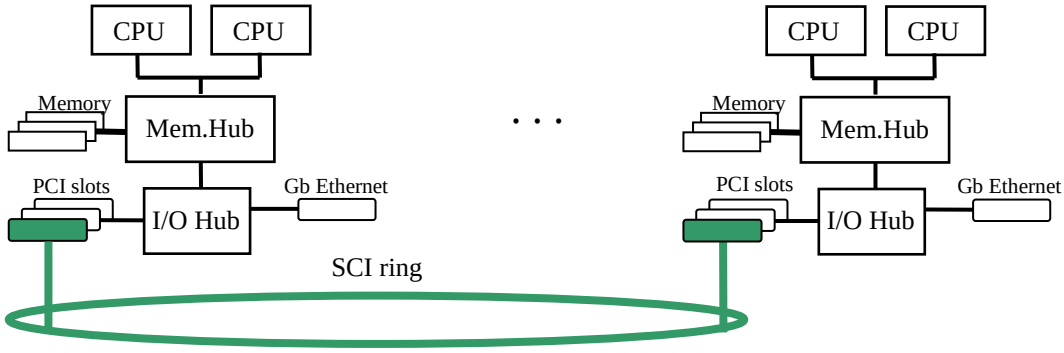


Figure 4.4 cc-NUMA architecture

Most commercial designs (exceptions were SGI and KSR – this one not exactly a "commercial" design) were based on the Scalable Coherent Interface (SCI). SCI is an IEEE standard that provides very high performance (both low latency and high bandwidth), bus-like functionality, to a large number of nodes [Gus92, IEEE92]. It uses a packet-based communication protocol over unidirectional links connected in a ring topology, and provides remote memory access capability, which, together with a cache coherence protocol (an optional feature on the standard), enables us to offer a unique globally shared memory across nodes. SCI was the first of a series of Remote DMA (RDMA) capable, high bandwidth, low latency standard interconnects; today, the most prominent ones are Infiniband [IBTA01], and Myrinet [Nan+95, VITA98, Myr00].

4.5 Cluster architectures

As the name suggests, a cluster is a group of machines (sitting "close" to each other); from this common ground, quite a few different interpretations of what a cluster is can be found, particularly if one includes vendor whitepapers and magazine articles.

Informally, a cluster is a group of nodes (with or without local disks), which we will call *cluster nodes*, interconnected by some sort of networking infrastructure. Thus, from an architectural point of view, a cluster is a close relative of the MPP – and thus well suited to implement the message passing programming model; it can also be pictured exactly as the MPP, and so Fig. 4.3 may also be used to describe a cluster. There are, however, differences: the cluster interconnect is often either a general purpose network (*e.g.*, Ethernet), or a more specialized (read: expensive) and better performing interconnect (but nevertheless, easy to "shop"), while the interconnect used in MPPs is just the opposite – an expensive, purposely built one; and a cluster node is usually a complete packaged computer (the "ultimate" cluster building block of today is the *blade*) or, at least, a complete motherboard, while an MPP node may be something ranging from a special board that is inserted in a frame (similar to blades, but no power supply), to a fully "boxed" item that is inserted into a cabinet – *i.e.*, expansion in a MPP can become impossible when the cabinet is full.

The strong and weak points of clusters are quite similar to those of MPPs: on the plus side, it's very easy to increase the raw computing power, as one just needs to add new nodes, and to increase both the I/O capacity and raw bandwidth, as all one has to do is either add disks to existing nodes, or add new nodes with their own disks. The less favourable characteristics of cluster architectures are the message passing programming model, and lack of off-the-shelf software. But there are two very special points that must be noted: on the positive side, for the same raw computer power, a cluster may be one or more orders of magnitude cheaper than its MPP counterpart; and, on the negative side, its "usable performance" may be more sensitive to the issues of application communication patterns, data distribution and I/O.

Wishing to eliminate, or at least improve on the weak points of cluster architectures, some researchers have successfully experimented with high performance interconnects, instead of being tied up to Ethernet only. Today, clusters can be found using Infiniband, Myrinet, or SCI making them usable in situations where sensitivity to the data distribution and communication patterns precluded the use of cheaper Ethernet (e.g., HPC and parallel DB clusters). Currently research efforts are underway to use these interconnects (previously SCI, today Infiniband) to implement distributed shared memory – in fact, turning the COTS cluster into a NUMA or even a cc-NUMA "single system image" (SSI) computer[7].

4.6 PoP and NoW

PoP (Pile of PCs) [Rid+97] and NoW (Network of Workstations) [And+95] are ways to aggregate small computers, and use them together; these terms have been used in a somewhat *ad hoc* manner, but we think that the term PoP should be used to describe a group of PCs sitting close to each other in a single room, perhaps aligned on a rack of shelves, while NoW should be used to describe a larger "cloud" of small computers scattered in a large building or in a campus.

If we stick to the above definition, PoPs are in fact "unpackaged" clusters, and so they share with them the same configurations and constraints: due to the inter-node distance, for example, it is possible to link them with high bandwidth interconnects and create a NUMA. But that may not be feasible in a typical NoW, as nodes may be somewhat far away from each other; so, NoWs use Ethernet, and if we want to implement a shared memory layer, it will have to be a software-only VSM solution.

4.7 Grid

The Grid is a structuring vision for a "flexible, secure, coordinated resource sharing among virtual organizations – dynamic collections of individuals, institutions, and resources" [Fos+01]. From the architectural point of view – the only that we're interested in here – it's

[7] For a brief description of SSI see section 6.4, "Operating Systems for Clusters".

just like any other large scale distributed system, one that may encompass many different resources, from single-user PCs to large clusters, from PDAs and mobile phones to sensors and 3D display devices, etc., all interconnected by a wide, geographically dispersed network.

Notice that, being "the Grid" a very hot research topic, and one that is not covered in our work, we've added this subsection (and the next one on cluster federations) just for completeness of the survey; it is, consequently very brief and incomplete.

4.8 Cluster federation

A cluster federation, as the name suggests, is made out of individual clusters; it is a federation in the administrative sense, *i.e.*, there is an agreement on policies such as resource management and access, user authentication, etc. From the architectural perspective, a cluster federation is another large scale distributed system; but, unlike a grid, it is more homogeneous, both in site and network homogeneity: each site is a cluster, and the network access point at every site is a high bandwidth, dedicated infrastructure.

As far as we could trace it (as with the grid, cluster federations are not a topic of study in this dissertation), the concept of a cluster federation seems to have been originated from two opposite directions: as an expansion, from clusters to larger systems (the term "federated grids" can also be found), as in the move from single administrative domains, tackled with Condor, to multiple administrative domains, through the use of Condor-G [Fre+01], and as a smaller, simpler, and more predictable "grid" [Xtreem].

5 Storage Architectures

5.1 Architectural building blocks

Gone are the days when the only direct access storage device (DASD) that could be plugged into a system was the magnetic disk; now we also have optical and hybrid disk technologies (which we will ignore together with other technologies, such as tapes), solid state disks and, more important, the storage disk cabinet.

The storage disk cabinet is an external device that has its own power supply (and very often redundant power supplies), hosts a fairly large number of discs (tens to hundreds), and has an I/O channel[8] interface of some sort. The storage cabinet is the basic building block for the storage array [Pat+89]; the array "feature" adds memory and processing power to the cabinet, allowing us to create logical volumes (also known as logical disks) out of groups of physical disks. A group generally adds some property to the "basic disks" that constitute it, such as higher performance or some sort of fault tolerance; commonly found groups use the different

[8] Here we deviate slightly from the historical IBM/360 I/O channel concept; we use the term to refer to an interface that is connected to a DMA-capable adapter which offloads the host's CPU for the most part of an I/O task.

RAID levels offered by the array, which often include levels 0 (also called striping, no fault tolerance), 1 (mirroring), 0-1 (combined striping and mirroring), 3 (bit interleaving with dedicated parity disk) and 5 (block interleaving with rotating parity disk). The host computer can only see each logical volume, not the individual physical disks that make the group (unless a configuration called Just a Bunch Of Disks – JBOD – where there is no grouping at all, is used, and therefore all physical disks in the cabinet are visible).

5.2 Direct attached Storage

Direct attached storage (DAS) is the oldest form of interconnection known to computer architects, as shown in Fig. 5.1: each storage device – disk or disk array – is connected to one and only one host computer, *e.g.* internal disks are connected to their I/O host adapter(s) in a pretty similar way as the array's storage controller is connected to its own adapter.

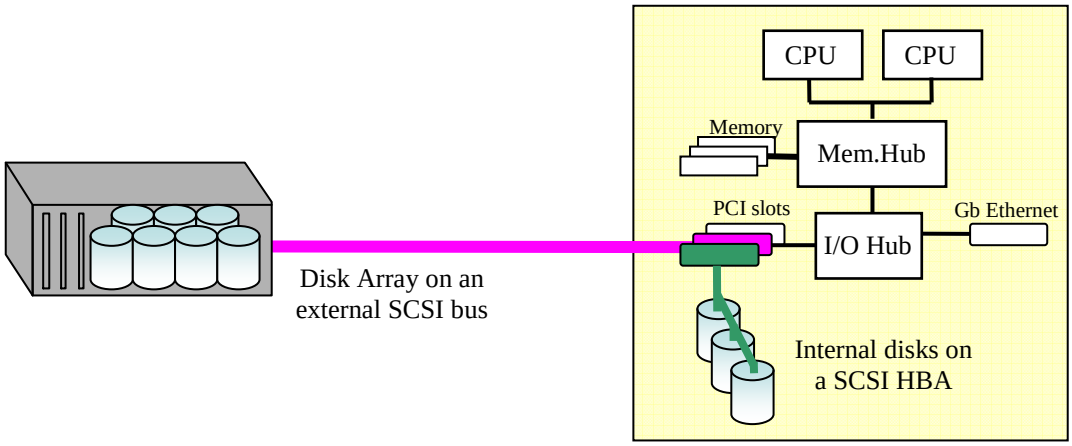


Figure 5.1 Direct attached storage with DASDs and Storage Arrays

5.3 Shared storage and storage area networks

A very important I/O channel technology that spread in the mid-eighties, and still prevails today, is the Small Computer System Interface (SCSI). Developed in 1981 by Shugart Associates and the NCR Corporation, it was submitted to the ANSI X3T9 committee and became an official standard in 1986 [SCSI-1]. SCSI introduced to the minicomputers of the eighties an inexpensive way to connect disk devices (called *targets*) to one (or more) host adapters (called *initiators*) via a shared bus – shown, in its simplest configuration, in Fig. 5.1.

The SCSI protocol allows an initiator to send commands to a target; all bus entities are uniquely identified by a SCSI ID, or, if they are target devices (*e.g.*, disks, tapes), by a SCSI ID/LUN (Logical Unit Number) pair. This has several interesting possibilities, but we will discuss just two: i) the SCSI bus can be "driven" by two initiators in the same host, and one may use a specialised fault tolerant driver which can detect a failed host adapter and "switch" to a good one, which was dormant, as shown in Fig. 5.2; and, ii) the initiators may be plugged

into different hosts, as shown in Fig. 5.3, and each host (initiator) may be configured to access only a subset of the disks ("its own subset").

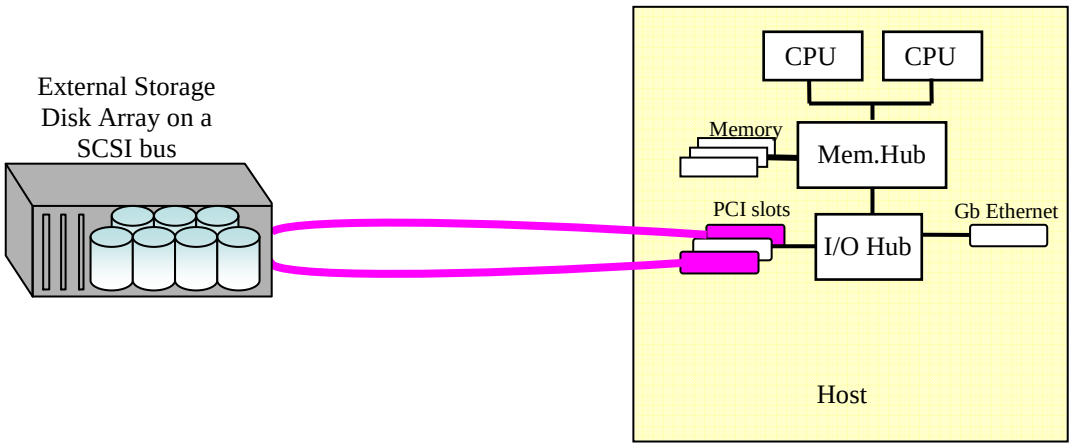


Figure 5.2 Host-based HA: shared SCSI bus over two host adapters

Option ii) is called a *dual-initiator* configuration, and requires adequately "enhanced" SCSI drivers to support *LUN masking*, a way to restrict the set of LUNs that the host (driver) is allowed to access. It is important to notice that we have moved from internal storage, where all disks are accessible only from one host, to an external cabinet with a pool of drives that can be configured to satisfy the storage needs of each system at a particular moment, and later on be reconfigured to satisfy a different set of needs (*e.g.*, "System A" has now a lot of free space, let's mask out one or more disks so that they can be used on "System B").

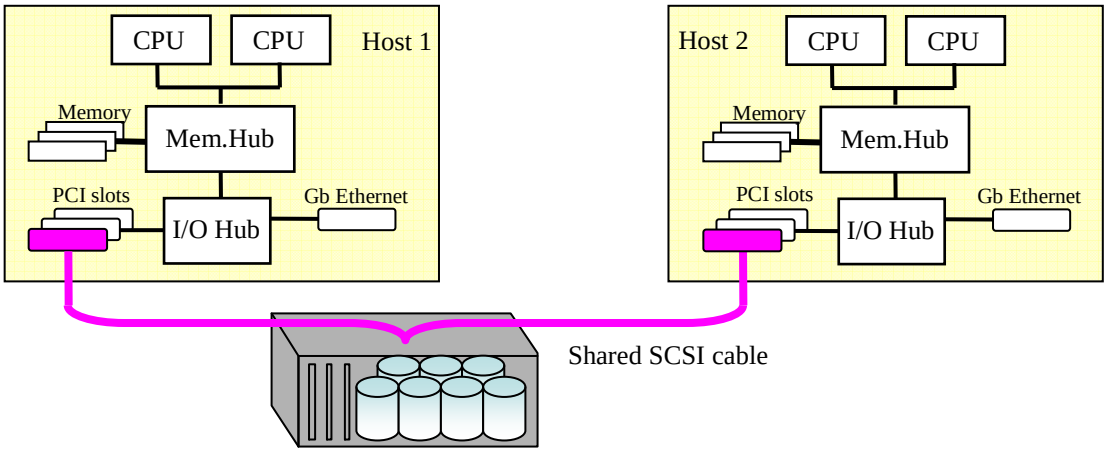


Figure 5.3 Node-based HA: Attaching an external storage cabinet to two hosts

In SCSI-2 there was an address space of 16 SCSI IDs, so we could connect up to 15 hosts (with one adapter each) leaving out one ID for the target device. It's not an easy task, to connect all these cables – in fact it is virtually impossible, so the SCSI hub was developed. A SCSI hub is a device that behaves just like an network hub: it's a star topology that

implements a shared bus; and it gives users a string of benefits, such as the possibility of, at the flick of a button, removing a device from the bus without causing a total failure – just like network hubs (well, almost; network hubs don't have buttons – one simply unplugs the cable to disconnect something). Now the configuration closely resembles a network; in fact, it's called a Storage Area Network, or SAN[9].

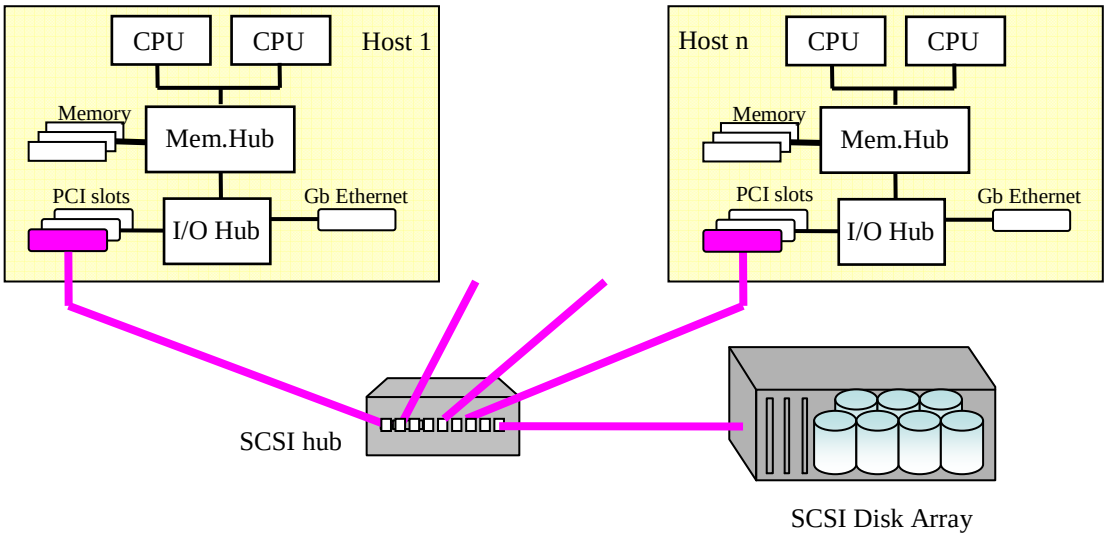


Figure 5.4 A Storage Area Network (SAN)

SCSI has been so widely used that has been the target of a lot of enhancements over the years, and specifications SCSI-2 [SCSI-2] (which replaces SCSI-1) and the SCSI-3 "family" are now part of the standards. SCSI-3 was a major step, because it separated the SCSI protocol from the cabling itself, thus allowing it to be used with any transport, as in Serial Attached SCSI [SAS-1], and also to be encapsulated in other protocols, as in Fibre Channel, where it was integrated in Layer 3 [FC-FCP], and over IP, as defined by the iSCSI protocol [Kru+02]. Thus, there is a specific SCSI-3 annex for each combination of protocol and interconnect.

Storage Area Networks are a hot topic: a lot of research has been done over the years, and every major player in the hardware arena – both computer manufactures and storage companies – has a string of products for SAN. Fibre Channel (FC) was the enabling technology, overcoming the 15 m maximum parallel SCSI bus length (with 30 m for copper and 10 km for optical fibre) and the complexity and fragility of the connectors (with 68 to 80 pins for parallel SCSI *vs.* two twisted pair conductors for copper FC, or two optical fibres in optical FC). Fibre Channel also offers aggregate speeds from 200 to 1600 MB/s (over dual 1, 2, 4 or 8 Gbps serial links) against an initial SCSI-1 offer of 10 MB/s (parallel SCSI now

[9] SAN may have another, totally different meaning: System Area Network. It is often used to describe a group of hosts interconnected by high bandwidth technologies such as Infiniband, Myrinet and SCI; we will use the SAN acronym only to refer to storage area networks.

boosts up to 320 MB/s but only in very short, internal cable runs). All previous figures can be "updated" to the latest FC technology just by replacing the words "SCSI" with "FC".

5.4 Network attached storage

The concept behind Network Attached Storage (NAS) is, to put it simply, to offer a plug-and-play file server with the administrative costs of an appliance.

The file server concept started in the mid-eighties with Sun's Network File System (NFS) [San+85], and progressed through with the integration of file sharing within Network Operating Systems (NOS), including Novell's Netware OS with the Netware File Sharing Protocol (NFSP), and Microsoft's Windows NT with the Common Internet File System (CIFS) [Her04]. A file server "internally" stores files and folders that may be remotely accessed (shared is term generally used in IT) by client machines, via network – usually, an Ethernet LAN. These systems have flourished for the last 15 years or so, but to some users the burden of the administration tasks needed to keep them running grew out of proportion – and one of the reasons was that the "box" hosting the file server also had a full fledged operating system, requiring regular OS system administration tasks (user profile maintenance, selective backups, software upgrades, etc.); to make things worse, the three file sharing protocols quoted above (CIFS, NFS and NFSP) are incompatible with each other.

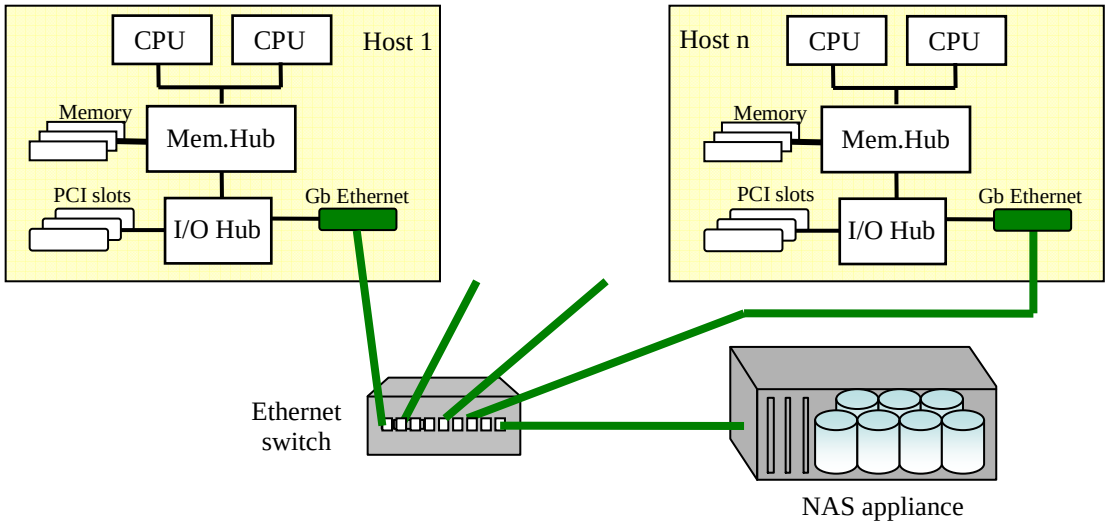


Figure 5.5 A Network Attached Storage (NAS) solution

The NAS appliance, developed throughout the nineties, was the solution: a storage cabinet, with CPU, memory, disk and network I/O; a special or general purpose, but stripped down version of an operating system, and software to implement one or more file sharing protocols; the appliance can interoperate with multiple NOS servers and clients, so the administrative costs are trimmed down to a minimum. Current versions of NAS servers are highly developed products, including their own proprietary internal file systems (with hardware or software

RAID capabilities), and being able to share the same file to several clients using different protocols concurrently: for example, the same file can be accessed, for example, on a UNIX client using NFS and UNIX file permissions, while on a Windows client using CIFS it will use Windows' Access Control Lists. Fig. 5.5 shows a LAN with two clients and a NAS appliance.

Network Attached Storage is, together with SANs, a hot topic and NAS proponents are actively pushing the concept as "the" storage solution; right now NAS is being used mainly to replace and consolidate file servers; it is not being used as a general purpose storage box – for example, it can not always be used to host a database, as some mainstream database technology vendors do not support their products in the NAS environment.

Network Attached Secure Disks (NASD) [Gib+98] is a research project at Carnegie Mellon University rooted on the same ideas of NAS – giving access to storage through the network. The idea behind their work is to get rid of all the excessive data movement inside servers and clients; for example, in a file server data has to be moved from the disk to the OS buffers and, from there, to the network buffers, down the network layers, and into the server's (or NAS) Network Interface Card (NIC); then it must travel through the network to the client's NIC, up the network layers into the client's OS buffer, from where it finally gets moved to the application program that requested it. NASD advocates that each disk must have its own NIC, processing power and software, and be directly attached to the network. Another project that originated from NASD, Active Disks [Rie+01], proposes that disk devices can be built where the device's processing power is enough to build file or data base management systems directly "on disk", thus eschewing traditional file and database servers.

5.5 Object storage devices and Object-based storage

The Object-Based Storage Device (OBSD, also known as Object Storage Device - OSD) is an abstraction used to redefine the roles and capabilities of storage devices (disks, tapes, etc.); an OBSD is able to accept commands that create object sets (or groups), with or without quotas, and then accept the creation of identifiable objects (instances, within the set) automatically managing the necessary storage space. To access such an object one may start with an "object open" for an identified object, then read and/or write, retrieve and/or store some attributes, and end access to the object with an "object close"; OBSDs thus "know" when an object is in-use. This approach will move most of the work currently performed by the host (running the Object Storage Layer, see section 13.3) to the devices, thus alleviating file system implementations, and therefore providing opportunities for performance increases in the file system [Mes+03, Fac+05].

The OBSD originated from the NASD work and, after years of cooperation between industry, academia and standard bodies, was transferred to the Storage Networks Industry

Association (SNIA). The SNIA and the International Committee for Information Technology Standards (INCITS) T10 committee “Object Based Storage Group” have already ratified a standard on SCSI commands for object-based storage devices [And99, T10-04].

6 Operating Systems

6.1 Operating systems for shared memory architectures

Research on operating systems for small-scale shared memory multiprocessors is quite stable at the small size SMP (*e.g.*, Intel IA-32/64, IBM PowerPC) and cc-NUMA (*e.g.*, AMD Opteron) architectures; Linux is quickly catching up with commercial UNIX derivatives (HP/UX, IBM AIX, Sun Solaris) with enhancements in scheduling (the so called O(1) scheduler [Lov03]), POSIX threads support [Dre+05], memory management (support for large memory, large pages and NUMA extensions [Gor04]), and I/O (volume managers, LVM [Lew05] and EVMS [Pra02, Lor+05], direct I/O [Bov+05], asynchronous I/O [IEEE04, Bha+03], and both Linux vectored [Bov+05] and POSIX list-directed I/O [IEEE04]).

However, the advent of high levels of on-chip parallelism either in the form of multicore architectures with tens of cores, such as Intel’s TeraFlops processor [Van+07], or with many hardware threads, as implemented in the Sun’s Niagara processor [Kon+05], has once again revived OS research; OS support for many-cores must not only address large numbers of threads efficiently (a problem that has been researched before), but also dispatch them in a way that application efficiency can be increased, e.g., leading to a decrease in execution time for parallel applications. Research in OS scheduling, such as on extensions to support gang scheduling [Raj+07] of related threads has therefore become a hot topic.

Furthermore, research efforts are also strong in highly available or near fault-tolerant OS extensions, e.g., for on-line subsystem configuration and de-configuration (such as hot pluggable memory, CPU, and devices [Bor+05]), self healing software and autonomic computing [Gan+03], and support for resource partitioning and virtual machine environments.

6.2 Distributed operating systems

The term “distributed system” has been used to describe a system consisting of several interconnected computers that do not share either memory or a clock, each one having its own memory and processor(s). Distributed systems range from strongly interconnected MPPs, to clusters or PoPs in a room, large installations of NoWs in a campus or in a metropolitan area network, country or continental-wide cluster federations, world-wide grids, etc.

Ideally it should be possible to present a distributed system to its users as “a single (big) system”, one which would roughly provide them with the same interface and set of services they’ve grown accustomed to, and not just a disparate collection of isolated computer

systems, each one running its own, independent, OS copy. Under the ideal distributed operating system (DOS) users would have to login only once, anywhere, and they would always get the same environment, being able to browse through the system's resources, list files, observe the status of running processes, etc.; they'd be able to launch their jobs (processes) in the same (or in a very similar) way they are used to do it in a single-node computer – and expect the DOS to schedule them on the "best node(s) for the job", and access files irrespectively of where they are stored. And, furthermore, applications would run efficiently, increasing user satisfaction! Transparency is the keyword that could be used to characterize the behaviour we've just described; fundamental in a distributed operating system are (adapted from [Tan92]) name, access, and location transparency. Other desirable properties which, in a DOS, should also enjoy transparency are (again, adapted from [Tan92]) migration, replication, concurrency, parallelism, and failure.

Unfortunately there is no distributed operating system capable of implementing all features on our "wish list", if we embrace the entire architecture range; however, there are partial solutions that come quite close in some cases, as we will see below.

6.3 Operating systems for MPPs

Operating systems for MPPs can be either general-purpose, or as specialized as the architecture itself; the Intel Paragon [Ber+94], an MPP of the nineties with hundreds of computational nodes plus a few I/O and service nodes, is an example of a system that uses both a generic and a special purpose OS, depending on the node.

Users access Paragon through *service nodes* running the Mach based OSF/1, a UNIX API--compliant OS (for enhanced compatibility with widely available code) that handles the usual chores: process management (with lightweight thread support), virtual memory management, and inter-process communication services.

Paragon I/O nodes (either with their own internal SCSI disks, or with fast HiPPI links to external disk arrays) also run OSF/1, seamlessly supporting, through the kernel's Virtual File System (VFS) interface, different file systems such as the UNIX File System (UFS), NFS, and Intel's own Parallel File System (PFS). Local file systems can, interestingly, be "unified" into a single MPP-wide Paragon Distributed File System (DFS).

The original design specification mandated that a) a distributed service layer would be added to each Paragon node to provide for a single system image (SSI) vision, one where the whole system would behave as a "single, although very large computer" for users, programmers, and system administrators; and b) that a computational node could be used either in "bare" mode, loaded with a message passing library, or in "full" mode, loaded with OSF/1. In the end, it turned out that OSF/1 was too inefficient to allow compute nodes to perform at their best "rate", so several installations have chosen to replace it with SUNMOS,

a single-task/single-partition OS developed by Sandia Labs and the New University of Mexico; as a consequence, the SSI vision was abandoned, too.

6.4 Operating systems for clusters

Clusters, when used as "commodity MPPs", want to provide their users with the "MPP look and feel"[10] and therefore they segregate nodes into three distinct roles: head, computational, and I/O nodes.

The head node performs the same function as the MPP front-end: it is the single point of administration (users, groups, permissions, etc., resource monitoring, and file system administration), and, in some cases, the only node where users may login (and thus develop, i.e., edit, compile, link their applications); if users are only allowed to log into the head node, then the head node must run a *job scheduler* which accepts user's requests, places them in a queue, and dispatches them to compute nodes according to some specified policy. Finally, I/O nodes store information needed by applications that are running in compute nodes together with transient files they produce.

To provide these functionalities, a common approach is to pick a UNIX-based operating system, such as Linux, and extend it with the necessary middleware. For example, in the head node Network Information Service (NIS, once called Yellow Pages) [Sun02] or Lightweight Directory Access Protocol (LDAP) [How95] may be used to centralise the administration of users, groups, etc., while resource monitoring applications, such as Ganglia [Ganglia] or Munin [Munin] provide vital resource information for cluster administrators; as for batch schedulers, openPBS [openPBS], LoadLeveler [Kan+01] and a plethora of others provide the required functionality. File systems for clusters are a major subject, and will be left to the next section; for now, it suffices to say that the "seamless environment" that we aim to provide would require a file system with the same functionalities of Paragon's DFS; but a usable, although less "transparent" environment can still be built with the more prosaic NFS by configuring the head node to be a NFS server while compute nodes are NFS clients.

The other, less used but nevertheless cleaner approach, is to use a "true" DOS to provide SSI functionality; recently there have been several efforts, fuelled by the availability of high bandwidth low latency interconnects, to provide Linux-based distributed operating systems, such as the distributed shared memory Kerrighed [Lot01, Mor+04]. If one provides a VSM at the kernel level, as Kerrighed does, then processes (and threads) have their address spaces transparently built on top of memory which may be physically scattered among several nodes, turning the cluster into a "big SMP", one who exhibits strong NUMA characteristics but is not a "set of independent nodes" anymore. So, ultimately, there is no need to modify existing applications, not even "command line utilities" such as **`ps`** (which now can report the list of

[10] Of course, in a cluster that was not "configured" to behave as a MPP, things can be quite different.

processes running in the whole cluster); and there is no need to replace runtime shared libraries with specialised versions, a characteristic of some middleware or hybrid approaches.

A third, remarkably efficient approach, is one that does not strive for a true SSI as above but that, for user and administrator tasks, behaves like one, as does Mosix [Bar+98, Bar+99]. Mosix provides the cluster with a dynamic load balancing capability (recall that the batch scheduler approach balances jobs statically, *i.e.*, resources are evaluated just before launching a job), thus allowing a process to dynamically migrate from one node to another. Process migration in Mosix is accomplished by leaving a proxy in the original node when a process migrates; communication with the user and among the migrated process and other processes requires a hop through the proxy, but is completely transparent to the application; execution of library/system calls related to a process' environment, such as `gettimeofday()`, must also take place at the proxy, before the result is sent back to the requesting process.

6.5 Operating systems for large-scale distributed architectures

"Large scale distributed" is an expression commonly and loosely used to embrace a diversity of architectures, environments, and applications; it includes sensor, and other forms of content distribution networks (e.g., peer-to-peer file sharing, video on demand), geographically dispersed collaborative applications and data processing, etc. The large scale distributed environments that we will cover here are cluster federations and the Grid; as we've pointed out before these are covered in the spirit of completeness of the survey and, consequently, are very brief and incomplete.

The paradigm for the Grid [Fos+01] is one of a seamless system for resource sharing; therefore, efforts to develop a grid-targeted operating system were not actively pursued, as it would compromise the grid's ubiquitous nature; instead, the majority of the research proposals is to build on layers of middleware which, if possible, should be operating system agnostic and built upon a minimum set of local services, *i.e.*, relying on the most primitive widespread functions for process management, communication, and storage access, as Globus [Fos+97, Fos05] does. Vigne [Ril06] is a notable exception to this route, aiming to demonstrate that a Grid aware operating system is not only possible, but can, by design, include mechanisms that offer highly available services.

At the cluster federation front, driven by such paradigmatic research facilities as Grid 5000 [Grid5000], where dedicated dark fibre links interconnect distant clusters and have bandwidths that are comparable to those commonly found on intra-cluster links (differing only in incurred latencies), current operating system research is focusing on the move from cluster-aware OSs to the next level, federation-aware OSs. Under the umbrella of the XtreemOS project [Xtreem] whose aim is to develop kernel extensions to provide for large scale SSI computing systems, the Kerrighed OS is being extended with contributions from the

PARIS Research Project [PARIS] (*e.g.* integrating a mechanism to support checkpoint/restart in the kernel) to operate seamlessly and efficiently in cluster federations.

A different approach, however, has been encouraged by the recent surge in virtual machine environments and proposes the use of virtualised resources (machines, networks, applications and data) to create virtual grids that run user applications across distributed environments. In--VIGO is one of these virtualisation based projects that advocates raising the level at which resources are "gridified"[11]: instead of dealing with concrete resources, In-VIGO middleware [Ada+05] deals with virtualised ones such as virtual machines, virtual (private) networks, and virtual data; for example, to provide for single sign-on, it decouples grid accounts from local accounts and then uses role-based access control lists to support user/resource access verification [Ada+04].

[11] A "gridified" resource is one that can be "shared among a dynamic collection of individuals, and institutions in a flexible, secure and coordinated way" *i.e.*, is a Grid resource.

Part III:

File Systems: Concepts and Performance

In this Part we discuss some fundamental concepts in file systems; we cover topics such as user-level views of file organization and access, sharing semantics and data consistency, and an array of techniques commonly used to enhance performance, such as data distribution and caching – starting from the perspective of a single-node computer and then moving to multiple node architectures. Each concept/technique is illustrated with a "real world" file system.

7 File Systems: Concepts and Performance

7.1 File systems

File systems are to computer systems what filing systems once were to archiving rooms: filing systems were used to organise records into files and folders, and these into cabinets; today's computer-based file systems organise files and directories in a tree-like structure whose root is usually contained within the bounds of a single logical volume, which we call a file system instance, or filesystem, for short.

7.2 File organisation and access methods

The sequential file is the most widespread file organisation model, one that reflects the earliest storage medium – the magnetic tape: an open operation will position the tape's begin-of-tape mark over the unit's head; then, a read command will scan through the tape, reading a record, and movement will stop at the next inter-record gap; each read scans towards the tape's end, and no more reads may be issued when the end-of-tape mark is over the unit's head. The disk based file system's analogy for this behaviour is to define a *sequential* file and a *file pointer* which is located at a particular offset (*e.g.*, zero on open), incremented after each successful read or write, or positioned with a seek operation.

Logical file *organization* deals with the file's logical structure: a file may hold either structured (fixed, variable length records, etc.), or unstructured (byte-stream) data; a sequential file, *i.e.*, a "file without holes", mimics a tape and contrasts with a sparse file, where "holes" may exist between regions which contain data. *Access methods* specify how one may access, i.e., read or write data to the file; for example, one can read data "forward" starting from a given *offset* in a logically contiguous file, while one cannot (always) do that in a sparse file; conversely, for indexed files, a key must specified prior to retrieving the corresponding data.

7.2.1 The UNIX heritage

UNIX popularised the byte-stream (unstructured) sequential and sparse file organisations, and a very simple file access API, consisting of five major primitives: open, close, read, write and seek[1]; as a consequence, both the sequential and sparse file organisations are supported today by the majority of local, as well as distributed file systems. Sequential files, being one of the simplest forms of storing data, are used both for persistent data storage and as a mechanism for data interchange.

[1] Unfortunately, in some texts, adherence to this set of five primitives is all that it takes for them to say that a given file system has (or has not) a POSIX API.

7.2.2 Business applications and file I/O

UNIX file organisations, access methods and primitive operations, although extremely powerful (and thus capable of being the building blocks for other, more complex file organisations and access methods), are quite detached from the needs of the typical business application developer. For example, business applications usually require advanced data structures and file organisations such as keyed, indexed sequential (ISAM), or even a fully fledged DBMS; these may be the ones needed to narrow the semantic gap between the (user's) problem and tools available to application developers.

7.2.3 Scientific applications and file I/O

Scientific applications commonly use data organisation and access methods that are quite different from those appropriate to other fields such as business, multimedia, etc. Regarding the amount of data accessed, business applications typically use large numbers of "data sources", be they files or tables (when using a DBMS); for each request, several files are accessed, but the amount of data moved to the application is usually quite small: a few "records" per accessed file[2]. Conversely, archetypal scientific codes use few but very large data files whose contents are, at first, fully (as much as one can fit) loaded into memory, in an I/O burst; then a sizeable amount of time is spent computing – a compute burst; finally, a large amount of data is written out, in another long I/O burst; of course, variations do exist, such as problems which require almost no input data, or others that do not generate much output, while some of them use temporary scratch files for debugging or out-of-core data.

7.2.3.1 Data storage vs. data distribution

Another big difference between scientific and business codes relates to file sharing: in business applications, several concurrently executing processes share data – they read/modify/ /write - and guarantee consistency through the use of file locking, while in scientific codes different processes usually access distinct, non-overlapping regions of a file, thus requiring no locking – in principle. Multi-process scientific codes may use files both as a way to store data and as a mechanism for data distribution; to illustrate this point we resort to a commonly used data structure, the 3D array (where each square is an **`n x m`** data block).

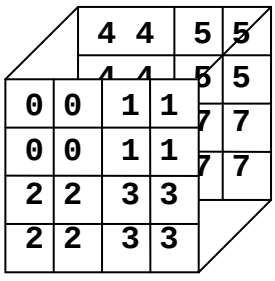


Figure 7.1 Logical view of a 3D array

[2] There are, off course, exceptions: Data Wharehousing and Data Mining spring to mind...

Supposing that we're executing a single-process application, the in-core (memory) image of the data array would be, for a row-major layout (*e.g.*, C codes),

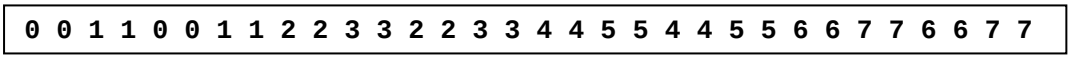


Figure 7.2 In-core row-major layout of a 3D array

while for a column-major layout (*e.g.*, FORTRAN codes) it would be,

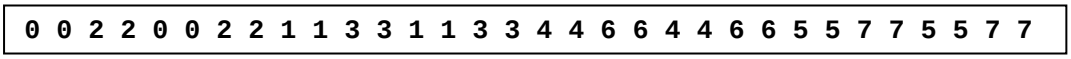


Figure 7.3 In-core column-major layout of a 3D array

Now, suppose that we wanted to store the in-core array out onto a file; in a C program we could accomplish it with a single[3] call such as `write(fd, array, size)`, one that writes the data out on a file; that would create a (sequential) file layout similar to the in-core layout of Fig. 7.2; the physical on-disk layout would, of course, depend on several things such as the file system itself, and whether a simple disk or an array of disks is used, etc.

Now, suppose that we do have eight processes, labelled from 0 to 7, each one holding in memory only those data blocks whose label is equal to the process number, and that we decide write them to a single file; among several possible layouts for the file, we highlight the following three [Mad+04]:

a) The canonical 3D block distribution, either in row-major or column-major order, just like those in Figs. 7.2 and 7.3 (although these figures were sketched to show in-core, not file data layouts). Notice that the algorithm for their creation cannot be as easily specified as, say, those for (b) and (c) below.

b) The 1D block distribution (Fig. 7.4), created by, *e.g.* sequentially executing the following: each process (starting with the lowest numbered one and then proceeding to the next in sequence) writes all its data onto the file, and then yields to the next process (which picks the file pointer offset left from the previous one, and continues writing),

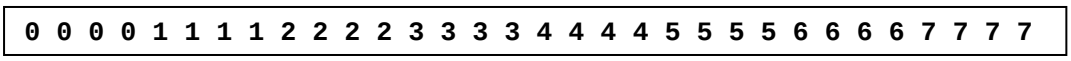


Figure 7.4 File layout of a 1D block distributed array

c) The interleaved sequential block distribution; again, a possible sequential algorithm for this layout is: each process (starting with the lowest numbered one and then proceeding to the next in sequence) writes its first block onto the file, and yields to the next, until the first

[3] Assuming that `size` is within the allowed bounds for the OS/file system call; otherwise, multiple calls would be used.

set of blocks for all the nodes have been written; then, each process proceeds to write the second block, etc.

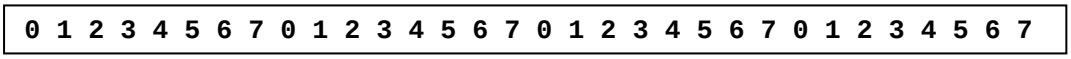


Figure 7.5 File layout of a sequential block distributed array

Block distribution algorithms (b) and (c) laid out above were specified in terms of a *shared file pointer* – when a process issues an I/O operation that moves the file pointer, other processes will "immediately" see the file pointer's new value; we could have use private file pointers instead, and the `lseek()` call; for example, the (b) 1D block distribution algorithm requires each process to seek to a location computed as `procID*wholeDataSize`, and then write its whole data chunk onto the file; but, for the interleaved sequential block distribution in (c), the algorithm using private file pointers now becomes more complex, as each process loops until done, successively seeking to file locations computed by `procID*DataSize+cnt*nbrOfProcs`, writing a portion of data onto the file.

After a file layout has been decided, and the file stored on disk, sometimes things change; for example, the number of processors may be changed (*e.g.*, more processors were bought) thus benefiting from an increase in the number of processes, which then leads to a different data distribution; or, some obscure bug must be sorted out by resorting to a single-process sequential execution. In any case, we now must resort to a different algorithm for loading the array and, if we want to cover "all" possible cases, the code may become confusing and inefficient. This is why access methods start incorporating the notion of views that "hide" the offset between successive data blocks to each process' eyes, making them look contiguous, such as strides for sequential files, or more sophisticated file organisation models, such as the sub-files in Galley [Nie+96] and Vesta [Cor+96] parallel file systems.

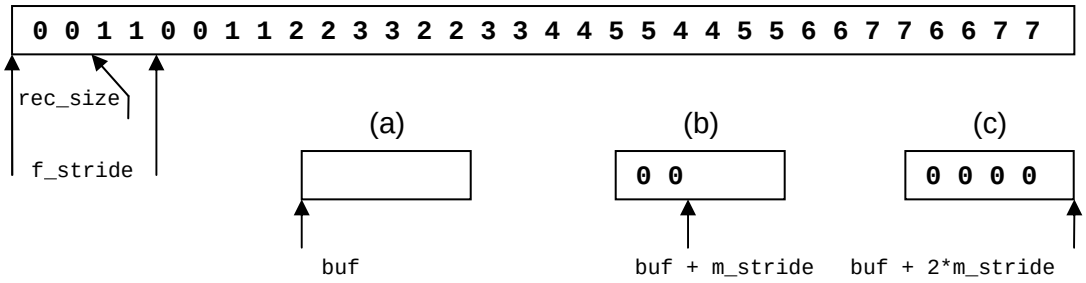
7.2.3.2 Closing the semantic gap in scientific applications

Strided access is very common in parallel applications: in a strided operation, several non-contiguous data chunks within a file are accessed; for example, in the Galley file system, to distribute data from a file with a canonical row-major layout to each of the eight processes, each one would perform (assuming that each array block, i.e., each "square" in Fig. 7.1, occupies 1024 bytes)

```
f_stride= 4096; m_stride= 2048; rec_size= 2048; quant= 2;
offset= procID*f_stride + (even(procID)?0:-rec_size);
gfs_read_strided(fid, *buf, offset, rec_size, f_stride, m_stride, quant);
```

Beginning at `offset`, the file system will read `quant` records, of `rec_size` bytes each. The offset of each record is `f_stride` bytes greater than that of the previous record; records

are stored in memory beginning at `buf`, and the offset into the buffer is changed by `m_stride` bytes after each record is transferred. When `m_stride` is equal to `rec_size`, data will be gathered from disk, and stored contiguously in memory. When `f_stride` is equal to `rec_size`, data will be read from a contiguous region of a file, and scattered in memory. It is also possible for both `m_stride` and `f_stride` to be different than `rec_size`, and possibly different than each other. Galley also allows us to express more complex access patterns, in the form of nested strides, and to organise data into sub-files.



Buffer filling: (a) before reading; (b) quant=1 read; (c) quant=2 read.

Figure 7.6 Process 0 getting its data from a 3D array stored in a file.

The MPI-IO approach to the data partitioning problem [Cor+02] is to define an elementary data type, `etype`, that contains the user "record" type structure, a `buftype` which describes the arrangement of `etype` elements into an application buffer, and a `filetype` which describes how `etype`s are laid out onto a file.

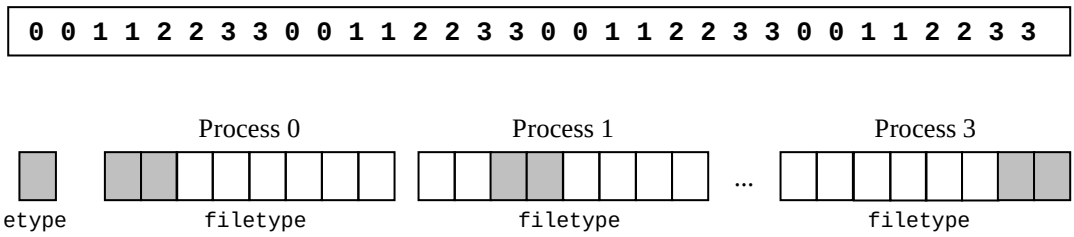


Figure 7.7 MPI-IO data partitioning.

7.3 Delivering high performance

Delivering powerful abstractions that ease the developer's burden by narrowing the gap between the problem space and its implementation is an important step, but it's not the only one; a file system must also deliver good performance. We will now look at some of the options available to tackle the filesystem performance problem.

7.3.1 Enhancing the file API for efficient data access

It is well known that local file systems do a better job when serving a small number of large-sized requests than when they have to serve large numbers of small requests; for

example, Linux places I/O requests in a queue where they are, if possible, coalesced with already existing requests before being submitted to the disk controller [Bov+05]. The reasons behind this performance increase are twofold: as each disk access experiences rotational and seek latencies which dominate *vis-à-vis* data transfer times, by submitting fewer requests we hopefully get better throughput; and, by submitting fewer requests we spend less time in system call processing, queue processing, programming DMA engines, responding to interrupts, etc., thus decreasing CPU usage (for a more in-depth coverage, see section 8, "The case for Caching in a Local File System").

APIs that allow programmers to submit fewer requests, such as the one for strided accesses shown in Fig. 7.6, or the POSIX API [IEEE04] for vectorised I/O (which allows a contiguous file region to be scattered into/gathered from non-contiguous memory locations with `readv` and `writev()`calls) and list I/O (which allow non-contiguous file regions to be accessed with a single `lio_listio()` call) are quite important, as they provide information to the file system that enables it to perform optimisations that can deliver better application performance.

7.3.2 Parallel access through data distribution

We have seen how advanced logical file structures and/or file access methods can be used to better map the problem-domain to the underlying data storage, or to convey to the file systems information on application access patterns, in order to increase their performance.

Another way to increase performance is through the use of parallelism: if we are able to distribute data across multiple disks in a way that, to fulfil a single I/O request, we have to access several disks in parallel, we may expect a performance increase due to the higher aggregated bandwidth. In the next subsections we will discuss two ways of distributing data across disks: one distributes data to multiple disks attached to a single computer system, while the other distributes data to disks hosted onto distinct, interconnected computers.

7.3.2.1 Scaling in: intra-node data distribution

Data distribution at the device level is implemented by resorting to multiple disks and "scattering", or de-clustering, data over them; this approach obviously increases bandwidth, by as much as the aggregated bandwidth of the disks "activated" in parallel to fulfil a single I/O request, and is applicable to single-node computers.

Hardware-based solutions call for RAID-capable processors installed either internally, in the host, or externally, in disk array boxes while software-based solutions are provided by logical volume managers (LVM) or software-RAID (Linux' `md` device driver) modules; both offer a set of choices as the RAID level to use. Usually, levels 0 or 5 (or "combined" ones, such as 0/1) are used to create a "virtual disk" which is, to the disk driver or file system layer, completely undistinguishable from a "real device"; parameters, such as stripe size and width,

for the RAID device are usually chosen to optimise a specific item (*e.g.*, application reads), as it is quite difficult to optimise everything – *e.g.*, as the file system is unaware of striping, it lays out its metadata structures (*e.g.*, superblocks, bitmaps, inodes, etc. – see Part VI) over the virtual disk just as it would do on a physical device, unaware of its "real" geometry.

Although we have not found any existing implementation of a local file system that supports data de-clustering on a per file basis, instead of per volume, there is no obstacle to building one; we think that reasons why such a feature is not available in local file systems may relate to their general-purpose nature.

7.3.2.2 Scaling out: inter-node data distribution

If a computer system, large as it may be, reaches its configuration limits on a resource, one has two options: a) replace it with a "bigger" model; b) keep it and add one more, connecting both together and using them in "parallel", hopefully solving the problem.

Using several interconnected hosts is thus another way to overcome the shortage of I/O bandwidth, as each computer gets its own set of disks (internal or external); to be beneficial, *i.e.*, to deliver increased performance, several things must happen: first, data will have to be de-clustered across the various server nodes and their respective disks in such a way that a single I/O request, *e.g.* a read, issued to the "server group" must be processed by several (if not all) hosts, which will respond by accessing their own disks, delivering the data over the interconnect; second, the interconnect must not become the new bottleneck – we do not want to be replacing one problem with another; and finally, there must be a measurable gain, otherwise we may be offering some sort of file sharing server, but not the high performance I/O system we were aiming at.

The use of several hosts in parallel to act as data storage servers requires a distributed file system to integrate and coordinate clients and servers; parallel file systems are a subclass of distributed file systems whose main target is performance: they support data de-clustering, either at the filesystem level, as in PVFS and GPFS[4], or at a finer grain, as in Vesta, which is able to de-cluster at the file level.

7.3.3 Caching for high performance

No modern computer systems can do without caching. Caches have found their way in from processors to disk controllers, from file systems to database engines, to web servers, etc. From the point of view that interest us, caching is used both by local and distributed file systems, although the later sometimes restrict their usage (*e.g.*, PVFS only uses server caches) having balanced the cost of maintaining cache coherence against the benefits it provides in their target environments. Caching is very important, and we devote the next two sections on

[4] PVFS and GPFS are described in detail later on; as for PVFS, it is constantly evolving, and version 2.7 (end of 2007) allows finer grained data de-clustering.

it: in the first one we analyse the benefits of caching in local file systems, while in the next we apply the same reasoning to caching in distributed file systems.

7.4 Closing remarks

In this section, we have presented some important topics on file systems. First, the adequacy of file organisation and access modes to real world problems, where we found that the POSIX sequential and sparse models are insufficient for some domains, such as business (where ISAM and DBMS are the answers) and scientific (where the five primitives are not enough, but vectorised and list I/O may help). Then we raised the issue of performance; to get good performance out of a file system several techniques must be used: suitable APIs that allow the programmer to convey to the file system information that enables it to optimise data access; parallelism, with data de-clustering over multiple disks and/or multiple servers, together with an adequate interconnection infrastructure; and, last but not least, caching.

8 The Case for Caching in a Local File System

8.1 A simple performance model

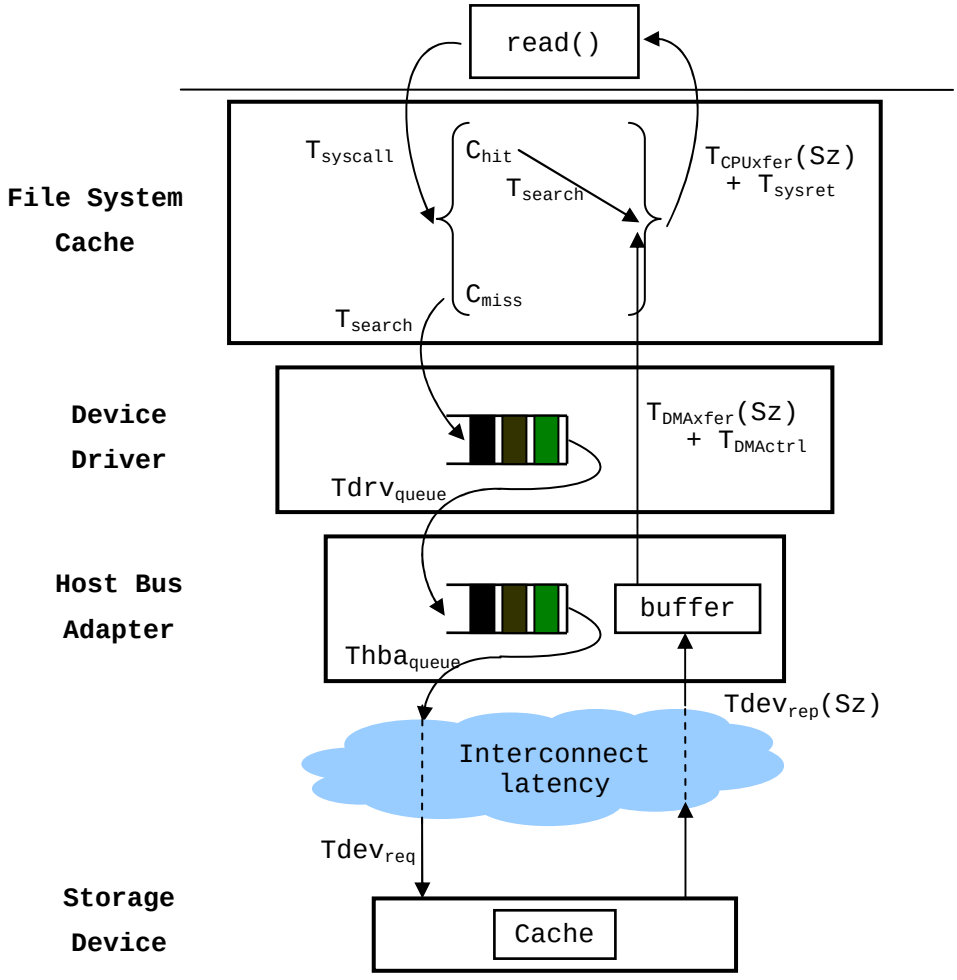


Figure 8.1 Contributors to latency and bandwidth on a read() call.

Computer systems have been using both hardware and software based buffers and caches to speedup tasks; Fig. 8.1 depicts the cache hierarchy commonly found in the I/O path on a local file system implementation, along with tags highlighting each one of the major contributors to the latency of a read request. A short description of each tag follows:

$T_{syscall}$, T_{sysret}	Time spent executing a system call, which involves a transition from user to kernel mode, and back.
T_{search}	Time spent searching a cache for a matching item (block, page, etc.).
$Tdrv_{queue}$	Time spent by a request in the driver's queue, waiting to be submitted to the adapter (HBA) or device controller.
$Thba_{queue}$	Time spent by a request in the HBA/controller queue, waiting to be submitted to the device.
$Tdev_{req}$	Time spent to transfer the request packet from the HBA, through the device interconnection network, to the device; for simplicity, we assume that issuing a device read involves just one packet.
$Tdev_{rep}$	Time spent to transfer data (we are reading) from the device to the HBA, across the interconnection network; for simplicity, we assume the response as a header plus data (so the time actually depends on the data size).
$T_{DMActrl}$	Time spent to program the DMA engine at the beginning and at the end of the data transfer.
$T_{DMAxfer}$	Time spent by the DMA engine to copy the data to the cache.
$T_{CPUxfer}$	Time spent by the CPU to copy the amount of data requested by the user from the cache to the user buffer.

A short description of the shortest path, *i.e.*, one where data is already in cache, for a user read, is as follows: $T_{syscall}$ will be the overhead of entering the read system call in kernel space; after a cache lookup that takes us T_{search}, we have a hit and copy the requested amount of data to user space in $T_{CPUxfer}(Sz)$ time; finally we return from the system call in T_{sysret}.

To get the breakdown for the `write()`, we just need to swap the roles of $Tdev_{req}$ and $Tdev_{rep}$, where $Tdev_{req}$ will carry the overhead plus data, and $Tdev_{rep}$ will be just an acknowledge packet.

8.2 Peak bandwidth

We want to compute approximate values for the bandwidths we can experience if, on a read, we: get the data from the file system cache ($BW_{fromCache}$); use direct I/O to bypass the file system (FS) cache, and move it straight to the user buffer ($BW_{DirectIO}$); and, go through the cache but end up fetching the data from the device ($BW_{DiskThruCache}$). The first thing we'll do is to identify values that are so small that they do not contribute much to the overall result; for

current COTS server hardware, $T_{syscall}$, T_{sysret} and $T_{DMActrl}$ take tens to hundreds nanoseconds and, as all the other values are in the micro to millisecond range, we'll ignore them. As we're aiming for a peak value, we will assume no requests are pending on the queues, so we'll also set $Tdrv_{queue}$ and $Thba_{queue}$ to zero. Therefore,

$$BW_{fromCache} = Sz/T_{CPUxfer}(Sz) \quad (8.1)$$

$$BW_{DirectIO} = Sz/[T_{DMAxfer}(Sz)+Tdev_{req}+Tdev_{rep}(Sz)] \quad (8.2)$$

$$BW_{DiskThruCache} = Sz/[T_{CPUxfer}(Sz)+T_{DMAxfer}(Sz)+Tdev_{req}+Tdev_{rep}(Sz)] \quad (8.3)$$

Expression (8.1) shows that $BW_{fromCache}$ depends only on CPU speed and memory bandwidth, and not on devices, as expected; in the lab servers used for this work[1], the peak value for memory bandwidth is 6.3 GB/s, while the sustained value we get from the STREAM benchmark [McC95] is in the 1.6 to 2 GB/s range.

Expression (8.2) shows that, if the device is to be accessed, peak I/O is reached when resorting to Direct I/O, *i.e.*, bypassing the FS cache; data still has to flow through the system's I/O bus, pumped by the DMA engine in the HBA. In today's small servers, I/O busses have bandwidths of 1.6 GB/s for 4x PCI-e or 1066 MB/s for PCI-X (at 133 MHz and a 64-bit bus) [IBM-07], which clearly shows that I/O bus bandwidth is adequate. The expression,

$$BW_{fromDevice} = Sz/[\ Tdev_{req}+Tdev_{rep}(Sz)] \quad (8.4)$$

computes the device transfer rate; even today's medium sized FC disk arrays from such companies as EMC, HP and IBM have several GB of cache and deliver aggregate transfer rates in excess of 1GB/s across multiple fibre links at 100 to 800MB/s per FC port.

Finally, expression (8.3) highlights the extra copy operation – from the kernel cache to the user buffer – that contributes to a slightly lower performance of (8.3) *vis-à-vis* (8.2).

8.3 Latency and sustained bandwidth

Given that peak bandwidth is quite adequate, we must look closely at expression (8.4), for the transfer rate of an I/O device and, along with Fig. 8.2, get a better understanding of what contributes to a sustained bandwidth.

The time taken by the request packet, issued by the HBA, to arrive at the I/O device can be approximated as $Tdev_{req} \approx Link_{BW}/Req_{sz}$; likewise, the time spent to transfer the data is $Link_{BW}/Data_{sz}$. The "processing delay", as tagged in the figure, is the time spent by the device to make the data available to be transferred; it may be insignificant if the data is available on the device cache and can be quickly located but, if not cached, may become quite important, as it could take about 5 ms even for fast (10K rpm) SCSI or FC disks.

[1] See Part VIII, "Benchmarking pCFS".

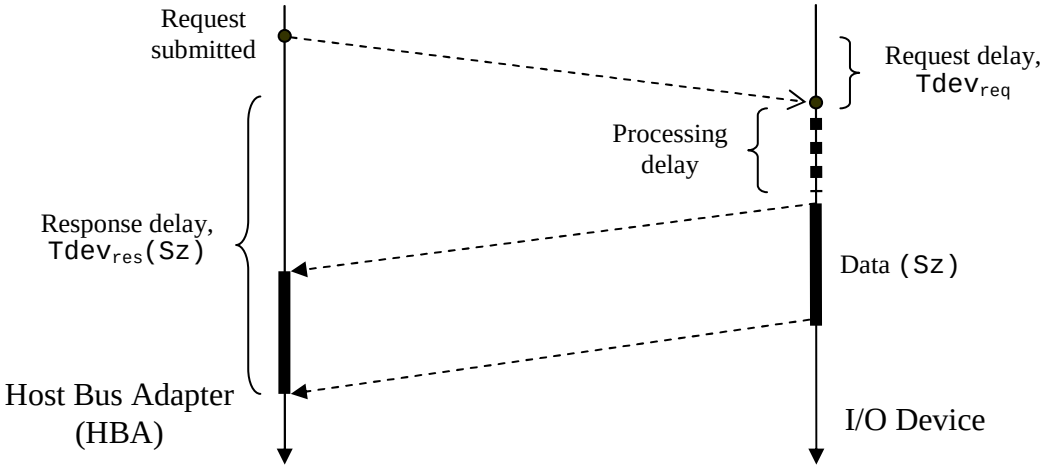


Figure 8.2 Contributors to latency and sustained bandwidth

Assuming a 16 byte request packet, a 200 MB/s FC link, a 5 ms processing delay per packet (*i.e.*, no cache at the device) and a "device" capable to sustain the full 200 MB/s transfer rate – *i.e.*, not a single disk but, *e.g.*, a RAID-0 volume of 5 disks with a sustained 40 MB/s per disk – we compute the sustained bandwidth as,

$$SustainedBW_{fromDevice} = Sz/[5*10^{-3} + (16+Sz)/200*10^{6}] \quad (8.5)$$

For blocks of sizes 1K, 8K, 64K and 512KB we get sustained bandwidths of 102K, 162K, 1230K and 68790 KB/s; so, our best case has a 34% use of the available bandwidth for one link only! Thus, as expected, *at the end of the I/O chain devices must have caches*; only then we will be able to exploit the full bandwidth of the I/O channel. But is it enough to have caches at the device, *i.e.*, at the end of the I/O chain? How large should they be? And for very large caches, is it still possible to perform a cache lookup in a few microseconds, or are we beginning to see the build up of another delay factor?

Today's large disk arrays "serve" not one but several hosts (in enterprise data centres some of these hosts have distinct architectures, and even run different operating systems) and they have really huge caches – up to 64 GB; understandably, the time to perform a cache lookup is now closer to the millisecond. If, in expression (8.5), we change the 5 ms value to 1 ms, our best case turns out to be 145 MB/s now, or 72 % of the single link bandwidth; but small-sized requests, such as the Linux default's page-sized 4KB I/O request, still use the bandwidth very poorly at 1ms, with 20% usage.

All the above intuitively[2] reinforces the belief that, even with the today's high performance infrastructures at the end of the chain, we still need a *host-based cache* if we want to provide high sustained bandwidths and faster response times to applications. While on this subject,

[2] A detailed study would be very long and complex, and is therefore outside the scope of this work.

some authors argue that caches everywhere (on disks, arrays, hosts, etc.) are not always beneficial, as, if not accounting for anything else, they surely are expensive [Won+02].

8.4 CPU use in I/O operations

Given that all HBAs worth considering are DMA-capable, and the fraction of CPU spent in programming the DMA is negligible when compared to amount of CPU needed to perform a copy from the VFS cache into the application buffer (assuming cached I/O), we can easily compute an approximate value for the fraction of CPU needed in a full I/O transfer as follows: let's assume that 100% of CPU is consumed in our lab server to perform a memory copy at 2 GB/s in the STREAM benchmark; then, to move data from the cache to the user buffer at 100 MB/s (our server's maximum FC rate), we would wear out 5% of the host CPU.

8.5 The benefits of caching

We conclude that caching and pre-fetching are both important to local file systems: reads and writes hitting the FS cache experience the memory subsystem bandwidth and latency; pre-fetching, for reads, as well as write-combining (whose role was not discussed in this section), for writes, both deliver higher I/O subsystem bandwidths as they batch smaller requests together into fewer I/O operations with larger sized "blocks"; some decrease in the CPU load, resulting from a smaller number of I/O operations, may also be expected as a consequence of fewer interrupts, less context switching, etc.

9 The Case for Caching in a Distributed File System

9.1 A simple performance model

For file systems, such as NFS, that access remote data over a network, Fig. 9.1 illustrates the contribution of each major step to the latency of a read request when reading a file from a remote server. A quick look shows a great resemblance with Fig. 8.1; new are the NFS client module and an "upper" network software layer (which includes the remote procedure call – RPC – and external data representation – XDR – layers), and the TCP/IP stack. Notice the much referenced double buffering/copy problem: from the NIC, data is DMA moved to a network buffer, where it may be moved around (for packet reassembly, format "translation"), then copied by the host CPU to the OS cache (in a best-case, our 1st copy) and finally from there to the user buffer (2nd copy). After some research proved the feasibility and superiority of a zero-copy approach in the network stack [Pai+99, Wu+04], NFS releases bundled with Linux 2.6 versions do profit from it.

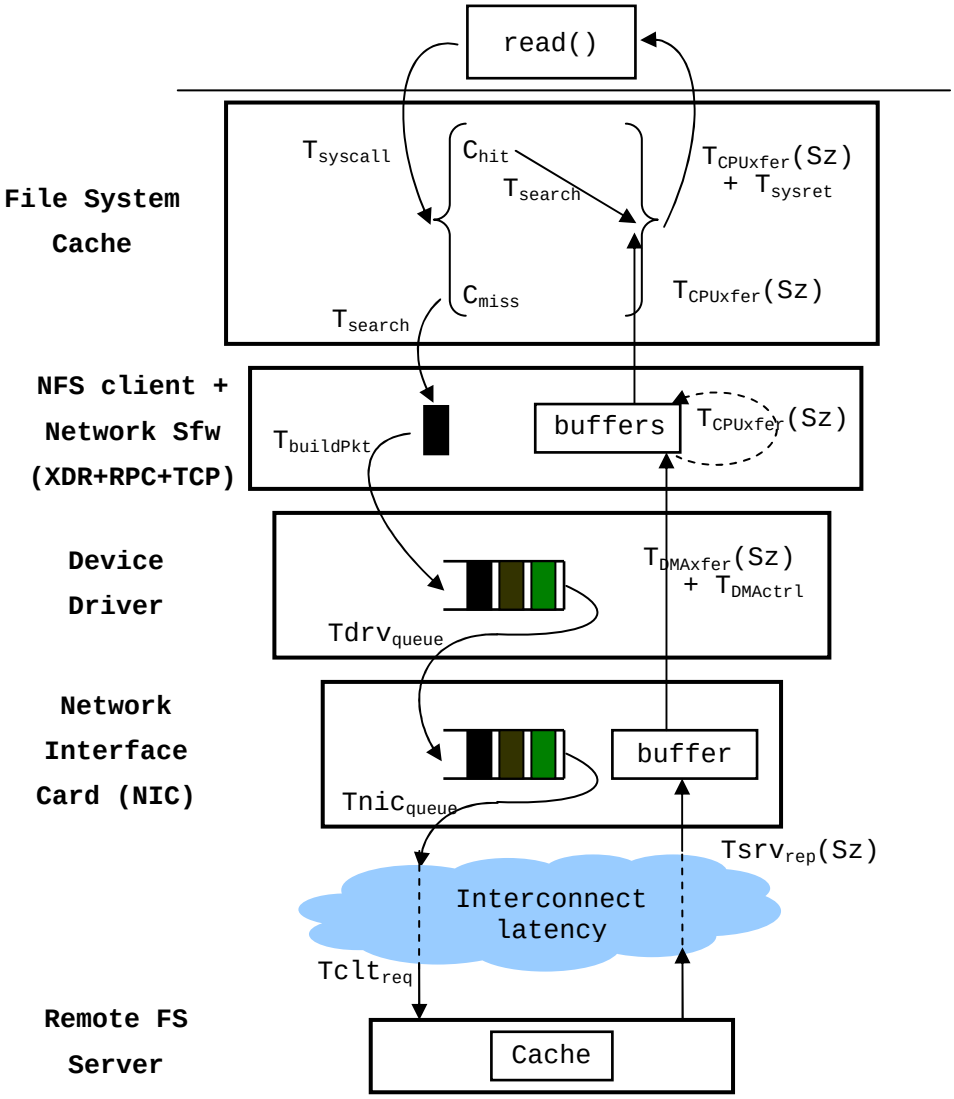


Figure 9.1 read() call flow on a NFS client.

A short description for each tag follows:

$T_{syscall}$, T_{sysret}	Time spent executing a system call, which involves a transition from user to kernel mode, and back.
T_{search}	Time spent searching a cache for a matching item (block, page, etc.).
$T_{buildPkt}$	Time spent to build, in the NFS client layer, a server request packet and submit it through the XDR/RPC, TCP and IP layers.
$Tdrv_{queue}$	Time spent by a request in the driver's queue, waiting to be submitted to the network interface card (NIC).
$Tnic_{queue}$	Time spent by a request in the NIC's queue, waiting to be sent out to the network.
$Tclt_{req}$	Time spent to transfer the request packet from the NIC, through the interconnection network, to the file server; for simplicity, we assume that issuing a read involves just one packet.

$Tsrv_{rep}$	Time spent by the server to answer the client's request and deliver the data (we are reading) across the interconnection network; for simplicity, we assume the response as a header plus data (so the time also depends on the data size).
$T_{DMActrl}$	Time spent to program the NIC DMA engine at the beginning and at the end of the data transfer.
$T_{DMAxfer}$	Time spent by the NIC DMA engine to copy the data to the network layer buffers.
$T_{CPUxfer}$	Time spent by the CPU to perform a memory copy; for simplicity we assume that the size of the data moved between the network buffer and the cache is equal to the amount requested by the user, moved from the cache to the user buffer.

9.2 The case for server-side caching

The read bandwidth, as perceived by the client, may be approximated by

$$BW_{fromNetwork} = Sz/[2*T_{CPUxfer}(Sz)+T_{DMAxfer}(Sz)+Tclt_{req}+Tsrv_{rep}(Sz)] \quad (9.1)$$

which shows up the double copy (from the network stack to the page cache and from there to the user buffer), the client's request transfer delay and the contribution of the file server, which we will now break down.

To break down the server's contribution, we will assume the same simplifications made before for the client. The first part is essentially the reverse route of the client's traffic: the client request is received at the NIC and "migrates" up the software layers, to the NFS layer, at an oversimplified zero cost; then, the NFS server will search in its cache for the requested data, at a T_{search} cost (which, to be coherent, should also be dismissed because it will be smaller than the network stack cost we've just ignored). At the server, the time to find the data is either T_{search}, if data is cached, or T_{search} plus the time to access the local storage, *i.e.*, plus the denominator from (8.3); but, as we've done for it we'll also omit the search time now, so (9.3) will be the same as (8.3),

$$T_{dataCached} = T_{search} \quad (9.2)$$

$$T_{dataUncached} = T_{CPUxfer}(Sz)+T_{DMAxfer}(Sz)+Tdev_{req}+Tdev_{rep}(Sz) \quad (9.3)$$

We can now proceed to compute the time required for the complete response, which will include the copy to the server's network buffer and the time to transfer it to the client's NIC; if we call the network latency $L_{net}(s)$, to denote it as a function of the packet's size,

$$Tsrv_{repCached} = L_{net}(Sz)+T_{CPUxfer}(Sz)+T_{dataCached} \quad (9.4)$$

$$Tsrv_{repUncached} = L_{net}(Sz)+T_{CPUxfer}(Sz)+T_{dataUncached} \quad (9.5)$$

We can thus expand (9.1) to its final form (renaming it to $BW_{fromSrvCache}$ to indicate that data is fetched from the NFS server cache)

$$BW_{fromSrvCache} = Sz/[3*T_{CPUxfer}(Sz) + 2*T_{DMAxfer}(Sz) + L_{net}(Clt_{pkt}) + \\ + L_{net}(Sz) + T_{dataCached}] \quad (9.6)$$

i.e., in a NFS environment without client caches (but with server caching) we may clearly see that: a) aggregated CPU usage is, at least, three times (we will see later that, in practice, it is much more) in a NFS client than in a local file system client; and, b) client bandwidth is degraded with respect to the network bandwidth by the amount $L_{net}(Clt_{pkt})$. From several sources, *e.g.* [Hug+05], we get round-trip latencies for Gigabit Ethernet (GbE) of circa 25 µs for small UDP packets (*i.e.*, client requests) and 12 µs to transfer a 1500 bytes frame (which we will assume carries a 1400 bytes payload); so we compute a maximum of

$$BW_{fromSrvCache} = 1400*8/(12.5+12) = 457 \text{ Mb/s or } 57 \text{ MB/s} \quad (9.7)$$

an utilisation of about 50% of the GbE bandwidth (in fact, much worse if all the contributions that we have discarded were brought in).

9.3 The case for client-side caching

Instead of using a 1-by-1 request/reply pattern with the limited size of an Ethernet packet, such as in (9.7) above, we could use larger requests, in an effort to profit from the TCP streaming capabilities, *e.g.*, using a TCP segment of 32KB for the reply (about 22 packets at 1460 bytes each). Then, we would get

$$BW_{fromSrvCache} = 22*1400*8/(12.5+22*12) = 891 \text{ Mb/s or } 111 \text{ MB/s} \quad (9.8)$$

Expression (9.7) unequivocally shows that, even using a NFS server that caches data in its memory, a NFS client using a request/reply read pattern where only the exact amount of data required by the application is transferred, performs very badly if that amount is less than, say, one full-length Ethernet packet. *To get suitable performance, one must use read-ahead and caching at the client.* The value predicted in (9.8) is very close to single-client bandwidth with NAS appliances (from companies such as EMC or NetApp) using high performance NICs designated TOEs, TCP Offload Engine boards, and heavily tuned software (lightweight kernels and network stacks). Small computers acting as NFS servers usually cannot provide such levels of bandwidth, due to the overheads of TCP/IP, OS kernel, file system and I/O devices; a typical value for sustained bandwidth in small NFS servers is around 30 MB/s, but they are nevertheless able to deliver similar peak values when accessing cached data.

At the client side, for cached data, we have

$$BW_{fromCache} = Sz/T_{CPUxfer}(Sz) \quad (9.9)$$

Clearly, bandwidth from the client's cache is the same as (8.1) – obviously there is no difference in the bandwidth delivered by a local or a remote file system to its "clients" if data is located in the host's cache

9.4 CPU use in remote I/O operations

Network I/O at full bandwidth using either a completely dumb (CRC performed on-board and checksuming performed by the host CPU) or even a "medium-smart" NIC (capable of on-board CRC and checksum processing) consumes a sizeable amount of CPU; in our test infrastructure we have measured around 40% of CPU usage with a 2.6 GHz Xeon and regular-sized 1500 bytes frames, and about 30% with 9000 bytes Jumbo frames to keep a Broadcom GbE NIC at 80MB/s [Lop+05]. Therefore, I/O in distributed file systems may require, depending on the "intelligence" of the NICs used, quite more CPU power than the corresponding operations in local file systems.

9.5 The benefits of caching

We have shown that both caching and pre-fetching (or read-ahead) are important to remote file systems and that they should be performed at the client as well as in the server; reads and writes hitting the client's cache experience its memory subsystem bandwidth and latency, while those hitting the server's cache experience the network bandwidth and latency; pre-fetching, for reads, as well as write-combining (whose role was not discussed in this section), for writes, allow requests to be batched into fewer I/O operations on larger sized "blocks" and thus require fewer request/reply packets, resulting in higher network utilisation (bandwidth) as well as a much reduced CPU load, a consequence from the decrease in the number of interrupts raised by the NIC.

10 Caching and Sharing in Local File Systems

Modern, widely used local file systems may offer similar basic characteristics but are usually quite different from each other when it comes to "advanced" features such as fault tolerance and resilience, time to recover from failures, and performance. Most have adopted the UNIX file organization model, which supports both (logically) contiguous and sparse files, as well as distinct access modes, including sequential and random; they eschewed the record-based file model in favor of the byte-stream "unstructured" approach, relegating more complex organization and access modes such as keyed, ISAM, etc., to application libraries and DBMS systems.

Local file systems make extensive use of both data and metadata caching to increase performance; some take even more steps, such as trying to predict the application's file access behavior and asynchronously reading data ahead, or batching together several reads (or

writes) together in order to minimize the number of I/O requests issued while performing larger, more productive, I/O transfers. In the next sections, we will show that defining the sharing semantics for a file system strongly limits the designer's choices on the caching subsystem architecture as well as on cache coherence policy options.

10.1 The page cache in modern operating systems

Modern operating systems, such as Windows and Linux[1] have benefited from research that was incorporated originally into SunOS 4 and showed the advantages of a unified page cache over two separated memory zones – a buffer-cache area for storing file system blocks and a page-cache area for storing program pages; those benefits include a cleaner (although more complex to implement) interface between the virtual memory and file management kernel subsystems (which simplifies the implementation of memory mapped files) and a unified approach to file access, independently of the file's "type" (e.g., program vs. data).

10.2 The file abstraction and the page cache

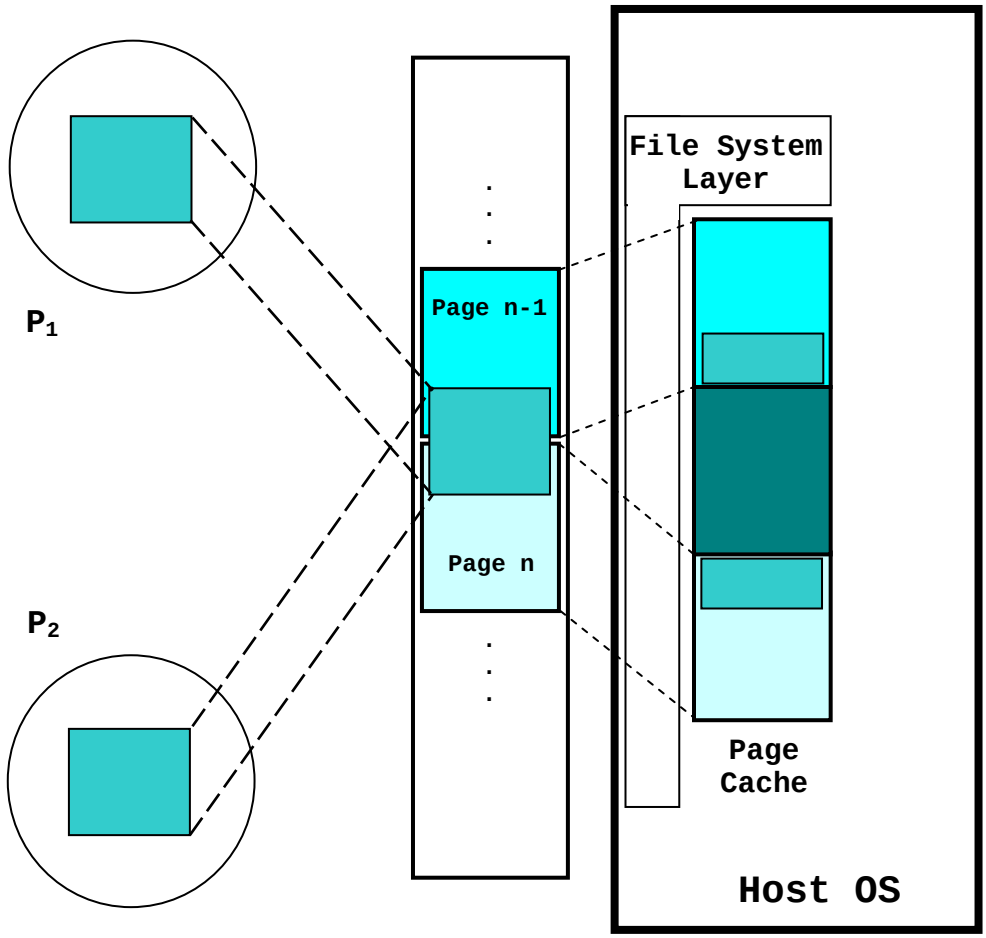


Figure 10.1 A file "image" is created from the page cache.

[1] From release 2.4.10 onwards.

As a side effect, the adoption of a Page Cache has created another mismatch (an interposed page layer) between the byte-stream file abstraction and the block structured I/O devices, resulting in a "cache line" increase from the size of a (device) block to a page. A page-sized cache motivates us to perform larger sized I/O operations when accessing the filesystem devices, and is even more beneficial if a) data is contiguously located, as only a single request needs to be issued, and b) if data pre-fetched[2] along the way will be used again in the (near) future. Figure 10.1 depicts a file, shown a sequence of pages; these pages contain the file data "records", and a record may be spanned across two (or more) pages. It also shows two processes accessing (again, we assume both are reading) the same record; and, despite being a complex data structure that stores pages from many distinct files, it still allows the upper layers in the file system to implement, at the API level, the various "file models" that applications expect, and depend on.

Definition 10.1: file view

A file view (FV) for some file f is the logical view (of the file) that we get when, at a moment t, we select all pages of f stored in the page cache:

$$FV(f) \equiv Page-Cache(f)$$

where *Page-Cache(f)* is the function that performs the "select" on the page cache, looking for pages that hold data from file f (and sorts them by page index).

When a process performs a read, only a subset of the pages in the FV is involved in the operation, i.e., those containing the "record" that must be copied to the process buffer; for example, in Fig. 10.1, the requested data spans pages n-1 and n.

Definition 10.2: request window

A request window is the smallest set of file pages (stored in the page cache) that satisfies a single, contiguous request of size r over a file (pointer) f:

$$W(f,r) \equiv \{P_i\} : i \in [start(f,r), end(f,r)]$$

where, *start(f,r) = f div PageSize,*
end(f,r) = start(f,r) + SizeofRequest(r) div PageSize

If the request is of the scatter/gather type, then the request window is clearly the union of the request windows for each contiguous sub-request.

Definition 10.3: overlapping requests

Let r and s be two requests made by distinct processes on a file f; they overlap *iff*

$$[start(f,r), end(f,r)] \cap [start(f,s), end(f,s)] \neq [\ ]$$

[2] As usual, we're using a read operation because it does a simpler job at illustrating the point.

Definition 10.4: overlapping request windows

Let r and s be two requests made by distinct processes on a file f; their request windows overlap *iff*

$$W(f,r) \cap W(f,s) \neq \{\ \}$$

False sharing arises when two requests do not overlap but their request windows do; it's a consequence of caching at a granularity level larger than the record itself.

10.3 Sharing: from the file system down to the file

An important feature of a file system is its *sharing semantics*, *i.e.*, how it behaves under concurrent access from user applications. In any modern file system there are two major "objects" as perceived by the users: files and directories. Files hold user data (and may hold other "data" too, such as source programs, executables, etc.) while directories organise files *e.g.*, into a tree. File system objects have associated metadata which holds information about the objects themselves; examples of file and directory metadata are timestamps (e.g., of creation, last access), size, ownership, etc.

10.3.1 File system sharing semantics

File system sharing semantics specifies how the *file system* itself behaves under concurrent operations that read and eventually update its own managed structures; it specifies, for example, the outcome of an execution where a process is reading a file while another is concurrently deletes it – as an example, Linux' ext2 allows the process to continue, even though the file entry is already missing from the directory and won't be seen by newer processes, while PVFS will return an error on the next operation issued by that process.

10.3.2 File Sharing Semantics

On the other hand, *file sharing* semantics specifies how a *file* behaves[3] when processes concurrently access it, with mixed operations that may read and write user data within the file itself (and, consequently, its metadata); it specifies, for example, the outcome of an execution whereby a process is reading a file section while other processes are concurrently writing to the same region – as an example, the ext2 file system nearly implements the so called POSIX file sharing semantics, as described in section 10.4.3 below.

10.4 Case study: caching in Linux

For the Linux operating system, the role of the relevant layers involved in a read or write operation from or to a disk device, is sketched in Fig. 10.2; it's just a starting point for our discussion (not the complete picture), but one that highlights the major aspects: it briefly shows that application I/O calls enter the kernel and are processed at the file system layer,

[3] This is a figure of speech; files do not "behave", of course...

where usually a cache lookup is performed to see if the desired data is already present in the cache; for example, if a `read()` is being executed and data is found in the cache, it is immediately moved to the application buffer; there's no need to access the disk device, here.

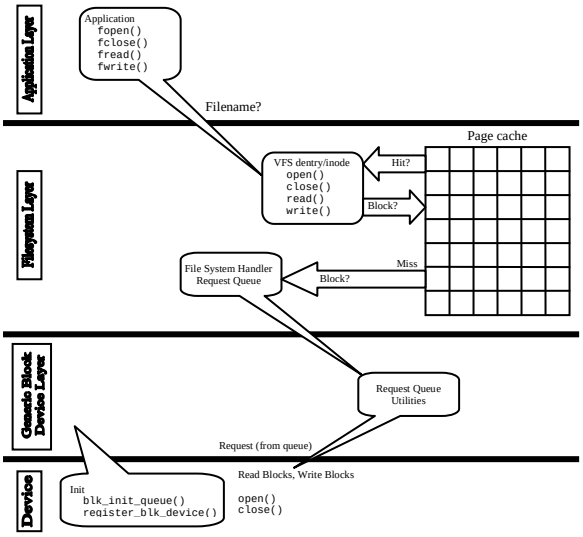


Figure 10.2 Architecture for file I/O in the Linux kernel (from [Rod+05])

10.4.1 The file system layer

The Linux file system layer is structured in two parts: an upper, Virtual File System (VFS) layer, and a lower file system-specific layer, where modules for each file system do "plug in".

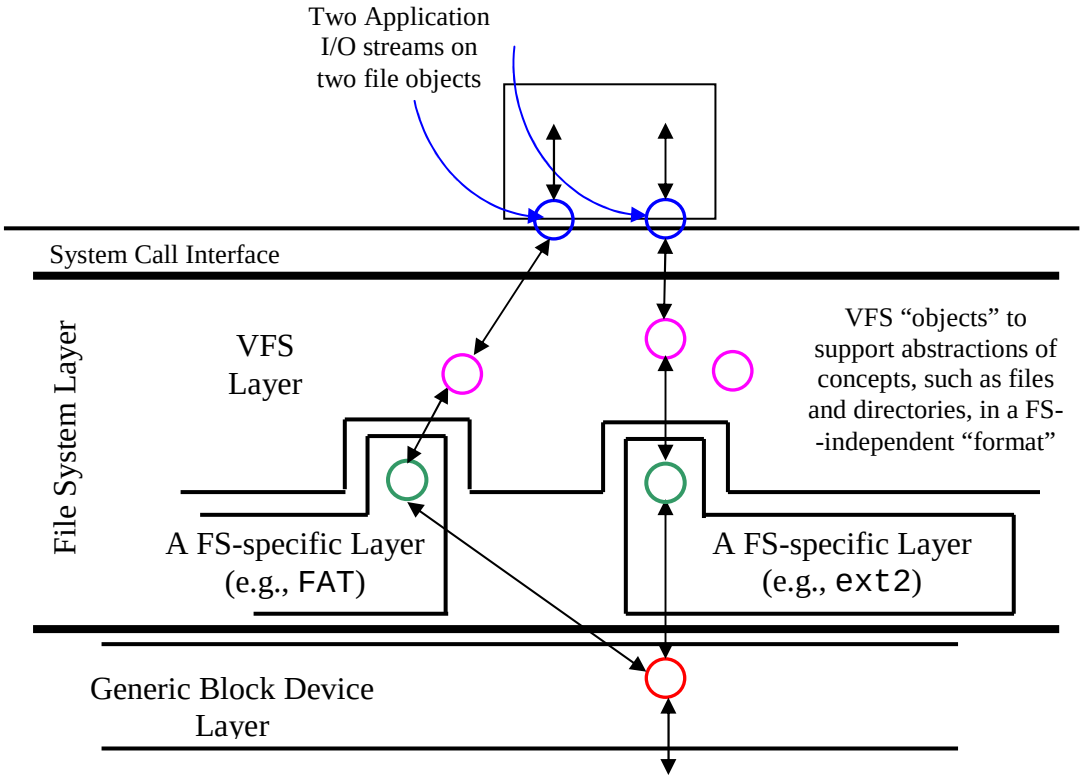


Figure 10.3 File system layers in the Linux kernel

The VFS is based on the Sun UNIX Vnodes architecture [Kle86]: a software framework that captures the commonality between different file systems and defines a kernel-level interface that enables simpler implementations of both the file-related system calls and the specific file system being supported, as depicted in Fig. 10.3.

10.4.1.1 The VFS layer

The VFS layer is implemented using an object-oriented (OO) approach, and VFS *objects* have *methods* to operate on them; methods live at the VFS layer and may be redefined at lower level layers, namely at the file system specific layer – in what resembles OO generic methods for a class that get specialized in their subclasses; for example, if a vnode object holds an inode that represents a regular file living in an ext2 file system, the `read()` method for that vnode ends up invoking the VFS `ext2_file_read()` function (see Fig. 10.3), while if it holds an inode that represents a "file" in the `/proc` pseudo file system containing information about a SCSI adapter, the `read()`method for that file ends up invoking some device driver function that accesses the hardware adapter and pulls out (reads) some information.

It is important to point out that Linux terminology can sometimes confuse the reader; for example, in Linux' terminology, inode is used to refer to two distinct concepts: the in-core VFS generic structure (which we have called vnode previously), and the on-disk inode[4]; it is up to the reader to make the distinction, using the context. There are even cases where the same name is used to refer to a third structure, an in-core image which is "slightly" different (e.g., the endian-format) from the on-disk layout.

10.4.1.2 The file system specific layer

When a disk partition is formatted to hold a specific file system type, such as ext2 or FAT file system, some data structures are created and laid *on-disk* to hold persistent data. The role of the specific file system implementation module is to provide methods to access these data structures – they must first be read from on-disk to their in-core images, and then re-arranged into generic, file system independent, VFS objects; modified objects must be written back to disk, later on, to update the persistent file system information – a step requiring a conversion from their in-core format(s) back to their on-disk layout.

Porting an existing (or developing a new) file system to Linux is thus a task that requires the developer to: a) understand which VFS objects provide generic file system abstractions such as vnodes, superblocks, etc.; b) port (or write from scratch) the code that reads and writes the on-disk data structures from/to their in-core data structures – let's call these the file

[4] Provided that the specific file system uses such a structure, as in ext2; if not, as in the FAT, the VFS vnode is "virtualised" from other on-disk structures.

system specific *private* methods; and c) implement, the file system specific *public* methods, ones that will be called by the VFS layer to perform the appropriate actions.

10.4.2 Caching

10.4.2.1 Introduction and terminology clarification

Caching is, as we've seen before, an important technique to boost file system performance, but it also brings in new problems that must be adequately solved, otherwise the whole effort will be useless. When using caches, important issues that must be appropriately tackled include the cache unit size and cache replacement policies, coherency among various caches and/or cache levels, and the possibility of loosing data upon system failures. For data (content) caching, Linux has evolved from an implementation based on two separate caches (a buffer and a page cache) into a single unified page cache. But caching may also be used to speedup accesses to metadata structures, something that will be covered further down.

It is worth noting that in the memory management (MM) terminology used in the Linux kernel, cache refers to a memory area that is used to hold (any) frequently created/destroyed objects, not only file system objects. Such a cache is further subdivided into slabs, each capable of holding a certain number of objects. That's why these caches are also referred to as *slab caches*; some, typically used for transient object allocation, may hold dissimilar objects. In this work we are interested in those (slab) caches which hold file system objects of one type only, e.g., ext2 inodes, or dentries, and are organised in such a way as to be efficiently searched (usually by some hash-based lookup function).

10.4.2.2 The concept of a buffer cache

An application requests (reads or writes) data in "records" of some specific size, while the data transfers between disk and memory are carried out in blocks; *buffering* is the technique used to handle the mismatch between the size of the data requested by the application and the amount that needs to be accessed on the device, while *caching* is a technique used to keep data, once retrieved from disk, in memory, hoping that it will be reused again in the near future. Due to the differences between access times to in-core and on-disk data, which span several orders of magnitude, caching is an important technique for increasing performance in file systems. Buffering and caching started out as two distinct, complementary approaches, but were soon merged in a unified structure, the *buffer cache*, available in the first UNIX implementations. On those days, the amount of memory to set aside for the buffer cache was a kernel configuration parameter, fixed at boot (or even worse, at kernel build) time; this was a nuisance for system administrators, who tried to tune it for a compromise between a good hit ratio for file access, and not stealing so much memory that forced the kernel to heavily page when running "memory intensive" applications.

10.4.2.3 The concept of a page cache

In early UNIX and Linux releases, caching of executable program file images was done at a page-structured cache (*i.e.*, the unit of caching was a page, containing logically contiguous data from a file), while buffering and caching of "regular" file data was handled in a separate buffer cache, as described above, one containing frequently accessed disk blocks; starting with SunOS 4 (1988), both were merged into a single unified page cache, one where the unit size was a full OS page; UNIX System VR4 implementation immediately adopted it, and Linux introduced it in version 2.4.10, about 12 years later.

The unified page cache brought in a number of benefits, such as: a) less code, as code needed to maintain the two separate caches consistent was removed and, b) code for regular I/O calls may was merged with code for memory mapped (`mmap`) file access; and c) reduced memory pressure – the page cache is dynamic and, when more and more data is cached and memory starts to become scarce, the MM and the FS may work together to shrink the page cache by discarding unmodified and/or flushing out modified pages. An added benefit, which further decreases memory pressure, is that it allows distinct file systems to "plug" themselves into the page cache, thus avoiding per-file system private caches which get "polluted" with multiple copies of the same data when a file is copied from one FS to another.

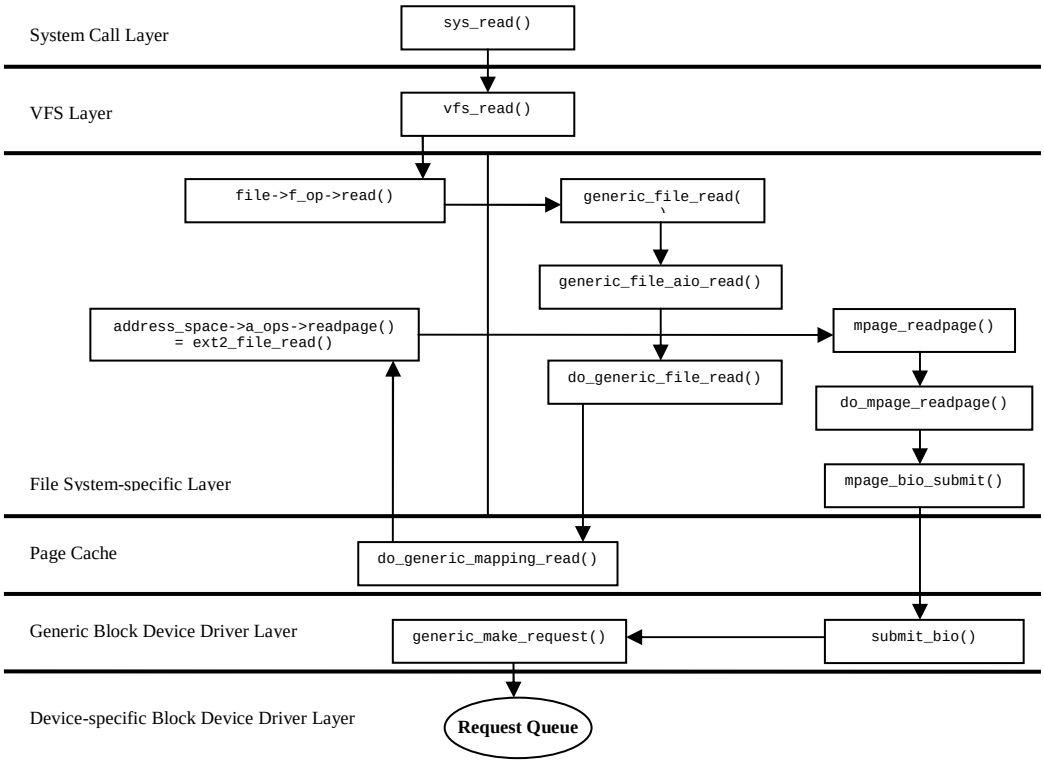


Figure 10.4 Read flow for an ext2-hosted regular file (from [Rod+05])

Fig. 10.4 illustrates the top-down flow of a `read()` call on a regular file living in a ext2 volume: a) the `vfs_read()` function executes the `read()` method for the file – the one

specified in the `read` field of the `f_op` vector of operations which, when referring to an `ext2` file, has been "loaded" with the `generic_file_read()` function; b) some further processing is done, and the `do_generic_mapping_read()` function is called to access the page cache; c) if data is found, it is copied to the user buffer, etc.; d) if data is not found in the cache, the `mpage_readpage()` method, stored in the `readpage` field of the `a_ops` vector of operations is invoked – this method is the same one which is called when accessing the file through the `mmap` interface, *i.e.*, from this point on, the two call graphs are merged.

As there is only one place to cache data, coherency between concurrent accesses from both user and kernel processes may be maintained by resorting to OS-level mutual exclusion mechanisms; the only added complexity here comes from devices performing DMA from, or into the cache while processes concurrently access it – the Linux solution is to define a flag in the page descriptor structure to signal that an I/O is in progress. The way coherency is enforced, thus, guarantees that a strict compliance with POSIX file sharing semantics (see 10.4.3, below) can be achieved by a particular file system implementation. Cached data may be out-of-sync – *i.e.*, be more up-to-date – with respect to data living on-disk, but that does not conflict with the sharing semantics, as long as all requests flow through the cache; to keep disk data synchronized a kernel daemon periodically flushes out modified pages to disk; data can be also flushed on-demand, either on the last close of the file, when explicitly requested by the process, or implicitly, when requested in the file open or at file system mount time, and/or on every write, if the `sync` option is used on the `mount`.

File systems usually provide a way for applications to bypass the cache; POSIX specifies an `O_DIRECT` option to request it. Bypassing the cache may create incoherencies if other processes are allowed to open the file "with caching"[5], so the usual way out is to disallow it, *i.e.*, if a process has a "direct open" on the file, any other process subsequently attempting a "regular" open will get an error. Direct I/O is used by highly tuned user applications, or, more commonly, by DBMS engines which perform their own caching on behalf of their clients.

10.4.2.4 Metadata caching

Metadata access must be fast, otherwise it gets in the way of data access and thus hinders performance; for example, indirect blocks must be accessed before the data blocks they point to, so indirect blocks benefit from caching. Metadata structures may be separated into two groups: file system metadata, which users are generally unaware of, and metadata for "user-visible objects"; examples of file system metadata structures are superblocks and space management bitmaps; examples of file metadata are indirect (a.k.a. index) blocks, and inodes.

[5] Because data buffered in the user space of a process using direct I/O could be out of sync with data maintained in the page cache on behalf of other (non-direct I/O) processes accessing the file.

File system metadata structures are block-based, so they do not truly belong to the page cache; however, Linux uses pages (called buffer pages) to contain metadata block-based structures instead of file data pages; for example, it holds the in-core image of some FS--specific "inode" in a buffer page, but store its VFS counterpart (vnode) in a slab cache (*i.e.*, there are functions to translate between the VFS objects and the in-core images – which will then mimic the on-disk – data structures).

10.4.2.5 Caching directory data

When performing a file open, the file's pathname must be broken into a series of filenames (separated by the *slash* token) all but the last one identifying directories. For each filename, a `dentry` object, associating (storing) the filename and its inode, is created and inserted into the dentry cache; future references to other filenames which have part of, or the same components in their pathnames will be much faster to resolve, as access to these components only requires cache lookups, and does not need disk accesses at all. Dentry objects stay in cache in a most recently used policy.

A special case is one of a negative dentry: when, in the "middle" of pathname processing, a filename component does not resolve to an existing file, the dentry is still cached, but with the inode pointer set to `NULL`. Negative dentries accelerate the resolution of failed paths.

10.4.3 File sharing semantics in Linux

To understand the POSIX [IEEE04] file sharing semantics wording, it helps to join sections taken from both its rationale and descriptions for the `read()` and `write()` calls on the issues of concurrency, stated as a) "This volume of IEEE Std 1003.1-2001 does not specify behaviour of concurrent writes to a file from multiple processes. Applications should use some form of concurrency control." and atomicity, stated as b) "I/O is intended to be atomic to ordinary files and pipes and FIFOs. Atomic means that all the bytes from a single operation that started out together end up together, without interleaving from other I/O operations." Furthermore, it is stated that c) "after a `write()` to a regular file has successfully returned, any successful `read()` from each byte position in the file that was modified by the write shall return the data specified by that `write()` for that position until such byte positions are again modified, and any subsequent successful `write()` to the same byte positions in the file shall overwrite that file data".

File sharing semantics is not an operating system, but a file system issue; thus we cannot talk about "Linux file sharing semantics" in general but instead we should refer to the specific file system being discussed, such as ext2/3 [Bov+05] or XFS [Chi+06]. For example, a `write()` to an ext2-hosted file follows a call graph similar to the one of Fig. 10.4; the `generic_file_write()` issued as a result of the `file->f_op->write()` does lock

a semaphore in the file's vnode, which results in an mutual exclusion between writes against the same file. But no call in the `read()` path observes that (or any other) semaphore, and thus the following is possible: a) in a multiprocessor architecture, a transfer from the page cache to satisfy a read is in progress while some remaining portion not yet transferred is being modified by a write from another process; or, b) in a uniprocessor architecture, a transfer from the page cache to satisfy a read is in progress and it page-faults in the user buffer – and the process sleeps, waiting for the page, while another process is scheduled and modifies the cached contents, so, when the first process resumes, the data in the user buffer is "half-old/half-new". Thus ext2 and all other file systems that use the same VFS generic routines for reading and writing will not preserve read, but only write atomicity. So, strictly speaking, ext2 does not comply with "POSIX file sharing semantics" as clause (b) above is not observed; this is contrary to established UNIX implementations which offer full file I/O call atomicity (*i.e.*, any I/O call issued against a file is atomic with respect to any other call issued against the same file).

11 Distributed File Systems

By its own nature, a DFS has many clients[1] with whom it shares one or more file system "objects"; but, contrary to what is commonly found in local file systems, some distributed file systems do not allow processes running on distinct clients to share a file for read/write, while others place sharing restrictions that some applications simply cannot tolerate. The reason behind it is caching vs. coherence: in order to get an acceptable performance out of a DFS, caching must be extensively used; but its use implies that multiple clients sharing the same file should agree on how, when, and where modifications made to a file by some process are going to be noticed by others, *i.e.*, the sharing semantics offered by the DFS will dictate the caching consistency policy, or *vice versa*.

In the following discussion we reuse several concepts introduced earlier in this document; but we have also drawn some new material (mainly examples) from [Lev+90] which, although not covering the current breed of file systems, still covers a lot of fundamental ground.

11.1 Sharing semantics for DFSs

Different distributed file systems do exist for distinct environments, covering the whole spectrum from high latency, wide area distributed architectures, down to the very low latency "in-a-box" MPP architectures (the later usually hosting some form of parallel file system). Such diversity determines which sharing policies can realistically be offered in some specific

[1] Here, client is a system that accesses data in the DFS; it does not imply a client/server architecture.

architecture/DFS combination when pursuing such goals as compatibility with existing applications and/or good performance. An overview of file sharing semantics offered by some well known distributed file systems, starting with more relaxed ones and ending with the strictest case, the POSIX single system equivalent semantics, follows. Metadata sharing is also important, and will be covered further down.

Several distributed file systems have been proposed along the years, together with various degrees of file sharing, ranging from immutable files to POSIX file sharing semantics; the reason behind the wide range of available options is a consequence of design choices: for a given environment, *e.g.*, storing user home directories (which generally are not used for sharing data among users) across a WAN, a DFS may favour performance over consistency, while for a different environment, *e.g.*, storing software development repositories shared across a user team, a DFS should favour strong consistency above everything else.

Applications developed to run on a particular file system, e.g. one which offers a specific sharing model, may not run correctly or with adequate performance when moved to a different one, which does not offer the same model; this happens quite often when applications are, e.g., moved from a local file system to a DFS – there may be a mismatch between the application expectations and what the DFS provides.

11.1.1 Immutable semantics

The simplest sharing semantic is the one of immutable files: every time a file is designated as shareable, its contents cannot be modified ever again. Although this idea has recently been pushed in a slightly different way, in Content Addressable Storage (CAS) appliances such as EMC Centera [EMCa06], it is not relevant to our work, so we will not continue on this path.

11.1.2 Versioned semantics

Versioned is also a simple sharing semantics: every time a modification is done to a file, a new version is created, but the previous one is also kept; clients that already had the file opened for reading, will continue seeing the "old" version. This idea has been used in versioned file systems such as CVFS [Sou+03]; as with the immutable semantics above, versioned semantics is not relevant to our work.

11.1.3 Transaction semantics

Transaction semantics follows on the same ideas as transactional data base systems and applies those ideas to file sharing. The main concepts of transactional data base systems are those of transactions, delimited by a begin/end pair, atomicity of changes, consistency of data that ends up in the data store, isolation between processes' views of data, and durability of stored data – the often touted ACID [Hae+83] properties. An example of a very simple use of transaction semantics is session semantics, described below, which uses open/close as

begin/end pairs; other more elaborate implementations include additional file system calls, thus deviating from the POSIX API for file access.

Transaction semantics is often used to guarantee metadata consistency in the presence of concurrent operations executing across several nodes in a distributed file system; it is not, however, commonly applied to data sharing.

11.1.4 Session semantics

Session, also known as close-to-open semantics, is the simplest form of transaction semantics: when a file is modified, clients currently accessing it do not immediately get the results of the modification; the updated version of the data will only be noticed after the updating process issues a close and then, either a new process opens the file, or processes that already had it opened, close the file and then re-open it again. Some DFSs do offer different semantics depending on whether interactions occur on the same node on or distinct nodes; for example, they may offer session semantics if processes sharing the file do not run in the same node, but otherwise, they offer regular POSIX semantics. Those DFSs obviously violate the transparency property (considered of utmost importance in a "good" distributed system) as sharing behaviour will depend on the process' location.

As with immutable files, session semantics works well with full file caching: when a client opens a file ("begin session"), it gets a copy of the file from the DFS' storage space[2]; the copy is then placed on a local cache where all accesses, reads and writes, will be handled; when closing the file ("end session"), a check is performed to see if the file was modified and, if it was, it is pushed back to DFS storage, overwriting the one that sits there.

The original version of the Andrew File System, AFS [How+88], is an example of a DFS with true (as defined above) session semantics; later AFS versions allowed for partial caching of a file in 64 KB segments, as a way to decrease cache pressure. NFS also implements "a sort of" session semantics, along with other types; we will study NFS in detail, later.

11.1.5 POSIX single-node equivalent sharing semantics

In a DFS, compliance with POSIX file sharing semantics is called "POSIX single-node equivalent sharing semantics" [Sch+02], and calls for a file sharing behaviour which is exactly the same as in a POSIX-compliant local file system (see 10.4.3); consequently, if a DFS supports it, then full transparency is preserved, *i.e.*, there is no "impedance mismatch": applications will see the same behaviour, with regard to sharing, when executed either in the local, or in the distributed file system. Or, to put it differently, in a DFS (such as GFS) which offers POSIX single-node equivalent sharing semantics, concurrent execution of file operations (reads and writes) is performed in a sequentially consistent way.

[2] Being irrelevant if the storage space is distributed across several nodes or held at a single one.

11.1.6 Other file sharing semantics

We have presented several types of sharing semantics which are both conceptually clear and important landmarks. Due to the quest for ever increasing performance, almost every distributed file system proposes its own semantics, one which may be close to but not always quite the same as those introduced before; they are better understood when we study them along with their respective file systems, something that we will do in a moment for a few, selected case studies.

11.1.7 Performance, cache coherency and file sharing semantics

As clearly stated in [Kaz+88], "many distributed file systems go to great extremes to provide exactly the same consistency semantics in a distributed environment as they provide in the single machine case, often at great cost to performance. Other distributed file systems go to the other extreme, and provide good performance, but with extremely weak consistency guarantees. However, a good compromise can be achieved between these two views of distributed file system design".

So, "strong" semantics, such as POSIX sharing semantics, leads to poor performance in a DFS; or vice-versa; but we can add another ingredient, one who may boost performance: caching. The problem with caching in a DFS is that strong semantics also requires strong cache consistency, and we're back to poor performance… or not? Breaking this circle is now possible by adopting new technologies, such as high speed, low latency, RDMA-capable interconnects, and combining them with smaller grained caches, low overhead invalidation protocols, and other novel solutions, such as throwing in another, often ignored issue: locking.

11.1.8 DFS as a part of a distributed operating system

Distributed file systems offering POSIX single-node sharing semantics are quite scarce when compared to the large number of DFSs implementing other semantics; furthermore, it is interesting to note that all examples (namely Locus and Sprite) quoted in [Lev+90] of DFSs that implemented UNIX semantics, were "embedded" within distributed operating systems.

These are interesting examples because, besides offering the atomic read/write behaviour, they used shared file pointers. In UNIX, for example, when a process forks a child, a shared file pointer is "created"; if the child keeps the inherited file open, then an operation that changes the file offset in a process, e.g., the father, results in the an observable (offset) change in the other process, in this example, the son. This is the current semantics for `fork()`in Linux and other POSIX compliant operating systems.

In a SSI operating system, where it is possible (performance issues apart) that a process (or thread) creation may result in the new one being created in a different node (as in the Kerrighed OS [Mor+04, Lot01]). For such an environment, a POSIX compliant file system able to support shared file pointers would be quite sought after, whereas for "typical", HPC

clustered or networked environments where each node runs its own private copy of the OS, "native" support of shared file pointers at the DFS level is not overly important as, if needed, it can be provided by runtime libraries, such as MPI.

11.1.9 High performance distributed file systems

High performance distributed file systems, usually referred to as parallel file systems, are specialised DFSs used in MPP and HPC cluster architectures, where they offer file models that enable carefully programmed and/or tuned applications to extract high performance from the underlying I/O subsystem (where "model" stands for organization, access, and sharing semantics).

11.2 The file abstraction and the distributed cache

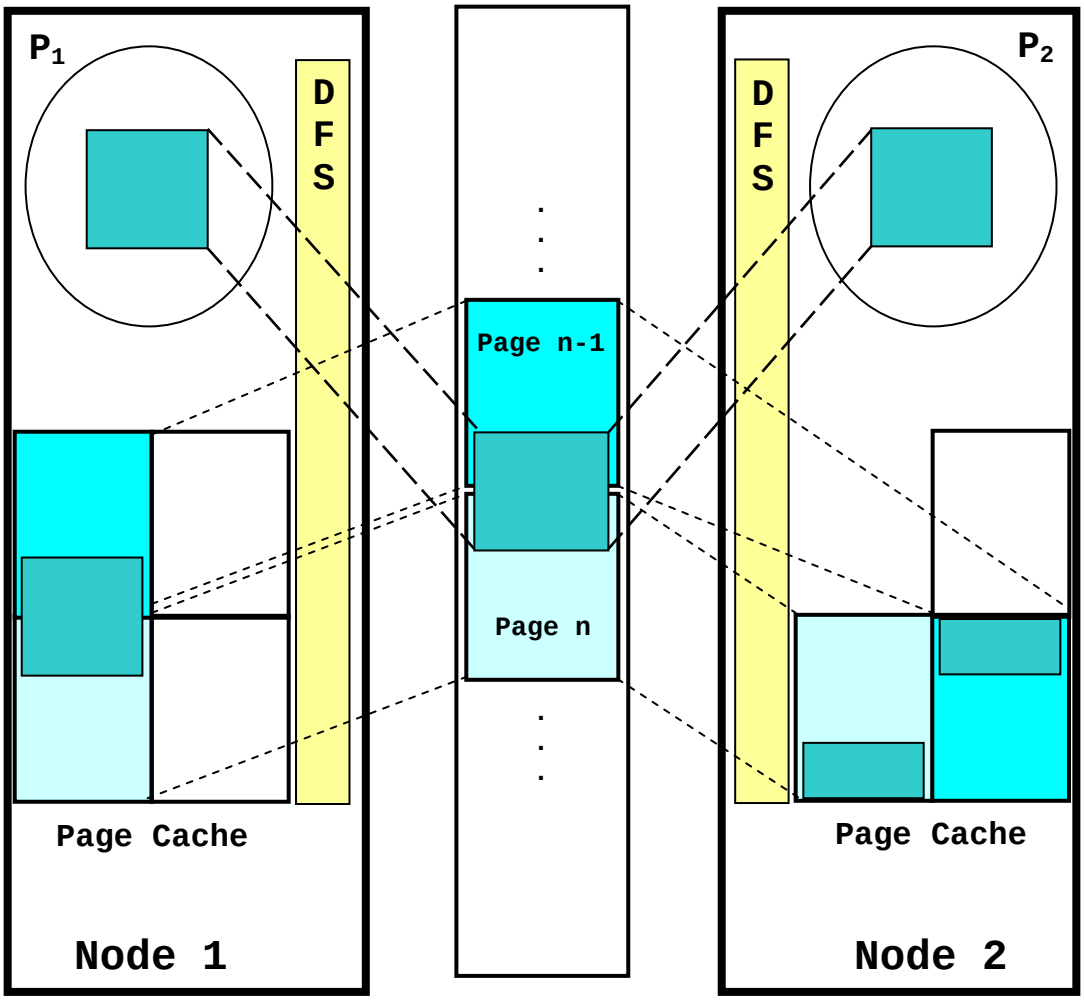


Figure 11.1: Global File View created from page caches of all nodes.

We will now take a look at an hypothetical distributed file system running in a multi-node, distributed memory architecture, where each node runs its own separate OS copy with its own, separate, page cache; each node also has (for simplicity) a set of local disks, and files are

striped across the nodes. Figure 11.1 depicts an example of such architecture: it has two nodes, each one running its own operating system copy, and the DFS is tightly integrated into the OS and its page cache. For this architecture, we want to apply the same reasoning as before: to satisfy a file access request submitted in a node, we want to build a Global File View (GFV) by resorting to a union of the page caches of all nodes.

Definition 11.1: global file view

A GFV is the view of the file that we get when we perform the union of the pages of a file *f*, stored in the page caches of all *i* nodes. We can thus express it as

$$GFV(f) \equiv \bigcup_{i=1}^{n} < Page-Cache(f) >_i$$

The above definition raises some issues; for example, some particular page of the file f, say $P_k(f)$, may be present in more than one page cache at the moment, t, when we perform the "select" operation. Clearly there should be no problem for our union operation if the contents of that page are the same in all the caches where it can be found; but what to do if it is not? Another issue we must look at is time: how do we specify *time* t across several systems? And how do we perform the union operation at *time* t?

So, buried in the apparent simplicity of the above formula, there are very important (and complex) issues such as cache coherence (or consistency), distributed time, and distributed operations. In the next subsections, we will introduce case studies for a few representative (i.e., broadly used) DFSs, in order to get a better understanding of the problems we face when caching is used in a DFS.

11.3 Case study: caching in NFS

NFS is the most utilised DFS. This alone mandates its inclusion as a case study; but NFS is also a representative of (distributed) client/server file systems, supports file locking, and client-side[3] caching – all strong reasons for its inclusion in this set of case studies. The following discussion applies, broadly, to NFS versions 2, 3 and 4; we will focus our attention mainly on v3; however, we may sometimes refer to v2 or v4, to illustrate some point.

11.3.1 Cache consistency policies

The protocols introduced for NFS v2 and v3 do not define policies for client or server caching; in particular, there is no support for strict cache consistency between a client and server, nor between different clients. Existing client and server implementations do usually offer distinct caching policies (detailed below) allowing the administrator to choose the appropriate one for each case.

[3] NFS' server-side caching is one of a local file system, and irrelevant for this discussion.

11.3.1.1 Time-based cache consistency

Time-based NFS client cache consistency is a best-effort policy: when a client gets its first data for a particular file from the server, it also fetches the file's *time of last modification* and stores it, along with a reference to the moment, t_c, when that piece of data was cached; for an access occurring at a later time, t, the data in the cache can be used to satisfy it if $t \leq t_c + t_{TTL}$, i.e., if the access was performed within the time bounds allowed for a cached copy to live in the cache. If that limit is exceeded, the client will contact the server to fetch the file's time of last modification, once again; if there was no change, t_c is updated (renewed) and the data in the cache is still valid, and may be used to satisfy the request; otherwise, the cache has to be purged and data will have to be fetched again from the server.

In the Solaris NFS implementation, for example, the t_{TTL} value can be chosen between 3 and 30 seconds for regular data files (or 30 to 60 seconds for directories), with smaller values guaranteeing better client cache consistency at the expense of increased traffic between the clients and the server [Cal00].

11.3.1.2 Open-to-close cache consistency

Open-to-close cache consistency was spurred by the observation that, when UNIX was used at university campuses file "sharing" was, most of the time, completely "sequential": first, client *A* would open a file, write something to it, and then close it; then, client *B* would come in, open the same file, and read the changes.

Open-to-close cache consistency [Cal00] is implemented in NFS in such a way that, when an application tries to open a file stored in an NFS file system, the NFS client first checks, by sending the server a `GETATTR` or `ACCESS` message, that the file exists and has suitable permissions. When the application closes the file, the NFS client writes back any pending changes to the file so that the next opener can view the changes. This also gives the NFS client an opportunity to report any server write errors to the application via the return code from `close()`. After closing the file, cached data needs not to be discarded, as it can be useful for another open; as an example, Linux implements this close-to-open cache consistency by caching the results of a `GETATTR` operation issued just after the file is closed, and comparing them to the results of the new `GETATTR` issued when the file is re-opened. If the results are the same, the client's cache is still valid; otherwise, it is purged.

Open-to-close cache consistency has certain similarities with session semantics: if all data in client *A* is flushed on the `close()`, and not before, and client *B* opens the file afterwards, the result is the same. A different situation arises if *B* had already opened and read some data from the file by the time *A* flushed it, and *B* continues reading – then it will get modified data; the same will happen if A writes periodically to the file, before the close: those are situations inconsistent with the definition of session semantics. This is not the case with AFS [How+88],

for example, as it offers true session semantics – the client will fetch the whole file to its local disk cache on the first open.

11.3.1.3 Weak cache consistency

Open-to-close cache consistency can create an enormous memory pressure at the client, as it will have to postpone all modifications until the `close()`. The NFSv3 protocol provides procedures and data that clients can use to implement another policy, one designated weak cache consistency (WCC) [Cal00, RFC1813]; procedures, and data submitted/returned by those calls, provide a way for a client to check a file's attributes[4] before and after a file modifying operation, such as a write of file data or setting of its attributes; as a consequence, a client can easily identify changes that could have been made to the file by others, and thus purge its cache.

11.3.2 NFS cached objects

NFS clients usually cache more than just file data; other cached objects include directory entries, lookup replies, and other metadata information such as file and directory attributes, file system information, etc.

When NFS clients perform `LOOKUP` operations they get replies which include file handles and file attributes, and they cache those replies; for example, a Linux NFS client caches them at the VFS' `dentry` and `inode` caches (see 10.4.2.5). When a client detects a change in the parent directory's time of last modification, it purges all cached entries for that directory; when the client itself modifies a directory, an NFSv2 client also purges all cached entries for that directory (to minimize the risk of changes performed by other client on the same directory getting unnoticed), while an NFSv3 client may use the WCC enhancements to avoid unnecessary purges (of course, it's the reloads that are expensive, not the purges themselves).

The results of `READDIR` and `READDIRPLUS` operations may be also cached; caching `READDIR` results is useful to avoid failed `LOOKUP`s to the server, because having all directory entries cached allows the client to reject references to filenames which do not exist without even querying the server; on the other hand, caching `READDIRPLUS` results allows us to skip both "negative" `LOOKUP`s, as above, but also "positive" ones (e.g., with insufficient permissions) as the `READDIRPLUS` call already returns "extended attributes" for the entries.

11.3.3 File sharing semantics in NFS

As hinted before, file sharing semantics in NFS (versions 2, 3 and 4) is of the "close to, but not quite" variety: a) its close-to-open cache consistency is roughly equivalent to session semantics, provided that clients open a shared file in turns, *i.e.*, reader(s) open the file after the writer has finished using (closing) it; b) its time-bounded cache consistency is roughly

[4] File attributes are NFS' parlance for file metadata.

equivalent to POSIX single-node equivalent semantics (strong cache consistency) provided that sharers wait enough time between accesses to allow the caches to expire – the limiting case being no caching, suffering the performance degradation it brings along; c) WCC just provides a faster procedure for a client that performs file modifications (*e.g.*, writes of data or attributes) that travel through the server, to detect modifications previously performed by other clients and thus invalidate its cache. Another important issue is that NFS is not *transparent*, in the sense of that desired distributed system property, with regard to file sharing: behaviour observed by clients will be different depending on whether a set of processes that share a file all run in the same, or in distinct NFS client hosts.

NFS has been designed to perform well in distributed environments where file sharing is an *infrequent* event; for situations where this is not true, the only way we can guarantee consistency in NFS (versions 2, 3 and 4) is through the use of record (also called byte-level) locking and turning client caching off (use of file locking in NFS requires some knowledge of its interactions with caching, otherwise the expected behaviour may not materialise; for a more in-depth coverage, see [Cal00]). This is, of course, very detrimental to performance.

11.4 Case study: caching in PVFS

PVFS, the Parallel Virtual File System [Car+00], is quickly becoming one of the most utilised high performance distributed file systems, at least in the open-source domain; it is also a representative of client/server file systems but, contrary to NFS, one which does not support client-side caching or file locking at all[5]. That being said, PVFS is well worth being studied as a DFS strictly designed with HPC in mind, an environment where file sharing is not uncommon but where processes sharing a file do not, as a rule, engage in "conflicting", *i.e.*, overlapping requests.

11.4.1 Cache consistency policy

PVFS does not use client-side caching, as it would compromise its ability to guarantee correct operation in a read/write (or write/write) sharing across client nodes; as one of the developers puts it, "Many network file systems like NFS have weaker consistency guarantees on file system data and meta-data, since they are primarily targeted at workloads where it is not common to have many processes accessing the same files or directories from many nodes simultaneously. PVFS, on the other hand, cannot afford to have such weaker file system semantics guarantees because it is primarily targeted at workloads that exhibit read-write data sharing. Therefore, PVFS (at this stage) does not cache file data and meta-data in the Linux page cache; in other words, all file system accesses have to incur a network transaction" [Vil+04]. Of course, one could implement client-side caching by resorting to a cache

[5] Recently, a locking API was proposed for the PVFS' MPI interface [Chi+07].

consistency protocol, perhaps even supplemented by mechanisms such as locking; but this is not the PVFS way, as stated by the original designers: “PVFS has no locking component. Instead, the metadata server supplies atomic metadata operations, eliminating the need for locking when performing metadata operations. This approach allows for a relatively simple system with no file system state held at clients, but it precludes client-side caching, which makes for very poor performance in a number of cases, particularly single process workloads” [Lig+03].

11.4.2 File sharing semantics

In PVFS, data operations are guaranteed by I/O servers to be consistent for concurrent writes that do not overlap at a byte-level granularity, and results are immediately visible to other clients; but byte-level overlapping concurrent writes result in an undefined file state, while concurrent reader(s) that overlap their accesses with a writer may experience a mixture of old and new data [Lig+04, Vil+04], thus violating the “POSIX single-node equivalent semantics” (*i.e.*, sequential consistency).

What is specific to PVFS (and all DFSs) is its distributed nature; thus, there are two issues here: one, being how to interpret the “after” in the POSIX fragment “after a `write()` to a regular file has successfully returned…” (see 10.4.3); the other, being the cost of implementing I/O call serialisation. The first issue is not that different from what happens in a multiprocessor: when two events occur in separate flows (processes or threads), asserting that *event B* (starting the read) occurs after *event A* (returning from the write) is only possible if both synchronise themselves either by exchanging messages, or through the execution of some synchronising call. As for the cost of serialising operations, it is at least one order of magnitude higher in a distributed than in a centralised system (such as in the above mentioned multiprocessor), where it could be implemented by directly accessing the system’s memory (at less than a hundred ns), instead of with resorting to messages exchanged among nodes (even if they are carried over a very fast communication infrastructure, such as Infiniband, they take a few μs per message); this is the main reason behind PVFS’ decision to drop both serialisation (among I/O operations) and file locking.

To conclude, PVFS does not offer the sequential consistency property of “POSIX file sharing semantics” and, furthermore, lacks file locking in its “POSIX” API.

11.5 Case study: caching in GFS

GFS, the Global File System [Sol97], is a fully symmetric distributed file system based on shared disk storage, where all nodes have equal access to block storage devices; the usual configuration is based on a FC SAN interconnecting hosts and disk arrays, but an Ethernet based SAN where hosts and arrays communicate through an iSCSI protocol is also possible.

11.5.1 Cache consistency policy

GFS implements strict coherence among its client's caches, thus paving the way to be able to offer POSIX single-node equivalent semantics (as defined in 10.4.3). Cache coherency is implemented in GFS by resorting to locking and invalidation; we now briefly present, in an overly simplified way, the main concepts used to implement it; for a more detailed study see section 19, "GFS internals: an introduction".

In GFS, some in-core "objects" have local, intra-node visibility while others may be shared among client nodes, *i.e.*, have a broader, global, inter-node (cluster wide), visibility; examples of objects having a cluster-wide visibility are ginodes (GFS structures within the VFS vnodes). When a ginode (or any other cluster-wide visible object) is "created" for the first time in the cluster, a cluster-wide global lock – G-Lock – is also created to protect that object; the G-Lock is uniquely identified by a value pair which holds the object's type (*e.g.*, "regular file" inode) and number (*e.g.*, inode number, based on its on-disk location).

When a process wants to perform an operation on a G-Lock protected object, the following protocol must be observed: first, the *process* must *acquire* the G-Lock in a suitable *locked* state; then, the process performs the desired operation(s); next, the process *unlocks* it, and finally, the *node* may *release* the G-Lock. Acquire/lock and unlock/release are cluster-wide operations that may involve a global lock manager (distributed or not) and, as such, incur in non-negligible communication latencies and processing overheads; several nodes may hold a G-Lock in the *shared* state, but only one is allowed to hold it in the *exclusive* state. Lock/unlock are used to implement mutual exclusion for intra-node operations.

As an example, consider a file being opened for the first time in the cluster, for reading: as part of the `open()` call processing, the node where the process requesting the file open is being executed asks to the Lock Manager to create a G-Lock for the new ginode (and vnode) object; then, it acquires the G-Lock in the shared state and locks it "local-exclusive" (preventing another process in that same node from simultaneously trying to open it), and fills the inode with data from the (on-disk) inode image; next, it *demotes* the lock state to "local-shared". Now, each time the user process performs a `read()`, as the process already holds a lock on the G-Lock, standard VFS-level mutual exclusion operations may be performed while the page cache is searched, or data is retrieved from disk and placed in the node's page cache and, from there, moved to the user buffer[6]. Now if another process on the same node happens to open the same file, G-Lock creation is skipped, as the node already "has" the G-Lock; all the process will have to do is acquire the G-Lock in the shared state and lock it "local-shared" – which it will be able to do, because the acquiring and locking intents of the new process are compatible with the lock's current state.

[6] We're assuming a typical file usage pattern, where the page cache is used.

If a process in another node opens the same file for writing, the same set of operations is carried out: an inode and a G-Lock (identified exactly by the same pair) will be created, and the G-Lock acquired in the shared state, and locked in the "local-shared" state. Now, each time the process performs a `write()`, the following sequence will be carried out: the process will attempt to lock the G-Lock exclusively, a request which can't be immediately granted because another node already holds the G-Lock in the shared state; the "offending" node is called back by the Lock Manager and asked to drop the lock, which it will do *after invalidating the inode and all cached pages*; now the lock may be granted to the writing node.

11.5.2 File sharing semantics

We've just described is GFS' implementation of an invalidation-based cache coherency policy along with the serialisation of "conflicting" I/O operations, *i.e.*, concurrent read/write or write/write calls; together, they enable GFS to easily offer POSIX single-node equivalent file sharing semantics.

Unfortunately GFS' POSIX-compliance is provided at a cost that is too high, as shown both in [Lop+05] and in the performance benchmarks section in this document; two factors contribute to the observed performance degradation: on one hand, the unit for coherency is, in fact, the whole file, as any attempt to write on any file segment immediately results in data cached on other nodes to be discarded, even when the region being written is not cached; on the other hand, reading a file region on a reader node forces the writer node to immediately flush all data to disk, even when that data does not include the region being read.

GFS, as currently implemented, is thus more appropriate for situations where write sharing of a file among processes running in different nodes is an infrequent event; it may be successfully used to replace NFS or CIFS in environments where users have their "home" directories, usually private, several read-only shared folders, usually holding executable ("binaries") and/or configuration files, and a few shared directories where files are modified, but usually not concurrently. It is not, however, suited to HPC environments where several processes in different nodes concurrently share a file for writing and/or read/writing, even if accessed regions do not overlap each other.

Part IV:

A Reference Model for Data Management Architectures

This Part starts by discussing I/O flow in modern architectures and operating systems, and from there, we extract precise definitions for Parallel I/O and Parallel Disk Access. Then we propose a Reference Model for Data Management Architectures (RM-DMA) and a taxonomy for the model's upper layer ("File System Layer"). A short evaluation of the model and accompanying taxonomy is carried out as a survey of some relevant, widely known, "parallel", "distributed", "client/server" and "cluster" file systems, I/O software stacks, and storage architectures.

12 I/O in modern Operating Systems

12.1 I/O flow in modern operating systems

In modern operating systems the flow of control in file system related I/O calls is a bit more complex than what it used to be just a few years ago; today, operating systems support the concept of Logical Volume (LV) [Van+00, Lew05], and they also have I/O drivers that support multiple I/O paths (MPIO) to the same storage[1] device. Each concept contributes with another degree of freedom: LVs allow better storage space management while MPIO allows higher availability (by switching to another path on failure), load balancing (using different paths to access different storage devices) and "parallel I/O" (using several paths simultaneously). We will use Fig. 12.1 below to present, in a very simple way, these concepts and layers:

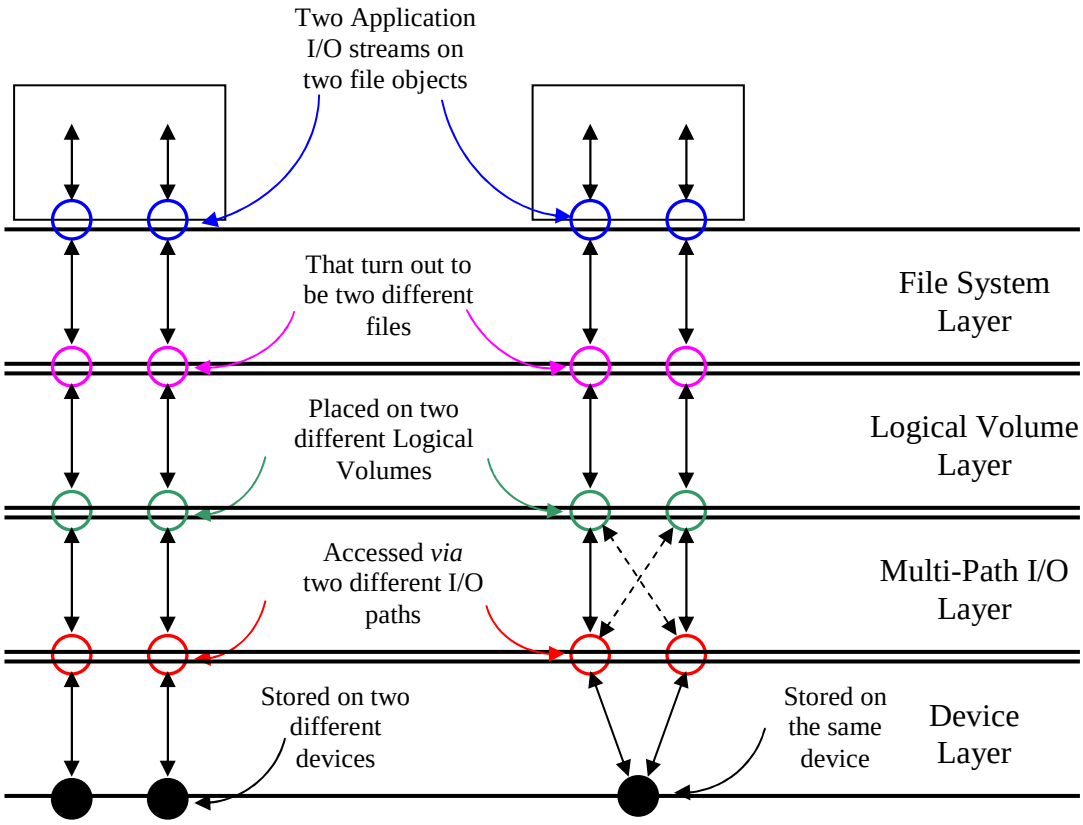


Figure 12.1 I/O data flow in modern Operating Systems

The rightmost part of Fig. 12.1 enables us to assert a simple fact: every layer, apart from the FS layer itself (at least in a typical, single node FS), can introduce a multiplex/de-multiplex function on the I/O path; here, the Device Layer multiplexes two distinct paths of the MPIO Layer into a single one. We've drawn a full mesh at the MPIO Layer (every green circle

[1] Multipathing has also been used in network drivers, where it is known as channel bonding, multi-rail, trunking, ether-channel (Cisco proprietary protocol), or link aggregation (IEEE 802.3ad).

connected to a red one) with dashed-arrows, to show that each volume can be accessed from both paths; that enables us to have higher availability (i.e., recover from path failure) and, if the driver supports it, load balancing (i.e., use both paths simultaneously to perform data movement – not a very interesting situation here, because there is only a single disk).

> Definition 12.1 **Parallel I/O**: We will say that parallel I/O is being performed in a system whenever multiple, concurrent data flows, exist in a layer (any layer) in the I/O path.

This is a broad definition, encompassing a lot of situations that are not commonly regarded in the literature as parallel I/O [Sto98]; according to definition 12.1, we do not care which layer is involved; but we are particularly interested in cases where multiple access paths to different disk devices do exist, so we'll formulate a definition that covers it. Therefore, we turn our attention to Fig. 12.2 which shows two situations where an application is accessing a single file whose data happens to be spread across several devices

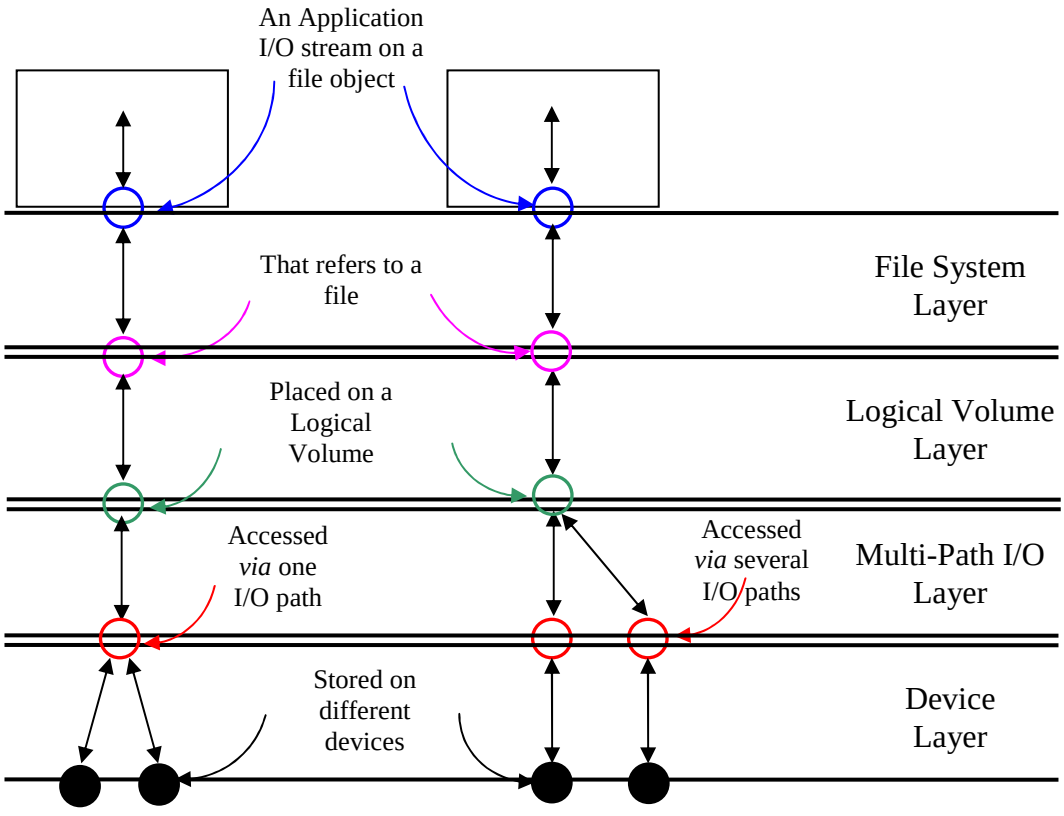


Figure 12.2 Parallel disk I/O

There are two ways to achieve the kind of parallelism depicted in the leftmost side of Fig. 12.2: we could rely on a *software solution* provided by the LV layer[2], which creates a RAID

[2] Other possibilities include layers distributed over multiple servers, as we will see later on.

logical volume out of aggregating several disks (or partitions in different disks); or, conversely, we could rely on a *hardware solution* provided by a storage array. Either way (provided we use an appropriate RAID level), we end up with file blocks being stored on different devices, a situation known as *data de-clustering* [Sto98] or *striping*, which enables *disk level parallelism* [Sto98], or, as we prefer to say, *parallel disk access*.

> Definition 12.2 **Parallel disk access**: We say that parallel disk access is being performed in a system whenever, to satisfy a single request for the transfer of a number of contiguous disk blocks, several disks are concurrently accessed.

From the definition above, we can see that the topmost FS layer interface – and thus the programmer – is not aware of the parallel disk access. Also, there is not much of a difference when we consider the rightmost part of Fig. 12.2: we just added another I/O controller, and disk devices were attached to different controllers – we're now able both to issue requests and transfer data in a truly parallel fashion, avoiding the contention that may occur on a shared interconnect. Having looked at some techniques that can be used at various levels – server architecture, I/O controllers, devices, and operating system layers (below the FS layer) – to increase I/O bandwidth in a single-server system when one or more processes are accessing one or more files in unrelated computations, we must now look at the case where a "parallel computation" is accessing some files, to see if this brings something anew to our findings.

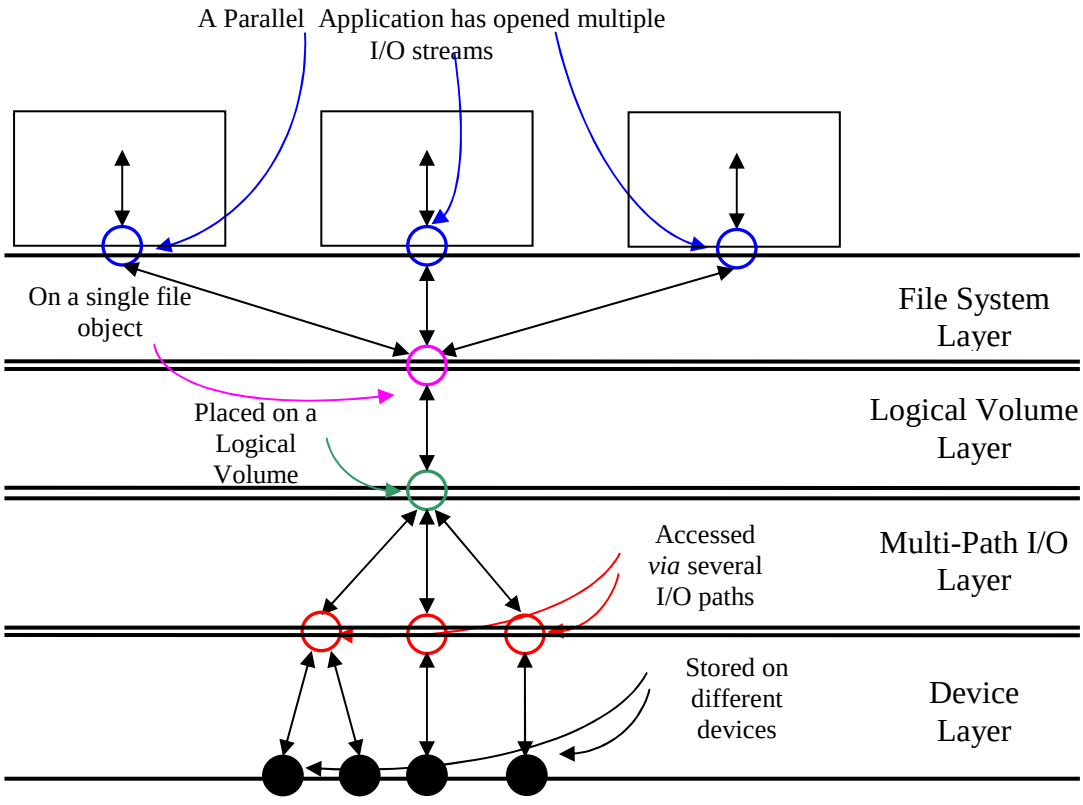


Figure 12.3 Is this parallel file I/O?

In Fig. 12.3 processes in a parallel application access single file, concurrently issuing I/O requests; the file is stored on a single logical volume, made up of four different disks; the system has three I/O adapters, and the disks are attached to different adapters, making multiple I/O paths available. Thus, according to definitions 12.1 and 12.2, the system is performing both parallel I/O and parallel disk access. Notice that if we wanted to draw a figure to represent three unrelated computations accessing a file, instead of a "parallel application", Fig. 12.3 could be reused; general purpose POSIX-like operating systems and file systems we use regularly have no API options to enable us to "tag" processes as "parallel", or convey to the FS that processes are sharing a file; so, the answer to the question posed in the legend of Fig. 12.3, "Is this parallel file I/O?" has to be, for the moment, "We don't know" – because we've not yet seen what the "parallel" adjective means, at the file system layer.

13 A Reference Model for Data Management Architectures

13.1 Introduction and motivation

Answering questions such as "Is this parallel file I/O?" and/or comparing features of distinct file systems in meaningful ways requires us to have a solid framework, one which will cover all aspects involved in I/O, and does not need to be changed to accommodate a new file system, storage device, or interconnect. Having looked around, we have not found a framework that is, at the same time, simple (i.e., easy to understand) but generic enough – in the sense that file systems, storage architectures, and configurations we wanted to study could be assessed – so, not unexpectedly, we have developed a new reference model.

13.2 Data management: the broad picture

Data management is another step in the quest for closing the gap between data abstractions that model "real world entities" and the set of tools at the disposal of application developers. Traditionally, data management has been split into two major camps, data base and file systems; FSs have been offering very simple file models (sequential, indexed, etc.) for years, but recently more sophisticated file storage and access methods in the form of semantic file systems [Gif+91] and content addressable storage [EMCa06, Tol+03] were proposed.

Today we find DBMSs in applications that have to deal with large amounts of data, organized as complex interrelated data structures, concurrently accessed by a large number of users, where accesses must be isolated in a way that data is kept coherent, and recovery from crashes should be "automatic". Conversely, file systems are used to support file-based applications that have requisites and access patterns quite different from those expressed above; they hold application files – for example, for office/productivity, multimedia, and scientific applications (which may access very large storage repositories) – but also OS

storage (for the OS itself plus all the software utilities); business applications usually do not use file systems as a major data repository technology anymore, they use DBMSs instead.

13.3 A reference model for data management architectures

We now present a Reference Model for Data Management Architectures (RM-DMA) that generalises the architecture depicted in the previous section; it is composed of two major pieces: the Storage Management Domain (SMD) and the Data Management Domain (DMD).

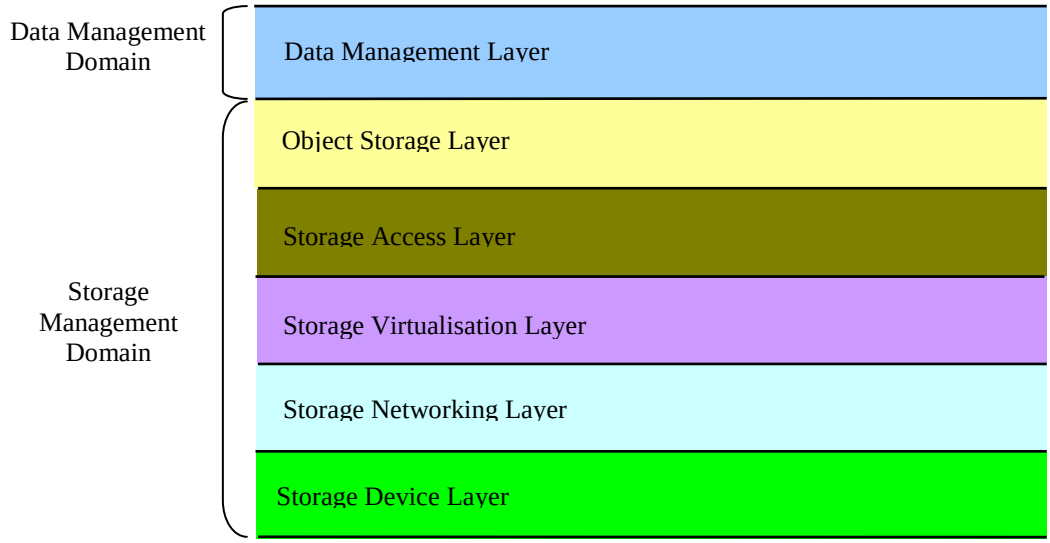


Figure 13.1 Reference Model for Data Management Architectures

The Storage Management Domain provides services that are used by DMD software to store data; it is structured into five layers: the Storage Device Layer (SDL) models the devices themselves; the Storage Network Layer (SNL) deals with protocols (packets, frames, coding and physical cabling) that are used to access the (e.g., block addressable) physical devices that ultimately store data; the Storage Virtualisation Layer (SVL) handles device virtualisation tasks, such as device partitioning and/or aggregation, and increased availability (e.g., RAID); the Storage Access Layer (SAL) provides the shared and distributed models of storage; finally, the Object Storage Layer (OSL) provides higher level storage abstractions, ones that go beyond the usual “array of blocks”.

The DMD may be split into several “vertical” layers, such as the File System Layer (FSL) that models file system software implementing the usual abstractions of files and directories as well as links, records, etc., and the Data Base Layer (DBL), for data base management software. Outside the scope of this work are DBMSs and some specialised FSs such as semantic, content addressable, and peer-to-peer FSs; therefore, the RM-DMA has not (yet) been validated against them.

Our RM is similar to the Storage Network Industry Association (SNIA) Shared Storage Model [SNIA03]; the most striking differences are that their model includes the file/data base

system software as part of the storage domain, does not cater for the object and access layers, and storage is (always) assumed to be shared.

13.4 RM for file systems: a layer by layer description

Decoupling a real-world file system into layers, i.e., matching it to our reference model may be far from trivial; this is especially true for single node (i.e., not distributed) monolithic file systems such as UFS and its descendants (e.g., ext2). The problem is further exacerbated because the FSL must interact with the operating system and therefore FS "objects" become managed in both worlds (e.g., they are allocated per FSL request, but may be flushed and de--allocated by the OS memory management layer – in close cooperation with the FSL).

And it gets even worse: in an attempt to reduce memory pressure and increase FS performance, UNIX descendants (e.g., Linux) have implemented a Virtual File System (VSL) layer [Kle86, Bov+05] and a Page Cache [Bov+05, Rod+05]; the net result is an FS-abstract layer (the VFS) whose generic structures (e.g., *vnodes* and *dentries*[1]) may not match with the FS "native structures" (ranging from quite similar for some FSs, such as in ext2, through similar, such as in GFS, up to completely different – to the point where they simply do not exist, such as in the FAT), and this complicates the "slicing" of the real FS into the layers of our RM.

13.4.1 File system layer

The FSL provides all those well known "objects" such as volumes (FS instances), directories and files, together with the API that allows them to be accessed and managed. However, there's more to it: at the "instance" (volume) level the FSL has to address security, reliability, fault tolerance and recovery, performance and scalability, and sharing semantics; at the "file" level, the FSL should define which file types, organisations, and access modes are supported, and what is the proposed semantics for file sharing. And, of course, a major aspect: is it distributed or local?

13.4.2 Object storage layer

The OSL provides the set of ADTs that will be used by the file system layer to offer the user-level data and metadata objects – files, directories, links, etc.; for example, in an ext2 file system those structures are the superblock, resource group, inode, and data and index blocks, while in a FAT file system they are the boot sector, cluster, root directory and the FAT itself. The OSL is responsible to perform transformations needed to map the ADTs it provides into the next (downwards) layer structures, e.g., implementing on-disk images in a local, block--based object storage, or accessing a peer OSL in another node in order to map them.

[1] For a more detailed description of the VFS layer see section 17, "VFS internals".

In commonly used file systems it may be difficult to identify the OSL, as it may be tightly integrated with the FSL; examples include single node local file systems such as ext2 or NTFS. Quite the opposite may occur with recent DFSs, where layers are more decoupled and, consequently, easier to identify; for example, in PVFS or Lustre [Bra03], the FSL (the "client" file system) accesses object storage servers (storage targets, in Lustre parlance) running in other nodes, i.e., making it a distributed OSL architecture.

13.4.3 Storage access layer

The SAL provides two distinct abstractions to the upper layers, namely distributed and shared storage; distributed (a.k.a. private) storage refers to the case where a logical volume (see below) is accessed by a single OSL entity (e.g., a single node), whereas in shared storage multiple OSL entities (nodes) access the same logical volume. It also implements the consistency and security models, defining who (which upper-layer entities) and how (credentials, constraints) one has access to the lower layers.

13.4.4 Storage virtualisation layer

The SVL is responsible for implementing the *logical volume*, or *logical disk*, an abstraction of a direct access storage device (DASD) that may, or may not, correspond to a physical device. For example, the SVL may present a volume out of a disk partition, while hiding other partitions of the same disk; or, it may aggregate two or more disks, or partitions, into a single (larger) volume either by appending them one after the other, or by striping their blocks; or create a highly available volume out of two identical "mirrored disks" – the possibilities are increasing everyday, as this is a fertile R&D ground[2].

Storage virtualisation may be performed at the host, with software products such as LVM [Lew05] or EVMS [Pra02, Lor+05]; or inside storage array boxes (or even directly at the HBA), where the usual options are RAID levels 0, 1, 0/1, 3 and 5, together with LUN virtualisation capabilities; and finally it may also be performed by highly specialised storage appliances that operate at the network (SAN) level [Tat+06, EMCb06].

13.4.5 Storage networking layer

The SNL encompasses the protocols layers required to carry out data transfer and control operations against storage devices; we call the entity that issues operations an *initiator*, while the "addressed" device is the *target*.[3] For the configurations we're interested in this work the initiator is an Host Bus Adapter (HBA) inserted into a host's slot (PCI/-X/-e), and the target is a "disk device"; an example of a SAL protocol is SCSI, either used directly over a parallel cable, encapsulated in FC, or over IP (iSCSI).

---

[2] For a more detailed coverage, refer back to Part II, section 5, "Storage Architectures".

[3] We have borrowed the SCSI terminology, but we will use it across every technology.

It should be noted that communication events that take place among hosts and do not involve storage (e.g., initiators and targets, as defined by block-level storage protocols) are not relevant for this layer; examples include data transfers triggered by clients accessing data in “shared folders” published by NFS or CIFS servers, or host-to-host transfers such as moving data over Ethernet or over RDMA-capable interconnects. NFS and CIFS are especially suited to illustrate the difference, as the communication between NFS/CIFS clients and their servers takes place (depending on where the split among layers is done) either at the FSL or OSL, never at the SNL.

13.4.6 Storage device layer

The SDL deals with the storage devices themselves – *e.g.*, magnetic media (disks, tapes), optical media (CD, DVD), solid state devices, etc. We do not intend to have it thoroughly analysed here, but we must mention it, for completeness of the model.

13.5 Applying the reference model to a few simple cases

The RM we have developed is, we hope, suitable to accommodate not only those file systems covered here, but also other classes (as previously noted); for the small set we will be evaluating in the next section, the RM will be used to highlight their most important characteristics, derive some properties, and establish a classification. As this will be done with a focus on the File System Layer, we now introduce a few simple examples to cover the remaining layers and, at the same time, establish a correlation with the model presented in the previous section. The examples will be presented as follows: we start with a short description on the environment (hardware, disks, RAID levels, FS, etc.), followed by both text and pictorial descriptions on how these “components” are mapped into the RM layers.

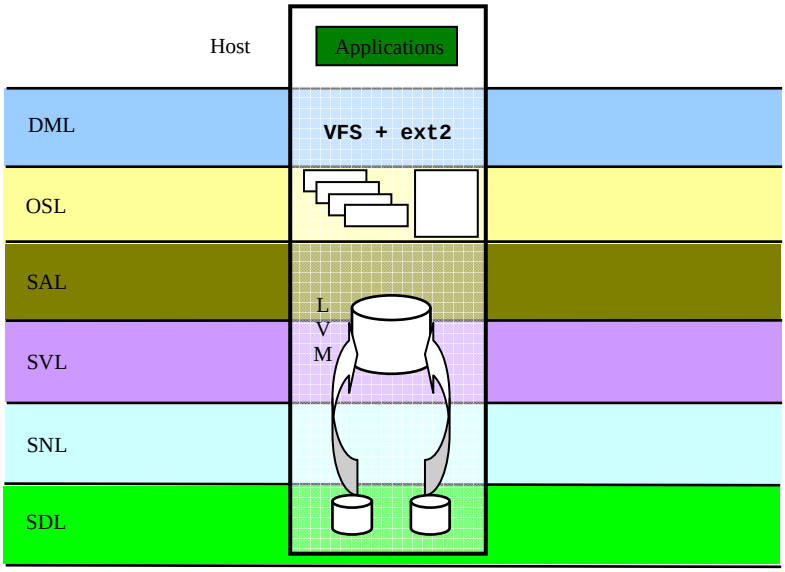


Figure 13.2 Reference Model for Example 13.1: ext2 with LVM-based RAID-0

Example 13.1: A Linux host with two internal SCSI disks aggregated into a RAID-0 volume with LVM software. On top of the logical volume, an ext2 file system is used (Fig. 13.2).

RM: The disks are accessed via the SCSI protocol, implemented at the HBA and its device driver – so, SNL is running at the host; the disks are aggregated by LVM software – thus SVL is running at the host, too. As for the SAL we can say that, conceptually, it does supply a partitioned (unshared) disk volume to the OSL. For an ext2 file system, as well as for the majority of local file system implementations, there is no clear line separating the OSL from the FSL; anyway, the OSL clearly runs at the host and supplies the needed ADTs (inodes, index and data blocks, superblocks, etc.) to the host running the ext2 FSL.

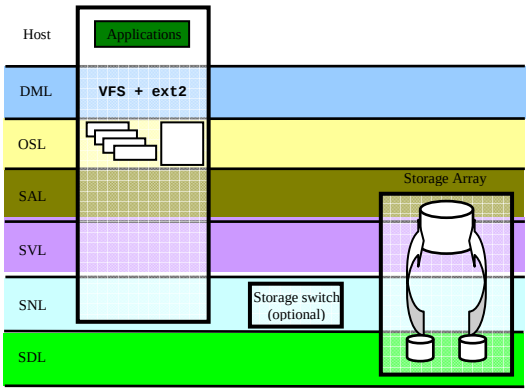


Figure 13.3 Reference Model for Example 13.2: ext2 with array-based RAID-0

Example 13.2: A Linux host with an external disk array supplying a striped (RAID-0) volume. On top of the "disk", an ext2 file system is created (Fig. 13.3).

RM: The array's disk drives are aggregated with SVL software running in the array; the access to the virtualised device is via the SNL protocol (but notice that this is protocol independent: the result is the same for any block based access protocol, *e.g.*, FC, and the optional router is a storage switch; or iSCSI, and the router is an IP router). As for the upper layers, their roles are identical to the ones in Example 13.1.

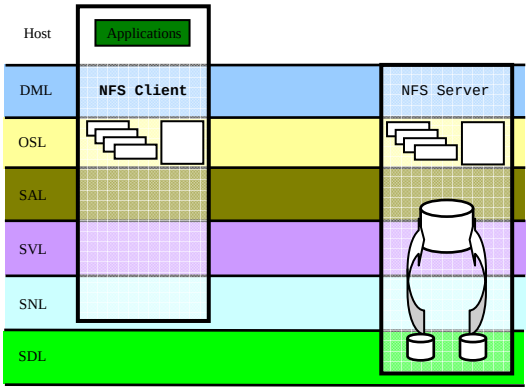


Figure 13.4 Reference Model for Example 13.3: a NFS client and a NFS server

Example 13.3: A host (Linux or other) as a NFS client of a "NAS appliance" with a "built-in" disk array supplying to the host a NFS file system.

RM: The NAS internal array's disk drives are aggregated with SVL software running in the array; the virtualised device is then exported as a NFS-mountable file system. Client/server interaction is performed at the FS (here NFS) Layer.

A final note: we have shown that the proposed Reference Model is able to model both internal and external storage, together with storage area networks and different access protocols; it also models virtualisation both "at the host" and "in the device". Space constraints forbids us from covering more examples here, but the RM-DMA can also model storage objects at higher level than the "disk block", including operating system entities, such as anonymous pages, upon which the Linux /proc file system is built [Bov+05].

14 A Taxonomy for File Systems

Using the reference model we've just introduced (one which, you may recall, subsumes the simpler layered approach we've used before), we now proceed to develop a taxonomy to classify file systems, where we will cover the not only the RM's File System Layer (collocated at the DML) but also the OSL and SAL. The proposed taxonomy will be used in the forthcoming survey on file systems for parallel and distributed architectures; the reader will be asked, sometimes, to look at examples laid out in the next section, to get a better understanding of the proposed taxonomy. The classification will, at the end of this Part, be presented as a table, such as the one below:

	FSL		...	OSL			
	Deployment	Roles		loyment	Partitioning	Scalability	
ext2	Local	---		tralized	---	None	
NFS	Distributed	Asymmetric		---	---	---	
GFS	Distributed	Symmetric		ributed	Homogeneous	Data & MD	
PVFS	Distributed	Asymmetric		ributed	Heterogeneous	Data-only	
Lustre	Distributed	Asymmetric		ributed	Heterogeneous	Data-only	

...

Figure 14.1 Preview of the Classification Table

For each layer of the reference model, we will now identify those attributes that denote major architectural decisions, and thus set different implementations apart; we want to stress

the word “major”: we do not intend to address every possible feature, here and now; in the survey we will study some influential file systems, and, there, as innovative concepts are introduced, other attributes will be added to our classification grid.

14.1 Data Management Layer (DML)

With regard to how the FSL is deployed, there are clearly only two major file system architectures: local and distributed.

> Definition 14.1 **Local File System**: Control and data flows in the file system layer are restricted to a single computing node.

> Definition 14.2 **Distributed File System**: Control and data flows in the file system layer are distributed across several computing nodes.

From the above definitions we assert that, no matter what happens in the layers of the Storage Management Domain, it will not influence our classification at the file system layer; for example, if virtualisation of a storage device is performed by distributed software running across several nodes, as in Petal [Lee+96], but the file system layer only runs in a single node (where the “virtual device” is mounted) the file system is local; furthermore, it should be clear that any file system which is not of the local type is distributed, and *vice-versa*.

Having defined what a DFS is, we now proceed to identify another important characteristic of distributed file systems: symmetry. This attribute sets apart distributed file systems where some nodes play specific roles (such as metadata or data servers) while others perform another, complementary role (such as file system clients) from those where all nodes play the exactly the same role, i.e., run exactly the same set of services.

> Definition 14.3 **Asymmetric DFS**: One or more nodes may assume distinct (file system) roles.

> Definition 14.4 **Symmetric DFS**: All nodes perform the same (file system) roles.

Of course, many more attributes can be used to characterise a distributed file system; as always, in a taxonomy one strives to retain those which are important (in the sense that, here, they really set a DFS apart from others) and discard those which aren’t; we have thus selected *partitioning* and *scalability* as very important characteristics in a DFS. As an interim format we will present those attributes in a tree-like structure in Fig. 14.2, before moving later to a table layout format; for completeness, we will start with DFS at the tree top.

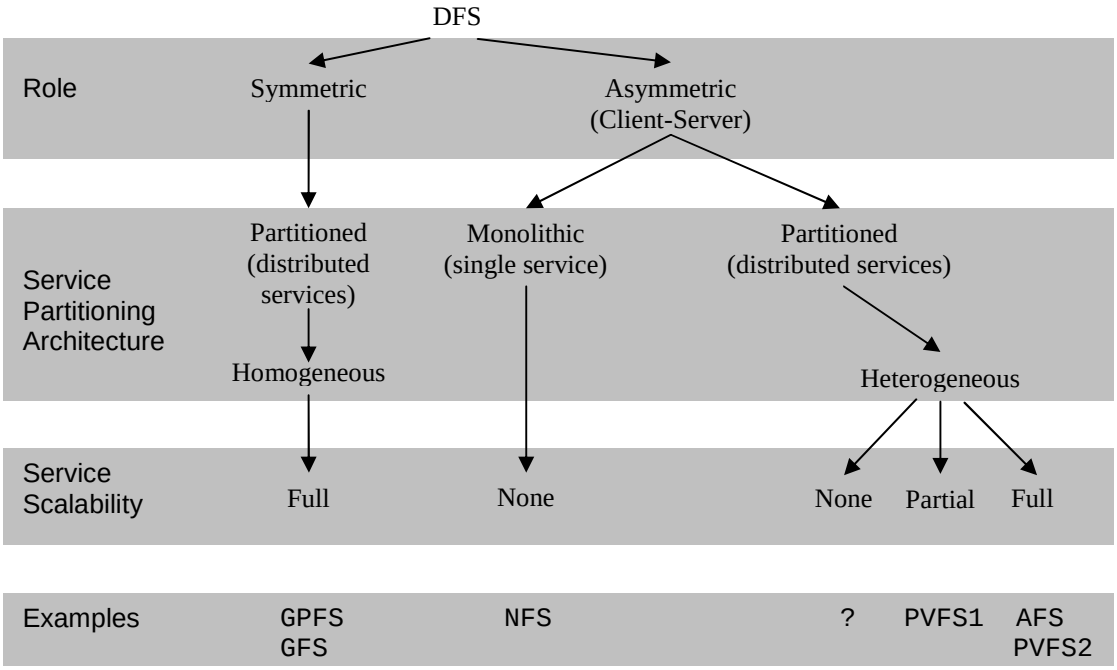


Figure 14.2 Characterising DFS architectures (FSL-only attributes)

Service partitioning allows us to express whether the DFS has some crucial set of services that must be deployed in a single-node as an aggregated/monolithic entity (e.g., an NFS v3 server), or, conversely, they may de deployed across multiple nodes, where those nodes may (PVFS) or may not (GFS) be used to run distinct sets of services.

Definition 14.5 **Partitioned Service Architecture**: File system (server) services run across multiple (server) nodes.

Definition 14.6 **Monolithic Service Architecture**: A single (server) node runs the full set of file system (server) services for that role.

Definition 14.7 **Homogeneous Service Architecture**: All (server) nodes run the same set of file system (server) services.

Definition 14.8 **Heterogeneous Service Architecture**: (server) Nodes may run distinct sets of file system (server) services.

Scalability allows us to assess whether the service architecture is scalable, *i.e.*, supports the addition of more nodes, possibly resulting in a performance increase; as examples of scalable DFS architectures, we've listed GFS and PVFS in Fig. 14.2. GFS supports the addition of

multiple nodes, and we expect increased performance both for metadata and data accesses; consequently we tag GFS' scalability as "full". But PVFS1 is tagged as a "partially scalable" architecture because while we can add more data (I/O) servers, only one metadata server may exist, while PVFS2 allows for multiple metadata servers, so it's "fully scalable".

All file systems used here as examples will be covered in detail later, and we hope that a thorough description of each one will help the reader to get a better understanding of the characterisation attributes and their "values".

14.2 Object Storage Layer (OSL)

Very few file systems (noteworthy exceptions are Lustre and PVFS) allow a clear separation between the OSL and FSL; generally these layers are "glued" together, in a sort of "monolithic" approach.

Just like in the file system layer, deployment is chosen as a major OSL attribute: a centralised object store is one where the OSL is confined to a single node (which may, or may not, be the same node where the file system layer it serves also runs), whereas in a distributed object store, the OSL runs across several nodes.

Definition 14.9 **Centralised Object Store**: Control and data flows in the object storage layer are restricted to a single node.

Definition 14.10 **Distributed Object Store**: Control and data flows in the object storage layer are distributed across several (object storage server) nodes.

Another important attribute is object partitioning across servers and how to accomplish it, *vis-à-vis* homogeneity and scalability; for example, distributed object stores may exist where each object server node plays a specific role (such as metadata server, or data server), thus being heterogeneous, while other object stores may be homogeneous, i.e., all nodes provide exactly the same set of services. The scalability attribute assesses how many servers of a specific type are supported – either one, or many.

Definition 14.11 **Heterogeneous Object Store Partitioning**: Separate server nodes may implement distinct storage object types.

Definition 14.12 **Homogeneous Object Store Partitioning**: Every server node implements the full set of storage object types.

The scalability attribute allows us to characterise how to increase object store capabilities (e.g. bandwidth, capacity, fault tolerance); allowed values are once again *none*, *full*, and *partial* (this one to cover those architectures where some capabilities may be increased while others may not). For example, the PVFS1 architecture supports separate metadata and data servers (thus being an example of a heterogeneous object storage architecture, as objects in the OSL of a metadata server are distinct from those in a data – file – server); it has a fully scalable data architecture, but an un-scalable metadata architecture, as the number of supported metadata servers can not grow (so, it is tagged as a partially scalable architecture, or as "data-only scalable").

Definition 14.13 **Scalable Object Storage Architecture**: The number of object storage servers may be increased and may result in a perceivable increase in the subsystem capabilities (bandwidth, fault tolerance, capacity).

Note that this classification is based on architectural features, not an evaluation of some implementation; *i.e.*, tagging an architecture as fully scalable does not imply that a product's implementation of that architecture is highly scalable, or conversely, that one that is partially scalable is not scalable enough for its application environment.

14.3 Storage Access Layer (SAL)

The SAL is the first (downwards) layer that deals with "raw" storage blocks, how they are accessed and whether they are shared. The two paradigms for storage sharing are: partitioned (a.k.a. private or distributed) and shared.

Definition 14.14 **Partitioned Storage**: All nodes access disjoint sets of storage resources (disks).

Definition 14.15 **Shared Storage**: Nodes access the same set of storage resources (disks).

The shared storage approach may be fully supported by an underlying architecture that goes all the way down to hardware devices, such as multiported disks, or by virtual shared disks (VSD) implemented by resorting to internode communication, much in the same way a distributed shared memory is implemented. For the shared storage case, allowed attribute values in our classification will be *shared disk* (SD) and *virtual shared disk* (VSD); to record the opposite case, unshared disks, we will use the *private* or *partitioned disks* (PD) tag.

14.4 Conclusion

Part IV of this dissertation starts with a question; we ask, in a slightly rephrased way, “what is parallel file system?”, referring to issues raised in figures 12.2 and 12.3. It may seem we have not answered it, after all. The simple answer is that “parallel file system” is, together with other often used labels such as “cluster file system” an imprecise term. The taxonomy we have presented is used in the next section (the focus being the two topmost layers, FSL and OSL) to guide us through some distributed file system case studies, hopefully allowing us to get a clearer picture out of a blurred field created by the above mentioned imprecise terminology or from marketing hype and/or terminology abuse from some FS “pushers”.

We have not covered some important attributes, such as security, resilience, and availability; we simply do not intend to cover them here; these aspects are pervasive to all layers, but time and space constraints deter us from pursuing this line of work. However, if something close to a definitive taxonomy is to be developed, they surely must be tackled.

15 File Systems for Distributed and Parallel Architectures

15.1 Introduction

Again, a simple figure will be used to chart the different file system types we are introducing in this section.

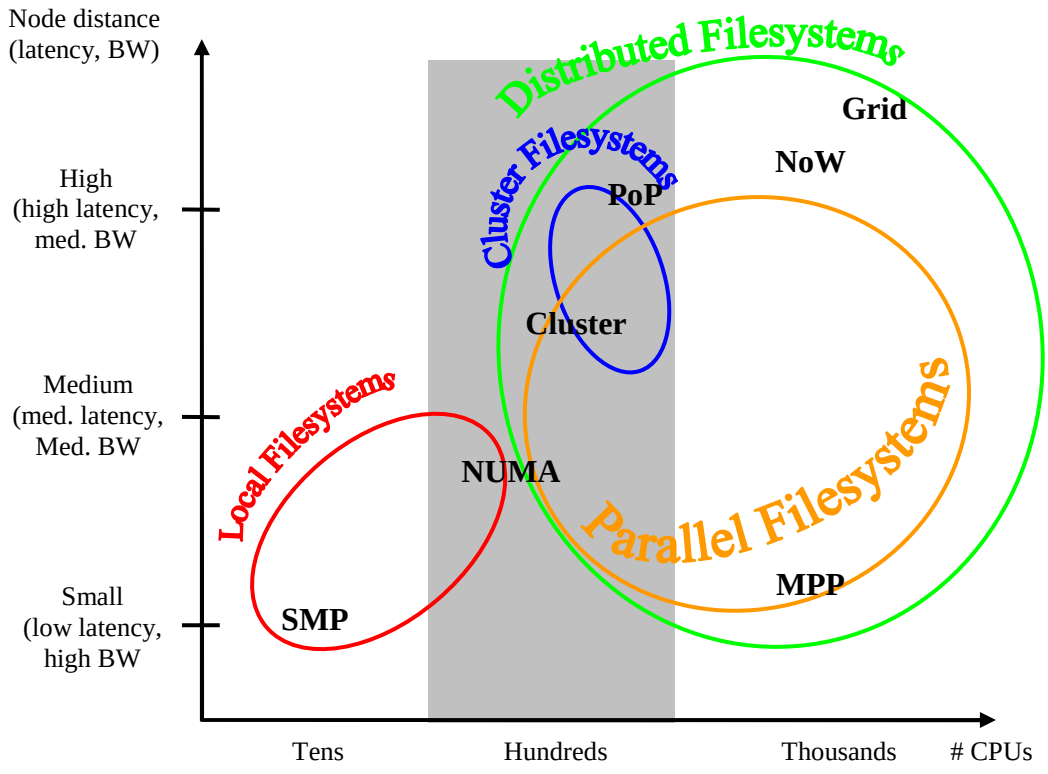


Figure 15.1 File system types “at a glance”

Fig. 15.1 is a two-dimensional grid built along the same axes used before in Fig. 4.1, but now with computing architectures grouped according to the kind of file system they commonly use. To illustrate the placement of "groups" in the chart, we use the SMP and NUMA architectures as examples: in a SMP, disks are completely managed by the node's operating system which runs a standard *local* file system (*e.g.*, `ext2`), while in multiple node NUMA/DSM architectures, either a local or a *distributed* file system may be used – local file systems may be used if nodes are running a true single system image (SSI) operating system (as in tightly coupled cc-NUMAs), while distributed file systems (with or without the "*parallel*" tag) of various "types" must be used in all other cases.

15.2 Local file systems

Local file systems, running both on uniprocessors and SMPs, are well known, and were already covered in Part III (sections 8 and 10), where we looked not only at features they make available to users, but also at some architectural and implementation details. As we're now focused on surveying relevant distributed/parallel file systems, we will no longer refer to local file systems in this section.

15.3 Distributed file systems

The taxonomy we have proposed will now be used to characterise those file systems we deem particularly relevant, ones that have been somehow loosely being called distributed, parallel, and cluster file systems; it will be used in the "**File System Classification**" entries one can find for each case we're surveying. File systems will be presented in the same (left--to-right) order they were depicted in Fig. 14.2.

15.3.1 Symmetric distributed file systems

A distributed file system with a symmetric architecture (with regard to node roles) must be based on shared storage – either physically or virtually shared; we will be surveying GPFS and GFS, two of the most representative "global" file systems – where global is a keyword commonly used to tag shared storage file systems (another, often used terminology is "cluster file system", which is also used to refer to Lustre, an asymmetrical DFS).

15.3.1.1 GPFS

Description: The General Parallel File System, GPFS [Sch+02], is a "closed-source" IBM proprietary "parallel shared-disk file system for cluster architectures", which runs on the AIX operating system based p-Series SMP clusters, and on IBM-certified Linux clusters; for the remainder of this overview we will focus on the Linux version.

File System Classification: GPFS is a fully-scalable, distributed services, symmetric DFS.

File System Architecture: GPFS is targeted to cluster configurations with very high node counts, such as the one depicted in Fig. 15.2; in those large configurations, not all nodes are required to be homogeneous when it comes to storage – some may be SAN-attached while others are not, being used mainly as computational nodes; non SAN-attached nodes, however, may still run GPFS, accessing Network Shared Disks (NSD) which are "virtual shared disks" implemented by a software layer that runs on top of a network infrastructure. Hence, GPFS is symmetric from the file system layer viewpoint, because all nodes access the same set of shared disks.

Storage Architecture: Storage devices are enclosed into storage arrays which are connected to the hosts through a SAN built around FC switches. General-purpose host interconnection is achieved either with IBM proprietary "cluster switches", or via more common infrastructures such as Ethernet, Myrinet or Infiniband; the interconnect is used for all non-FC traffic: application, inter-node locking and, if NSDs are used, FSL/OSL traffic.

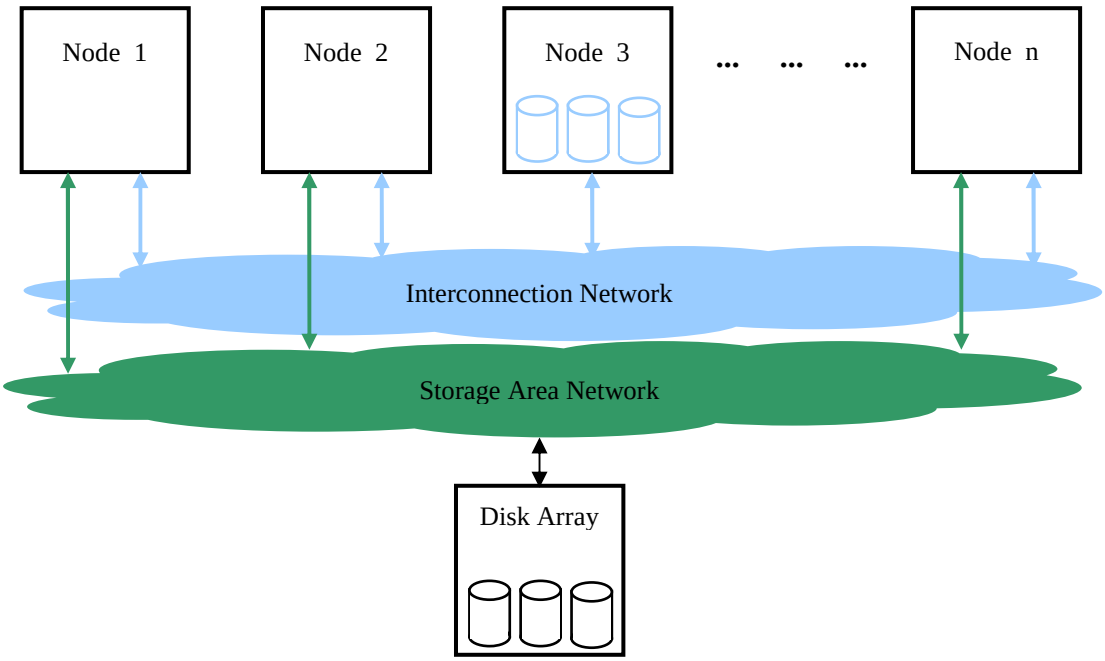


Figure 15.2 Architecture of a GPFS site

Storage System Classification: GPFS is a shared storage system.

Target Application Environments: GPFS is targeted to serve applications requiring I/O bandwidths that exceed the capacity of a single node, and thus have to be executed in a cluster; although supporting correct execution of multi-process applications developed for single node environments, it excels at serving high performance parallel applications.

OS integration: GPFS is provided as a module to be loaded into the Linux kernel, together with another, portability layer module (source available under the BSD license), which plugs it into the VFS and provides a portable interface to the "closed source" GPFS module.

Performance: Data: Under GPFS a file is built from relatively large sized blocks, from 16K up to 1MB; block size is chosen at filesystem creation time. Very small files (and the last data block on large block-sized files) can use sub-blocks, which are 1/32nd of the size of a regular block. Consecutive data blocks may be stripped onto different disk units, to achieve load balancing across host adapters, storage controllers, and disks – *e.g.*, two LUNs on different arrays, each accessed through a different host adapter. **Metadata:** When creating a new filesystem the administrator may individually select which disks will hold the metadata and, consequently, stripe metadata across them, resulting in increased I/O bandwidth for metadata accesses. GPFS supports efficient file name lookup in very large directories (reportedly millions of files) using extensible hashing to organize directory entries. Like Linux (and all UNIX-like file systems) GPFS uses inodes and indirect blocks, but not the allocation strategy of either "standard UNIX" or the "Berkeley Fast File System" cylinder groups. The allocation map is geared towards minimizing conflicts between nodes accessing it; further information can be gathered in [Sch+02]. **Caching:** GPFS for Linux implements its own private cache, independent from the Linux page cache; this is probably to reduce differences between the AIX and Linux versions to a minimum. Caching is used extensively to increase performance, both for data and metadata. **Prefetching:** GPFS recognizes sequential, reverse sequential, and some forms of strided access patterns, and prefetches data into its buffer pool, issuing I/O requests in parallel; in the case of a single-threaded application, this results in requests to as many disks as necessary to achieve the highest bandwidth possible in the fabric. Flushes out of the buffer pool are also carried out in parallel, and the write-behind technique may also be used to increase performance. Irregular access patterns can be hinted by the application programmer, in an attempt to increase performance.

Availability: GPFS is a highly available file system: fault tolerance of storage devices is provided by disk arrays; each node maintains a separate journal for each file system it mounts and all metadata updates that affect file system consistency are recorded in the journal; if a node fails, any other node can immediately start the recovery of the failed node's journal.

Concurrency, Consistency and Sharing: GPFS guarantees single-node equivalent POSIX semantics for most file system operations across the cluster except when "data shipment" mode is used; also, when "time attributes" (`mtime`/`atime`/`ctime`) are modified in a node, it takes some time to propagate them to other nodes. Performance of concurrent file sharing is satisfactory thanks to dynamically elected "metanodes" for centralized management of file metadata; fine grain sharing applications that do not require POSIX semantics, such as MPI-IO applications, can use data shipping to achieve better performance; data shipping resorts to a technique where file blocks are assigned to nodes, in round-robin fashion, so each data block will only be accessed by a single node; other nodes requiring that block will get it shipped from the general-purpose interconnection network. A Distributed Lock Manager

(DLM), which uses both a centralized global lock manager (running on a node of the cluster), as well as local lock managers (running on all other nodes), is used to support both user-level file locking and cache consistency; the global lock manager hands out lock tokens to local lock managers, conveying them the right to distribute locks without the need for message exchange each time a lock is acquired or released.

Locking: GPFS supports POSIX byte-range locking.

Further references: Other than the previously mentioned [Sch+02], interested readers can consult "Concepts, Planning and Installation Guide" [IBMa06], the "Administration and Programming Reference" [IBMb06] and browse the IBM Redbooks site for documents such as Redbooks and Redpapers.

15.3.1.2 GFS

Description: The Global File System (GFS) is a shared-disk file system that runs on Linux clusters. It started out (in 1995) from the desire to exploit FC technology to post-process large scientific data sets, and was implemented on top of Silicon Graphics hardware and the IRIX operating system (GFS-1); later, it was refined, re-implemented, and reported on a PhD thesis (GFS-2) [Sol97]. The key objective for GFS-2 was to design, prototype and test a shared file system based on well known distributed file system research, with a novel extension: the file system consistency mechanism was to be based on Device Locks (D-Locks), a proposed extension to the SCSI standard [Pre+99]. GFS-3 was a re-write and porting to Linux, and a company, Sistina Software Inc., was formed to sell GFS; source code was then closed, and versions 4 and 5 were released. Later, Red Hat Inc. bought Sistina, and source code was released again to the open source community. D-Locks, although proved useful (initial testing was done on modified Seagate disk drives and Ciprico disk arrays), were never included in the SCSI standard and were replaced by another concept, the SCSI Device Memory Export Protocol (DMEP), an extension to the SCSI protocol [Bar+00], which was not accepted, too.

File System Classification: GFS is a fully-scalable, distributed services, symmetric DFS.

File System Architecture: GFS is targeted to medium-sized (currently, 300 nodes) Linux clusters where nodes, which we will call GFS clients, are homogeneous when it comes to storage: all access the same set of SAN-provided shared disks. Fig. 15.2 may be used to depict a GFS setup if all nodes are drawn as SAN-attached.

Storage Architecture: Storage devices are enclosed into storage arrays, and these are connected to the hosts through a SAN built with FC switches or with an iSCSI-capable infrastructure. The only requirement for the general-purpose interconnection infrastructure is that it must support TCP/IP, so anything from plain Ethernet to Infiniband can be used.

Storage System Classification: GFS is a shared storage system.

Target Application Environments: GFS guarantees single-node equivalent POSIX semantics for file system operations across the cluster; in file sharing situations, concurrent readers executed across distinct nodes can benefit from the aggregated I/O bandwidth, but write sharing of a file across multiple nodes has very low performance. GFS is then quite appropriate for situations where one needs sharing of mostly-read data, such as directories containing application binaries and configuration files, or where files are shared, but not concurrently updated across nodes, such as in home directories.

OS integration: GFS is delivered as a single Linux kernel module (but depends on others, such as lock managers); GFS is closely integrated both into VFS and the Linux page cache.

File System Organization, Resources and Metadata: A GFS file system volume is based on SAN-exported LUN(s) and organized into several Resource Groups (RG); RGs are similar to the BSD Fast File System cylinder groups (and Linux `ext2` block groups [Bov+05]) and include a superblock, bitmap, dinodes and data blocks. A dinode is similar to the UNIX inode; key differences are: dinodes use a full file system block (4096 bytes), so files that are small enough can be stuffed into the dinode; and the indirect block tree is uniformly deep.

Performance: Data: in GFS a large file is automatically spanned onto resource groups, and as different RGs may reside in different devices, it is consequently striped out onto different disk units, allowing applications to achieve disk-level parallelism. **Metadata:** The resource group metadata structure previously described contributes to enhance performance by minimizing conflicts between nodes accessing metadata that happens to reside in different RGs. **Caching:** GFS nodes keep both data and metadata cached as long as no other node needs to access the file. Write caching is write-back: modified blocks in cache are marked dirty and flushed by Linux daemons when appropriate, or a by a user requested sync operation. **Prefetching:** GFS resorts to the Linux standard VFS functions to perform device access and populate the page cache; so, GFS prefetchs are, in fact, Linux prefetches.

Availability: GFS is a journaled file system, and each node maintains a separate journal for metadata transactions. Any node can start the recovery of a failed node journal without having to wait for the failed node to come back online – either by detecting, at mount time, a previously “unclean shutdown”, or by detecting an “expired” client node.

Concurrency, Consistency and Sharing: GFS guarantees single-node equivalent POSIX semantics for file system operations across the cluster, so multiple nodes may issue concurrent reads and writes to the same file. GFS locks are used to maintain cluster-wide coherency; in short, every `read()` or `write()` places, respectively, a shared or exclusive lock over the file’s inode for the duration of the operation. Two locking protocols are available – one based on DMEP, and another based on DLM. As no DMEP-capable hardware exists, a user-level TCP/IP daemon that implements a DMEP server is provided. GFS clients specify the locking protocol they wish to use at filesystem mount time.

Locking: GFS supports POSIX byte-range locking.

Further references: The Red Hat site (www.redhat.com) has both Administrator and User Guides available for downloading.

15.3.2 Asymmetric distributed file systems

Asymmetric distributed file systems (also designated client-server DFSs) are those where a functional separation exists between *server* nodes, which do run the "server part" of the file system layer and provide services that are used by *client* nodes, which run the "client part" of the FSL to access stored data.

In a distributed system, a protocol is defined to regulate interactions among members; in a DFS, it is used to specify how the client's FSL talks with its peers – and it may define that interactions strictly happen between a client and a server, such as in NFS, or that they may be of a broader nature, and involve not only multiple servers, as in PVFS, but also other clients.

15.3.2.1 Single-server asymmetric DFS

A single-server asymmetric distributed file system is, as its name suggests, a client-server DFS where multiple clients access stored data through services provided by a single server.

15.3.2.1.1 NFS

Description: The Network File System is one of the most well-known client-server distributed file systems; once extensively used in all domains, it has been replaced by Microsoft's CIFS (Common Internet File System), namely in Windows environments. NFS originated around 1984 at Sun Microsystems, and has been improved over the years; the currently most widespread release is NFS version 3 (NFSv3), which is available for all general purpose operating systems, and even for some more "esoteric" ones; version 4 has been available for some time, but has not yet displaced v3. A parallel version (similar to PVFS) initially designated Parallel NFS, or pNFS (now NFS 4.1) was scheduled for inclusion in the Linux mainstream release in 2008, but has yet to appear.

Classification: NFS is an un-scalable, single-service asymmetric DFS.

File System Architecture: As shown in Fig. 15.3, NFS is a client-server DFS where a single server is accessed by multiple clients over a TCP/IP interconnection network; a typical NFS usage scenario resorts to UDP to perform data transfers between clients and servers, while more demanding environments (e.g., HPC) use TCP to perform data transfers.

Storage Architecture: A typical NFS server is a single SMP node with DAS storage, *i.e.*, with its own local disks; storage may either be internal or else LUNs provided by external disk arrays. NFS may be also found in "bridge" configurations, *e.g.*, to gain access to data stored/mediated by other file systems. For example, a GFS file system may be exported on a

(cluster) node and then mounted by NFS clients – however, this is not an architecturally different configuration, as the exporting node becomes "the" single-server.

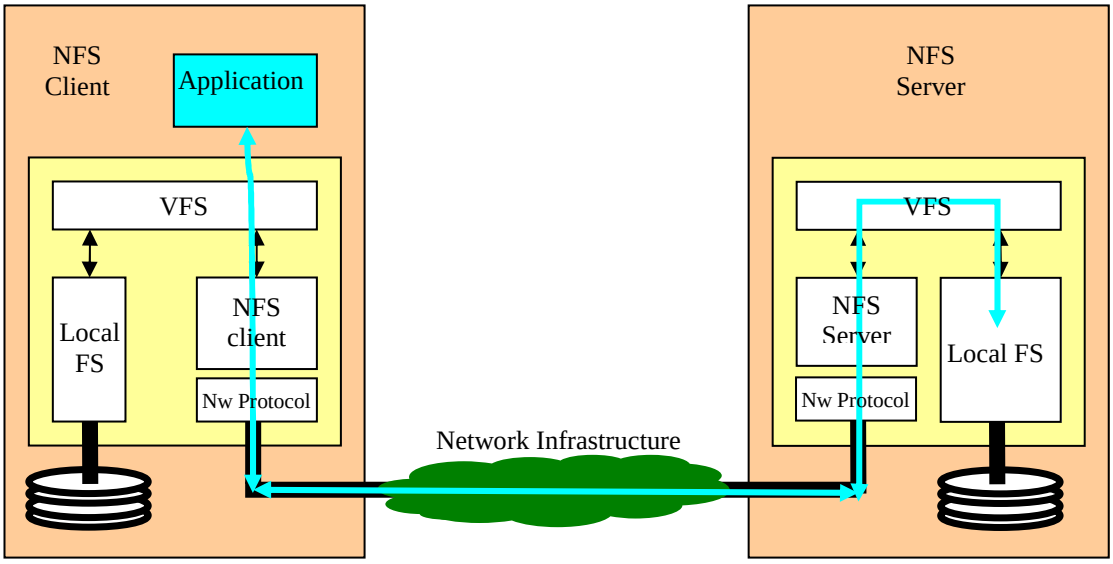


Figure 15.3 Architecture of a NFS site

Target Application Environments: NFS is particularly well suited to support environments that have moderate bandwidth requirements and infrequent sharing events, such as having, e.g., a head-node server which exports user home directories and application binaries to client cluster nodes. NFS does not comply with POSIX single node equivalent semantics; it does, however, support file locking by resorting to a companion user level daemon, called `lockd`. Through the use of locking (even in situations where one should not normally need to use it) it is then possible to guarantee correct execution for processes sharing a file, albeit at the expense of reduced performance caused by lock traffic and repeatedly flushing of data blocks cached in clients.

OS integration: As shown in Fig. 15.3, the NFS client code is tightly integrated into the kernel (in the UNIX/Linux VFS); conversely, server code needs not to be in the kernel, although most implementations do it, for performance reasons.

File System Organization, Resources and Metadata: As any other DFS, NFS is both a file system and a protocol for client-server interaction [RFC 1813]. As NFS has been designed to be OS independent [Paw+94], the specifics of the client-side FS organisation, semantics, etc., are implementation issues but its client-side part, for a given OS, is usually implemented as to mimic some "favourite" local FS, thus easing the burden for users, which will welcome the similarities. So, in a Linux host a NFS filesystem will look like a local FS, such as ext2, and similar metadata structures (inode, index blocks, etc.) will be visible for local users of the (remote) file.

Performance: Data Caching: NFS clients may choose whether they want to cache data, for enhanced performance but at the risk of using stale data, as referred in 11.3 "Case study:

caching in NFS". The NFS protocol does not address client caching and cache coherency, although return data provided by the protocol calls does provide some help for implementations. A strategy commonly used by NFSv2 clients to reduce the chance of using stale data is to ask the server for the file's modification time (`mtime`) periodically; if it matches the one stored in the client, cached data is valid. This procedure does not guarantee full consistency because the client only asks for the file's `mtime` on file opens and whenever cached attributes (which include `mtime`) expire; between those events, a second client may modify data that is cached by the first client. Weak Consistency is a policy available on NFSv3 that may offer a performance increase over the previously described NFSv2 strategy, but still suffers from the stale data problem [Paw+94]. **Metadata Caching:** Caching of metadata at clients is performed as a result of (remote) access operations; metadata cached objects include, among others, directory entries, file handles and file attributes. Policies used to promote consistency are the same ones previously described. **Prefetching:** Data prefetching occurs both at the client and in the server; at the client, prefetching may be triggered by assigning successive "NFS read calls" to distinct threads (`biod`s); at the server, as a consequence of standard Linux read-ahead behaviour; if the server can process the requests arriving at the `nfsd`s in parallel, the client will see a high prefetch rate; if not, it will, at least, benefit from the reduced latency that results from the overlapping of the client requests. Complementary to the read-ahead is the write-behind; the same threading approach can be used by the client to submit multiple "NFS write calls" against the server [Cal00].

Availability: The NFS protocol is stateless, *i.e.*, each request carries enough information to be processed independently from other requests, past and future. Server crash recovery is then simple: a client keeps retrying a request until the server responds; a client is not able to differentiate between a slow server and one that crashed and was subsequently rebooted [Paw+94]. Failover NFS solutions do exist, where data is stored in an external array and when a "primary NFS server" fails, the dormant backup server (a configuration usually called active-passive) will mount and export the disks (and it may even grab the primary server's IP address); Highly-Available NFS (HA-NFS) [Bhi+91] is a similar solution that uses an active--active, load balancing approach, with both servers acting as independent NFS servers (each one is a backup of the other), dual-ported disks (made obsolete by today's disk arrays) and mirroring software, to be able to recover from disk failures.

Concurrency, Consistency and Sharing: Default consistency semantics for NFS can be very easily stated: data written by a client is noticed by others at most 3 seconds later; metadata (file, directory, symbolic link) changed (created, removed) by a client will be noticed by others at most 30 seconds later. These are NFS' *time-to-live* policies for cached data and metadata: default values are respectively 3 and 30 seconds, but the minimum value

could be as low as zero – no caching at all; these policies are not a part of the NFS protocol, but are fully dependent on client's implementations [Cal00].

Locking: NFS' consistency model does not, per se, provide sufficient guarantees for consistent updating between cooperating clients in the absence of explicit locking, as won't any other FS. Advisory byte-range locking is provided by the Network Lock Manager (NLM) in conjunction with the Network Status Monitor (NSM); NLM provides the locking calls and maintains state, while NSM provides information about crash/restart so that NLM can initiate lock recovery. Locking and caching, when used together, create some delicate problems, *e.g.* a client locks, writes, and then unlocks the first byte of the file, while another does the same for the second byte; the "performance road" would be to get the first 8KB of data for the first client, and change the first byte while the same sequence was repeated for the second client, but now acting upon the second byte. At the end, the 8KB of data would be pushed to the server at distinct times, possible resulting in a lost update. A solution adopted by Solaris NFS client implementations is to disable caching and transfer the exact amount of data requested by the clients [Cal00]. Another would be to extend the lock range to cover the full amount of data transferred (but that would decrease the degree of concurrency).

Further references: NFS is extensively covered in of books, papers, technical reports, etc. and, furthermore, several implementations have their source code freely available; therefore we feel no more references are necessary.

15.3.2.2 Partitioned asymmetric DFS

Partitioned asymmetric distributed file systems (PADFS) distinguish themselves from the previous group, single-server asymmetric, because in PADFSs the server side of the file system service is itself distributed across multiple nodes. A PADFS where all server nodes must run exactly the same set of services, is a homogeneous PADFS, while another, where some nodes may run some services, such as data access services (data movers), while others run different set of services, such as directory services, is designated heterogeneous.

15.3.2.2.1 PVFS

Description: The Parallel Virtual File System (PVFS)[1] is an open source file system that was developed at Clemson University and Argonne National Laboratories; its primary objective is to provide high performance I/O for MPI applications running on COTS Linux clusters [Car+00]. PVFS is widely used today, including in environments where it is not the most appropriate FS (e.g., in environments, where Samba/CIFS is used on top of PVFS to provide "shares" for Windows PCs) because it's free and offers good scalability, provided some conditions are met. In production mode PVFS I/O nodes should store data on external

[1] The PVFS discussed here is the latest version, called PVFS2, available since November 2004.

disk array LUNs; that's because if an I/O node fails, then its LUNs can be "transferred" to another node, and PVFS may be restarted; PVFS is not, contrary to "popular belief" (lying around on several HPC-oriented web sites), an inexpensive solution anymore.

Classification: PVFS2 is a fully-scalable heterogeneous partitioned asymmetric DFS (PVFS1 was partially scalable, as only one metadata manager was supported).

File System Architecture: As shown in Fig. 15.4, PVFS has a client-server architecture where multiple clients (compute nodes), data servers (I/O nodes) and metadata servers communicate via a general-purpose interconnection network; typical networks found on small cost effective PVFS installations use Gigabit Ethernet, while more demanding ones resort to high bandwidth low latency interconnects, such as Infiniband or Myrinet (using their native transport interfaces, not just simply TCP/IP on top of them).

Storage Architecture: PVFS is based on metadata and I/O nodes with private disks, which may be either DAS internal disks, or LUNs provided by SAN-attached disk arrays; it is, consequently, a file system for a distributed storage architecture.

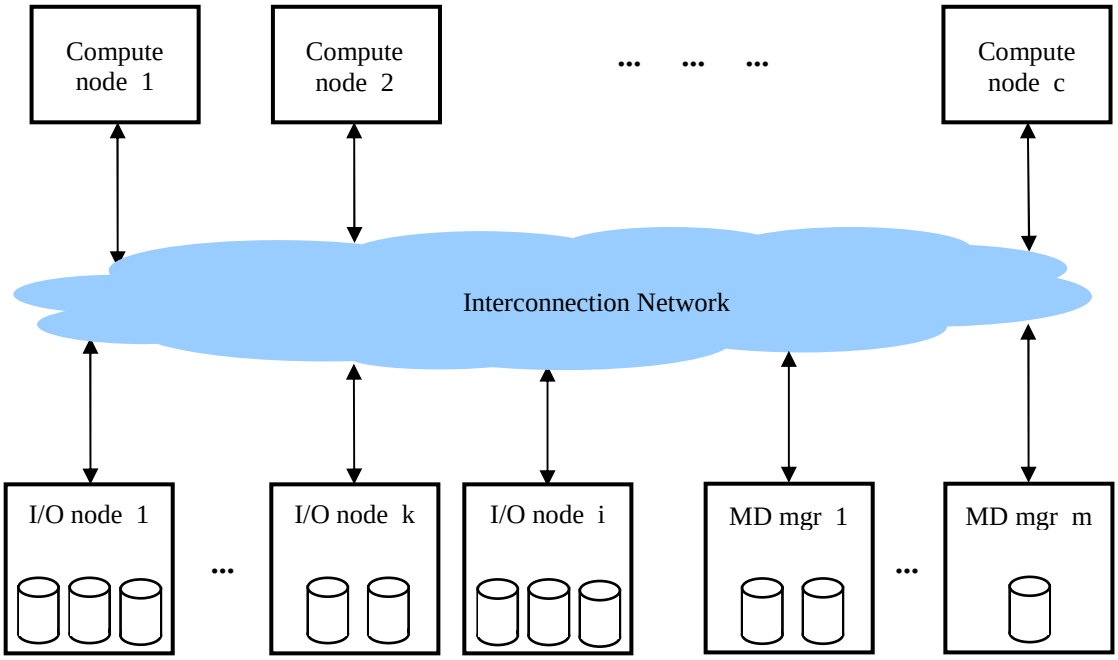


Figure 15.4 Architecture of a PVFS site

Target Application Environments: PVFS is particularly well suited to support MPI parallel applications that require high bandwidth access to data; it efficiently supports concurrent access both to distinct files, and to distinct, non-overlapping regions of a single file. PVFS is accessible via two[2] different APIs, each addressing a different need: a standard POSIX interface (with the exception of locking), and an MPI-IO interface. The POSIX

[2] Access to the native API referenced in PVFS1 papers is not documented under PVFS2, and traffic on the pvfs2-users mailing list discourages its use.

interface extends the applicability of PVFS to support generic, *i.e.*, non-parallel applications, at the probable expense of reduced performance [Vil+04]; the MPI-IO interface supports parallel MPI applications, and is the main PVFS *raison d'être*.

OS integration: Both server code and client code for the MPI-IO interface, run in user space; only the client's POSIX interface is implemented as a Linux kernel module that plugs into the VFS and thus allows applications to use a subset of the POSIX file interface API.

File System Organization, Resources and Metadata: A typical PVFS installation has a few, say `i`, I/O nodes, where each one contributes with a locally managed (using a local file system, *e.g.*, ext2) storage area of size `s`, to form a "storage pool" of size `S = i x s`, and one or more metadata managers to store and track metadata information about existing files and support the filesystem hierarchy. When a client wants to read an existing file, it (the PVFS library) contacts the metadata manager, which returns the file's base node (`b`), striping size (`s`), and number of stripes (`n`); then, the client gets the first stripe of data from node `b`, the second stripe from node `b+1`, etc., up to the last stripe, which comes from node `b+n` (a file striped across all nodes would have `n = i`).

Performance: Data striping: a file is striped across I/O nodes either according to a predefined striping policy (for the POSIX interface), or via parameter values supplied when the file was created (MPI-IO interface, only). **Metadata striping:** PVFS supports multiple metadata servers, each one handling a non-overlapping partition of the full metadata space and storing information about files and directories it manages in a Berkeley DB [Ols+99] database. **Data Caching:** PVFS clients do not cache data – a decision taken to greatly simplify the PVFS implementation – but I/O servers automatically benefit from the standard Linux page cache; client writes reaching a server are submitted as local file system writes, and thus share the same cache policies, *i.e.*, cached pages are marked dirty and periodically flushed by a Linux daemon, or immediately as a result from a user initiated sync or close. **Metadata Caching:** Caching of metadata is tuneable at clients (from 0 – no caching, up to some duration, in seconds) and handled by the Berkeley DB at the metadata servers. **Prefetching:** Data prefetching only occurs at the I/O servers, as a consequence of standard Linux read-ahead behaviour. There is, however, a sort of metadata prefetching at the client's POSIX kernel module: to reduce multiple network "transactions" when fetching directory entries (*e.g.*, for the `ls` command) a directory read operation issued at the kernel module triggers the metadata server to perform an aggregated read of at most 64 entries and report those to the client kernel module, where they will be used to fill up VFS dentries; however, it is not clear if this feature reported on PVFS1 [Vil+04] is still available on PVFS2.

Availability: When using internal disk storage PVFS does not withstand any permanent server failure, be it data or metadata. However, with external storage and failover software,

one can recover from a node failure by re-mounting LUNs on another node and re-starting the PVFS daemons. Some attempts to provide software based replication solutions that still keep the internal disk storage approach have been proposed, such as CEFT-PVFS [Zhu02].

Concurrency, Consistency and Sharing: PVFS guarantees consistent data from file system operations across the cluster, allowing concurrent readers and writers, as long as they operate on disjoint locations within the file (as, then, reads and writes are atomic with regard to each other).

Locking: Recently, a locking API was proposed for the MPI interface [Chi+07].

Further references: The documentation page on the PVFS2 site (www.pvfs.org/pvfs2).

15.3.2.2.2 AFS and DCE/DFS

Description: The Andrew File System (AFS) is a client-server distributed file system, pioneered at Carnegie Mellon University in the mid-eighties, and supported and developed as a product by Transarc Corporation (now IBM Pittsburgh Labs). IBM branched the source of the AFS product (in Sep, 2000), made a copy available for community development and maintenance, and called the release openAFS. The OSF (Open Software Foundation, now Open Group) Distributed Computing Environment (DCE) endorsed a distributed file system, called DCE/DFS, which was also based on AFS (DCE does not seem to be supported by any vendor or group for quite some years).

Classification: Both AFS and DCE/DFS are fully-scalable heterogeneous partitioned asymmetric DFSs.

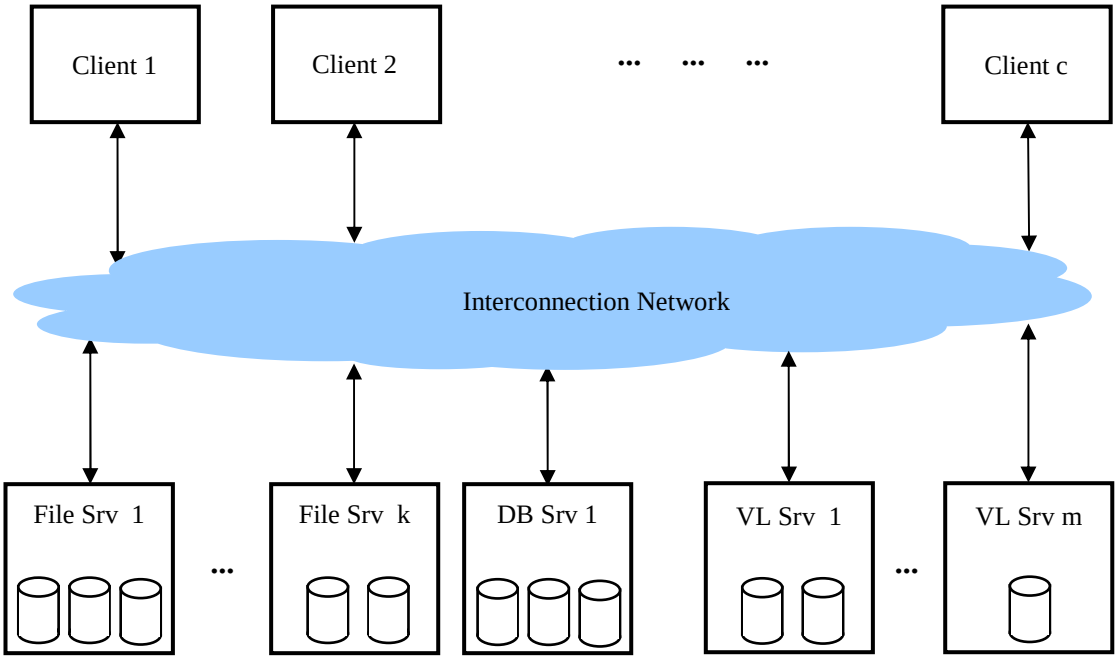


Figure 15.5 Architecture of an AFS site

File System Architecture: As shown in Fig. 15.5, both AFS and DCE/DFS have client-server architectures, where multiple clients and servers communicate over an interconnection network. They are different from other distributed file systems in the degree of specialisation they confer to servers: each File Server Machine[3] runs a local file system that holds a portion of the global filesystem tree and exports it, contributing with the stored data (files) and metadata (directories) to the global filesystem; Volume Location Servers maintain databases that are queried by clients to discover which File Server holds the volume containing the file (see File System Organization, Resources and Metadata below); Database Server Machines maintain replicated administrative databases (configuration and runtime information); other servers (*i.e.*, services) exist, such as Authentication, Protection, Update, Backup, etc., but we are not covering them in this short survey.

Storage Architecture: AFS and DCE/DFS are based on I/O nodes with local disks, be they DAS internal disks (as in Fig. 15.5), or LUNs provided by SAN-attached disk arrays; it is, consequently, a file system with a distributed disk architecture.

Target Application Environments: File sharing, in AFS' view – motivated by research into UNIX file usage patterns mostly on academic environments – is that users infrequently perform concurrent read/write sharing of a file, but, conversely, frequently read-share the same binaries, *i.e.*, executable files; therefore, AFS efficiently supports concurrent access to distinct data files, and read-sharing of file among concurrent users. It does not, however, support any flavour of file locking. AFS is particularly well suited to support environments that require highly available access to data, as its architecture includes automatic replication, data backup, and redundant services distributed across multiple machines. DCE/DFS is quite more general, and provides POSIX single-node equivalent semantics.

OS integration: AFS is supported on Linux, several UNIX flavours, and Windows. AFS client code, the Cache Manager, plugs into the Linux' VFS and therefore allows applications to use a subset of the standard POSIX file interface, while AFS server code has kernel-level as well as user-level components. DCE/DFS' Linux integration is similar to AFS'.

File System Organization, Resources and Metadata: A typical AFS or DFS installation, such as the one depicted in Fig. 15.5, has a few File Server machines, where each one contributes with a locally managed (using a local file system, *e.g.*, `ext2`) storage area called *partition*; each partition then holds one or more *volumes*, and each volume stores a portion of the global AFS/DFS filesystem hierarchy in the form of data (files) and metadata (directories, etc.). When a client wants to read an existing file, it (the Cache Manager) contacts a Volume

[3] We will deviate for a moment from the usual terminology of servers (computers) running services, and adhere to the AFS terminology of machines running servers (processes); the reader is warned that although DFS and AFS concepts are quite similar, they use different terminologies.

Location Server, which informs the client about the file's File Server location; from that moment on, all traffic is exchanged solely between that client / server pair.

Performance: Data and Metadata Load Balancing: both in AFS and DCE/DFS, a file system "object" (file, directory) must be contained in a single volume, so it can't be striped across multiple file servers; so, these are not solutions for high performance I/O to a single file; however, load balancing can be achieved by separating regions in the file space across multiple volumes, and then segregate volumes to multiple file servers. **Data Caching and Prefetching:** early versions of AFS clients performed caching at file granularity, *i.e.*, when they opened a file, a private copy of the whole file was fully transferred to the client's local cache, implemented either in memory or in a local disk (depending on the client's configuration); recent AFS and DCE/DFS versions, however, perform file caching in contiguous chunks of 64KB for file data, and 8KB for memory data (these are default values).

Availability: Both AFS and DFS provide an architecture where a complete fault tolerant solution can be built, at the expense of replication of data (volumes), services (multiple servers), configuration databases, etc.; it is up to the site administrator to choose the desired level of availability, and appropriately configure the site servers.

Concurrency, Consistency and Sharing: AFS semantics, known as private copy until close [Hog+02] (a.k.a. session semantics) is highly scalable, under the assumption that read/write sharing is a rare event, and does not guarantee client side caches: each client is supplied, at `open()` time, with a copy of the file, a *callback* promise; if the node modifies the cached copy, when the `close()` is performed, the modified file is sent back to the server, which calls back other clients so they can invalidate their cached copies on the next (re-)open. If two or more clients are concurrently modifying their local copies, the last one to perform the close operation is the one who gets its file onto the server. AFS' version of the copy-on-close is an improvement over the standard version because an AFS client can keep on using a cached copy until the callback expires or is reclaimed by the server; otherwise, it does not need to contact the server, and LAN traffic is reduced. As for DCE/DFS, it uses a complex token manager to provide POSIX single-node equivalent semantics [Aga95].

Locking: From a practical perspective, no file locking in available in AFS – it only supports full file locking, and the lock state is guaranteed to be visible only within the node that initially locked the file; however, DCE/DFS supports POSIX advisory locking [Sal96, And96].

Further references: The AFS Administration Guide and other IBM AFS documentation, is available online at [IBMafs], and also available from the openAFS documentation page at [openAFS]. For DCE (including DFS) the Open Group's DCE bookstore [OG-DCE] has the most up-to-date documentation.

15.3.2.2.3 Other partitioned asymmetric heterogeneous DFSs

There are several file systems in this class, besides those surveyed here, PVFS and AFS; the two most important ones are the new NFS v4, and particularly the NFS v4.1, also known as Parallel NFS, or pNFS [Hil+06], and Lustre [Bra03]. pNFS is architecturally similar to PVFS1: multiple clients, multiple data servers, and a single metadata server; therefore, pNFS is only partially scalable, as it does not support addition of more metadata servers. Conversely, Lustre is architecturally similar to PVFS2: multiple clients and multiple data and metadata servers; therefore, Lustre is fully scalable.

15.4 Conclusion

We conclude our survey with Table 15.1, a classification of all file systems and storage paradigms that make up the case studies previously presented, plus some that were briefly mentioned; we do it according to the taxonomies proposed for the File System (FSL) and Object Storage (OSL) layers. We also include the ext2/3 file systems to show how a local file system compares with distributed file systems.

<table>
<tr><th></th><th colspan="4">File System layer</th><th colspan="3">Object Storage layer</th></tr>
<tr><th></th><th>Deployment</th><th>Roles</th><th>Partitioning</th><th>Scalability</th><th>Deployment</th><th>Partitioning</th><th>Scalability</th></tr>
<tr><td>ext2/3</td><td>Centralised</td><td>N.A.</td><td>Monolithic</td><td>None</td><td>Centralised</td><td>Aggregated</td><td>None</td></tr>
<tr><td>GFS
GPFS</td><td rowspan="5">Distributed</td><td>Symmetric</td><td>Homogeneous</td><td>Full</td><td>Distributed</td><td>Aggregated</td><td>Full</td></tr>
<tr><td>NFS3</td><td rowspan="4">Asymmetric</td><td>Monolithic</td><td>None</td><td>Centralised</td><td>Aggregated</td><td>None</td></tr>
<tr><td>NFS4.1/pNFS
PVFS1</td><td rowspan="3">Heterogeneous</td><td>Partial</td><td rowspan="3">Distributed</td><td rowspan="3">Heterogeneous</td><td rowspan="3">Full</td></tr>
<tr><td>PVFS2
Lustre</td><td>Full</td></tr>
</table>

Table 15.1 Classification of some well known file systems

Part V:

The parallel Cluster File System proposal

In this Part we start with a critique of traditional shared-disk cluster file systems, listing their features and benefits as well as limitations; while we specifically refer to Red Hat's GFS here, remarks also apply to other CFSs. To overcome those limitations, we propose a new architecture for shared-disk CFSs, one that moves data sharing from the device to the file system cache while preserving POSIX semantics across cluster nodes; we call it the "parallel Cluster File System", pCFS. To validate whether fundamental ideas, e.g., using the LAN as a secondary path to move data among nodes, were sound, we have developed a pre-prototype and some preliminary tests were carried out.

16 pCFS, the parallel Cluster File System

16.1 Introduction

As we previously pointed out, it's easy to perceive a division among academy/research and general IT communities on both storage and file systems endorsement. From a storage point of view, the former group favours an approach based on I/O nodes with internal disks while the later adopt SAN infrastructures based on disk arrays supporting a variety of RAID levels (providing users with high availability, a basic requirement for "near continuous operation" of their data centres). Also, file systems used in these environments are quite dissimilar: IT choices span from the run of the mill ext3 and NTFS to the more sophisticated cluster file systems, such as Red Hat's GFS or Oracle's OCFS, Oracle Cluster File System [OCFS, Fas06], both supporting continuous access to stored data even in the presence of node failures; on the other hand, HPC communities do prefer parallel file systems such as PVFS or GPFS.

Definitely, the above mentioned "advanced" – parallel, cluster shared disk – file systems perform very well in their target environments, provided that applications do not require some "lateral features", e.g., no file locking on parallel file systems, and no high performance on cluster-wide write-shared files on CFSs. In brief, we can say that no approach has provided high levels of reliability and performance to both worlds.

Our pCFS *proposal makes a contribution to change this situation: the rationale is to take advantage on the best of both – the reliability of cluster file systems and the high performance of parallel file systems*. We don't claim to provide the absolute best of each, but we aim at full POSIX compliance, a rich feature set, and levels of reliability and performance good enough for broad usage – e.g., "regular" as well as HPC applications, support of clustered DBMS engines that may run over regular files (i.e., the engine should not be required to bypass the file system to access clustered raw partitions), and video streaming.

16.2 Sharing and caching

File sharing is something that sets apart IT and HPC environments; having watched for quite some time both sides of the "fence", we have noticed that

- The IT paradigm of choice is primarily one of file system sharing, *not* file sharing; in the typical IT environment multiple clients access the same file system, sharing some of its "folders" (directories), but use files either exclusively or share them with other readers. Notable exceptions are DBMS engines (where multiple processes running in the same node RW share a set of files) and collaborative applications. File systems endorsed in this environment must efficiently support various file locking paradigms, such as POSIX locks (to support a broad range of applications), and mandatory locks or leases (for more "collaborative-oriented" ones).

- In the “HPC world” there are two major file sharing patterns:
  - Read sharing of input data file(s), e.g., for parameter scanning or pattern searching, usually in embarrassingly parallel applications.
  - Read/write (or even write/write) sharing of a single file by processes accessing disjoint (i.e., non-overlapping) segments of the file.

Sharing semantics in a distributed file system is of paramount importance because it is closely related to caching, something that we’ve discussed in section 11 “Distributed File Systems”, particularly 11.1 to 11.3; and caching is one of the most important ways to enhance performance in any case, let alone a distributed file system: it gives applications a low latency and high bandwidth path to data. For pCFS to succeed in both environments (HPC and IT) it must support byte range locking and make good use of caching.

16.3 Caching in pCFS: an introduction

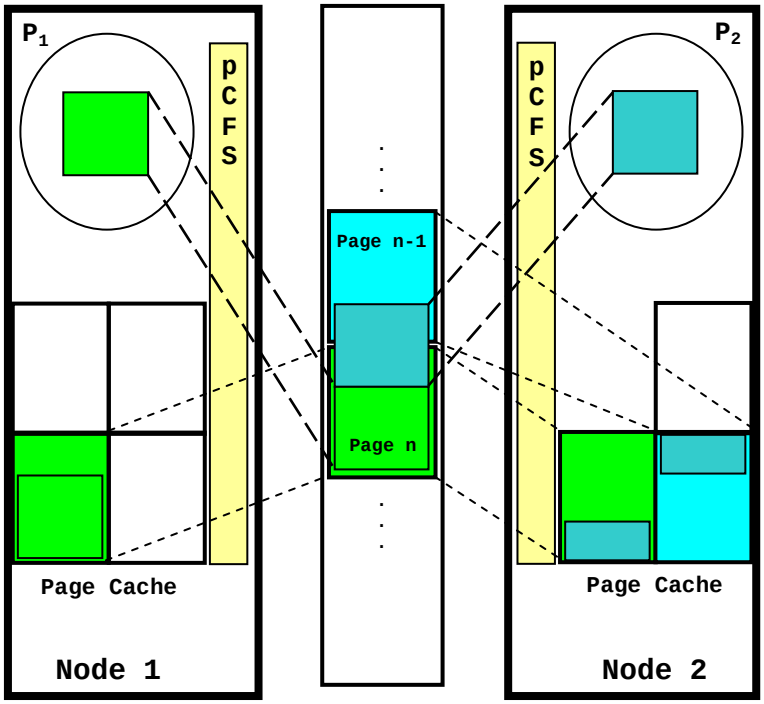


Figure 16.1: pCFS page caches are not fully coherent across all nodes

The way cache is implemented in pCFS is explained with the aid of Fig. 16.1, where P_1 is a reader while P_2 is a writer. P_1 is allowed to access a file segment that starts “in the middle” of page `n` and ends somewhere further down; conversely, P_2 is allowed to access a file segment that ends precisely in page `n`, just before P_1’s segment starts. Notice that page `n` is coloured light green while page `n-1` is light blue, and that P_2 writes onto page `n`, so a dark blue record is superimposed on both pages (because the records are not page-aligned); however, the cached images are different on both nodes: data cached in node 2 is up-to-date, as both page `n`

and `n-1` have been superimposed by the record's respective fragments, but page `n`'s cached image in node 1 is a "before image" (data modified by P_2 does not show up).

Caching, as implemented by pCFS, does preserve POSIX sharing semantics because it is complemented by the byte range locking mechanism which prevents processes, such as P_1 above, from accessing data belonging to other processes' segments (which we call *regions*), such as P_2's. Only when P_2 removes its region lock, it's time to enforce coherence: modified data is written back to disk and other nodes are sent invalidation messages for that page. Notice that these "slight un-coherencies" can only develop in "frontier pages" between segments that are accessed by processes running in different nodes, and which carry incompatible lock states, such as read/write or write/write.

The whole subject of pCFS caching will be detailed in Part VII "pCFS implementation", but while we're on it let's just briefly describe how things would be handled if both P_1 and P_2 were writers: there, writes triggered to the frontier page by the last process to lay out its lock would be forwarded – *shipped* – to the other process' node, and there it would be merged into the node's page cache (the node would be called the owner of that page).

In short, pCFS, while adopting files system techniques – such as caching and locking – that are applicable both to distributed as well as shared-disk architectures, nevertheless uses them in innovative ways; when compared to other file system architectures, pCFS' major differences are:

- pCFS uses a *cooperative cache* approach, a technique that has been used in file systems for distributed disks (e.g., xFS [And+96]) but, as far as we know, was never used either in SAN based cluster file systems or in parallel file systems. As a result, pCFS *may use all infrastructures (LAN and SAN) to move data*.
- pCFS uses *fine-grain locking*, allowing the user to explicitly lock byte-range regions instead of the whole file, and that fine-grained approach is carried out down to the FS *implementation*.

16.4 Cooperative Caching

pCFS uses cooperative caching: where a local file system only has its host cache to access, in a distributed file system such as pCFS, *a node can access data that is cached in the memory of another node* – and that's exactly what pCFS does, as described in [Lop+05, Lop+06]. Accessing another node's cache may improve performance because latency on a LAN is about one order of magnitude smaller (a hundred µs at worst) than for a local disk (a few ms).

File systems for distributed disk architectures move data (and lock/coherency traffic, when applicable) over LAN interconnects, while those for shared disk architectures[1] only use the LAN to move coherency (control/lock) traffic and the SAN to move data. Using cooperative

[1] We have not found any file system – shared or not – that uses both infrastructures.

caching means that pCFS can effectively explore all available infrastructures (LAN and SAN) to move data and, therefore, its I/O bandwidth should be able to approach the sum of all interconnect bandwidths – at least in some cases.

To validate our fundamental assumption i.e., that in a CFS, using the LAN to move cached data around may decrease the latency and increase bandwidth both by an order of magnitude when compared to the established approach (using disk writes/reads to move data around), we decided to make small modifications to a well established, production-level CFS, in order to prove (or dispel) its feasibility. After carefully evaluating Oracle's OCFS (a pre-release at the time) and openGFS [openGFS] (seemingly phased out when GFS moved to "open source" status), both somehow documented, our choice was to use GFS. We ended up studying thousands of lines of GFS code (as "internals" documentation was/is not available) and decided to carry out tests through a mixture of real and simulated operations inside GFS kernel code. We have modified GFS' kernel module to follow one out of two different code paths when reading a file:

- **SAN path:** When a process in a node is reading a file, the regular GFS code path is followed: a shared read lock is placed on the file's ginode for the duration of the read; if another node wants to modify one or more file blocks, the node has to wait for the read to complete, get an exclusive lock over the ginode – which forces other nodes to release any shared locks they may hold and invalidate all cached data for that file – modify the blocks, flush them out to disk, and then, if necessary, release the lock to other nodes (e.g., they're waiting to resume their reads).
- **LAN path:** When directed to do so, by the simulation test, kernel code on the reader node follows another code path, where a) locks (requests and grants) are simulated by message exchange between the nodes, and b) the writer node supplies, from its own page cache, a copy of the modified page(s) to the readers.

To implement the LAN path we have built two kernel modules: a client module, which is called by the modified GFS code when a decision has been made to get data directly from another node, and forwards the request to the other node; and a server module, which handles a client request and ships the data back (for the proof-of-concept we opted for minor modifications to GFS, without any changes to the locking subsystem; but we nevertheless simulated the latency of lock messages through packet exchange).

Assessment of the proof-of-concept was carried out with a single-writer/multiple-readers parallel application where a single file is shared across nodes: after producing new data, the writer node signals reader(s) to consume it. Tests were run on the infrastructure depicted in Fig. 16.2. We used four IBM x335 dual-Xeon nodes, with clock speeds ranging from 2.6 (node 3) to 3.03 GHz (nodes 4, 5 and 6) and 4 GB of RAM per node. Two of the nodes were connected to the SAN through just one FC HBA, while others had two adapters per node; all HBAs were Qlogic QLA-2200F working at a speed of 1 Gbps. This heterogeneous

configuration allowed us to experiment with the use of dual vs. single paths to access the SAN and the array. The FC switch was an IBM 3534-F08 (a re-badged Brocade SilkWorm 2800) and the disk array was an IBM FAStT-200 model 3542-2RU with two storage processors (SP) and a total amount of 88MB of usable data cache per SP. For the experiments reported here, one independent 36 GB 7200 rpm FC disk has been "attached" to each controller, and both were visible on the SAN as LUNs 0 and 1.

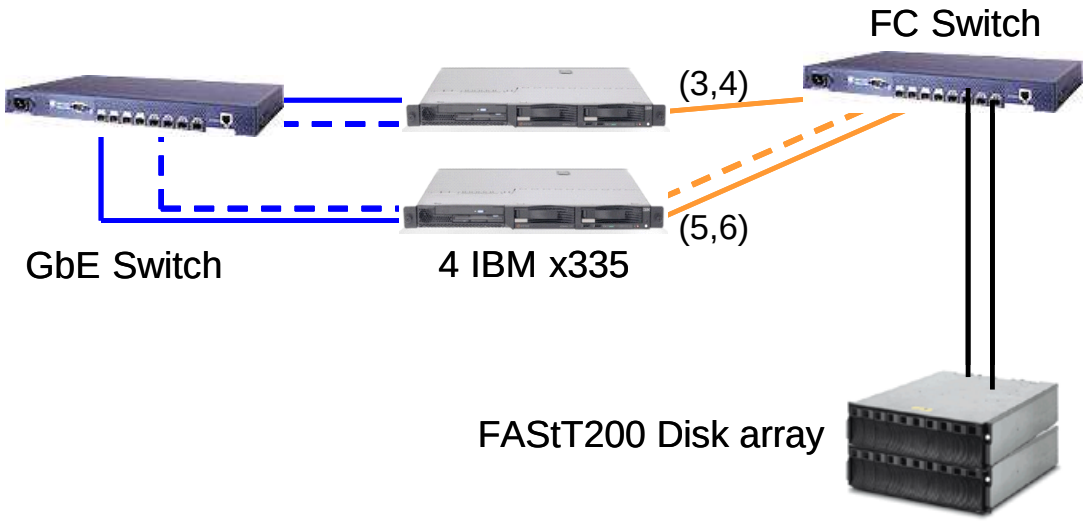


Figure 16.2 Proof-of-concept tests infrastructure

The nodes were running Red Hat EL3 (Kernel 2.4) and GFS (6.0). Each LUN was partitioned in half (18 GB) and both halves were joined together with GFS' clustered LVM version (CLVM) to form a 36 GB logical disk; on each node a different path was configured to access each LUN, which in fact doubles the bandwidth on the nodes which have two FC adapters; the 36 GB logical disk was then mounted by all cluster nodes.

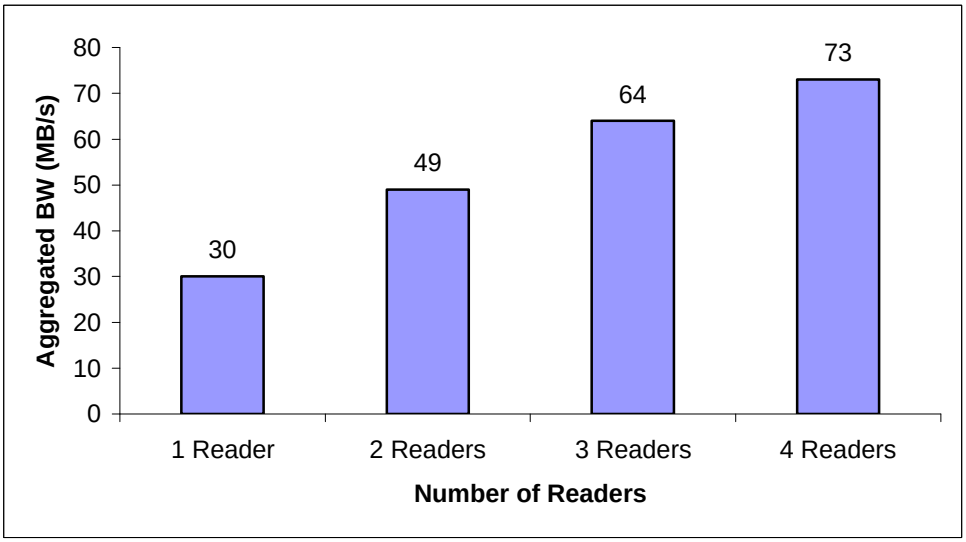


Figure 16.3 GFS' scalability: single file, multiple-readers with sequential access

Results were reported in [Lop+05] and we copy them here, for ease of reference. GFS' scalability with the configuration under test is shown in Fig. 16.3, where multiple readers were started in parallel to sequentially read a file; each reader was run in a different node, and, for each test, a new node was added. The test with four readers was able to reach the sustained rate quoted by IBM for the FAStT 200 disk array, which is 70 MB/s; the read buffer size used for the reads was 4 KB (bandwidth was computed dividing the total amount of data read by the time – taken at the slowest node – it took to read it).

Figure 16.4 below highlights the common problem of most SAN-based shared disk file systems: a single GFS writer is able do produce data (write a file) at 14 MB/s, but this rate decreases drastically if the file is shared with processes – readers, in this test – running on different nodes. Here, when a single writer shares the file with a single reader (1 W – 1 R test), the bandwidth is 0.16 MB/s for a 4 KB buffer; increasing the buffer size and/or the number of readers also increases bandwidth, and a maximum of 18 MB/s is reached for a 512 KB buffer when, following every write, three reads are fired in parallel (1 W – 3 R test); this behaviour shows that, for very large request sizes, write sharing files across nodes with GFS may provide an acceptable performance for some applications.

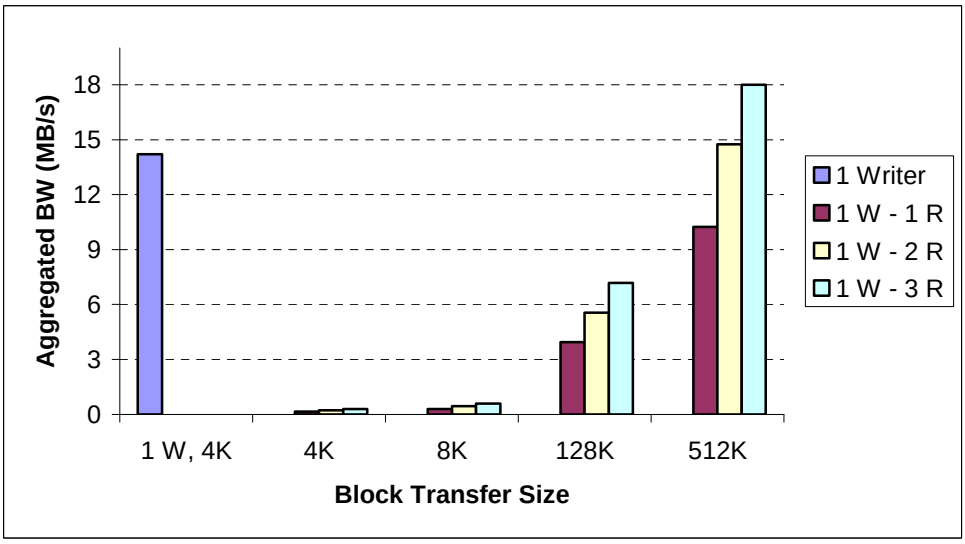


Figure 16.4 GFS' poor sharing: single file, one writer/multiple readers

Our assumption that using the LAN to move cached data around decreases the latency of data movement and increases bandwidth, both by an order of magnitude, is validated by results exhibited in Fig. 16.5: even for a single writer and a single reader, both using a 4 KB buffer, bandwidth jumped from GFS' 0.16 MB/s to 35 MB/s, a 200 times increase. The price to pay is an increase on the CPU usage; and this is a sharp increase, as we were using inexpensive Ethernet adapters, and those consume much more CPU to move the same amount of data than FC adapters (as reported in Part VIII, "Benchmarking pCFS").

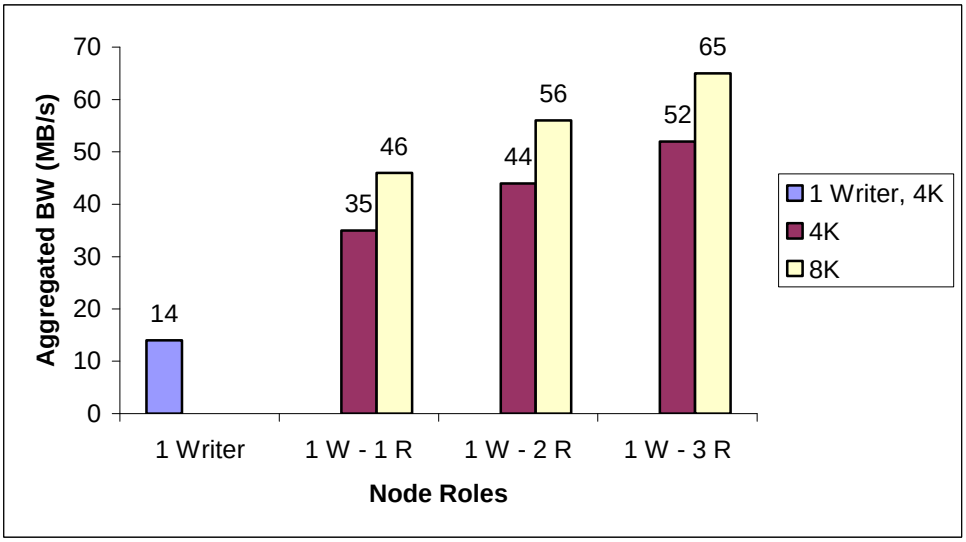


Figure 16.5 pCFS proof-of-concept: single file, one writer/multiple readers

16.5 Caching, fine-grain locking, and regions: the complete picture

Caches are only effective if they provide good hit ratios; in a distributed file system, a node's cache can only be effective if it satisfies both reads and writes and keeps them "away" from disks as long as possible; and that won't be possible if the cache is invalidated often, as it happens with GFS.

16.5.1 Caching and locking in GFS

But if shared disk cluster file systems such as GFS implement byte range locking, why do they invalidate all data cached in other nodes when a write occurs? The answer in GFS' case is that every write operation requires an exclusive lock against a ginode, and this triggers cache invalidations on other nodes; this is GFS' way to provide the so called "POSIX single node equivalent semantics" [Sch+02], and is a necessary step because GFS allows processes in a node to unreservedly access (for reading and/or writing) data that is concurrently being modified by processes in another node. GFS makes no effort to use, at the implementation level, the fine-grain locking mechanism it provides at the user level (byte range locks).

16.5.2 Regions, fine grain locking and caching in pCFS

Concurrent file sharing, as offered by pCFS requires processes to lock regions (with byte range locks) before accessing them, and this makes cache invalidation an infrequent event, as it occurs only when a writer region is removed (unlocked) and, even then, only frontier pages cached in other nodes get invalidated. This is possible because pCFS uses fine-grain locking also at the implementation level, and writes that do not result in major changes to the file's metadata (such as when its size grows, or "holes" in a sparse file are filled-in) can be carried out concurrently while the file's ginode is locked in shared mode (as described in [Lop+08])

and any access is checked to be valid within its region. However, operations that trigger major metadata changes still require us to use an exclusive lock (on the ginode and, if necessary, on other relevant metadata structures, e.g., resource groups, bitmaps, etc.).

16.5.3 Data forwarding vs. data shipping

pCFS, as we have seen, keeps coherent caches through updating, in writer nodes, and invalidation, in reader nodes. Updating is used when two writer nodes share a (frontier) page; there, one must forward all writes over that page to the other node – the so called page owner.

Data shipping is an extended form of data forwarding where "all" data – and not only some portion of a frontier page – is shipped to/from another node; it may be used, e.g., in situations where a) major metadata changes are quite frequent (such as in the producer-consumers sharing reported in [Lop+05], with a file that was initially empty), or b) regions are so small that there are many frontier pages[2] and, for performance reasons, it is better that a node is elected the file's data and metadata owner while all the other nodes ship data to/from it (in a NFS-like way, where the owner acts as an NFS server).

16.6 Programming with pCFS

Our proposal for pCFS requires that programming should not deviate from the use of the standard POSIX API; in fact, we merely propose a few additional option flags for the `open()` call, and a new way of looking at the semantics of existent locking primitives. Both were the result of some observations on currently available file systems, namely that there is no way of specifying the degree of sharing for a file at open time in the POSIX API (which in a distributed file system, results in all sorts of tricks being used to "implement" it), and that our region concept is closely related to the one of mandatory locks.

16.6.1 pCFS files and the extended `open()` options

We add three, mutually exclusive, options to the `open()` call:

- `O_CLSTXOPEN`, to request a cluster-wide exclusive open (i.e., if a process is able to open the file, any subsequent attempt by any other process to open the same file will fail);
- `O_NODEXOPEN`, to request a node exclusive open (i.e., if a process in a node was able to open the file, any subsequent attempt by any other process to open the file on the same node will fail);
- `O_CLSTSOPEN`, the flag for a cluster-wide (un-restricted) shared open.

The introduction of these flags was carried out without violating our premises, namely the "no VFS changes": all code was confined to the GFS layer (fortunately Linux does not check all flag combinations and allowed these new flags to "flow in"). And, furthermore, they have a very important side effect: they allow the user to choose between "pure GFS" or pCFS

[2] This will happen in access patterns with a per-record lock/access/unlock sequence where a small record is locked, accessed and then immediately unlocked.

behaviour just by omitting or including these flags. The simplicity of this process is also highly beneficial to the debugging and benchmarking tasks.

> Definition 16.1 A pCFS file is a file that lives in a GFS filesystem and is opened with one of the following flags: `O_CLSTXOPEN`, `O_NODEXOPEN`, or `O_CLSTSOPEN`.

16.6.2 pCFS regions

After the open (which, for a pCFS regular file must include one of the above flags), the user may declare a region over which accesses will be made, by specifying its start and end byte offsets, and how the region will be used (for reading or writing – i.e., in shared or exclusive mode). Region declaration is performed with `fcntl()`, and region modes are expressed and enforced at declaration time, but can later be changed by choosing a mode which is more or less restrictive than the current one (e.g., going from exclusive to shared, or vice-versa). Regions may overlap if their modes are compatible, i.e., both are shared.

pCFS regions behave as a sort of mandatory locks: every pCFS file access (read or write) is checked against the region boundaries; if an access would violate them, an error is returned and the operation is not performed. Regions also guarantee consistency among sharers on the file: readers striving to access a region occupied by a writer may either try to get in (and keep retrying if they can't) or queue up waiting for the writer to leave; when a process is granted access, it is guaranteed that it will see the latest version of the data.

16.7 The pCFS prototype

16.7.1 pCFS conceptual architecture

The conceptual architecture we propose for pCFS does not differ from other typical SAN-based CFS architectures, as Fig. 16.6 shows.

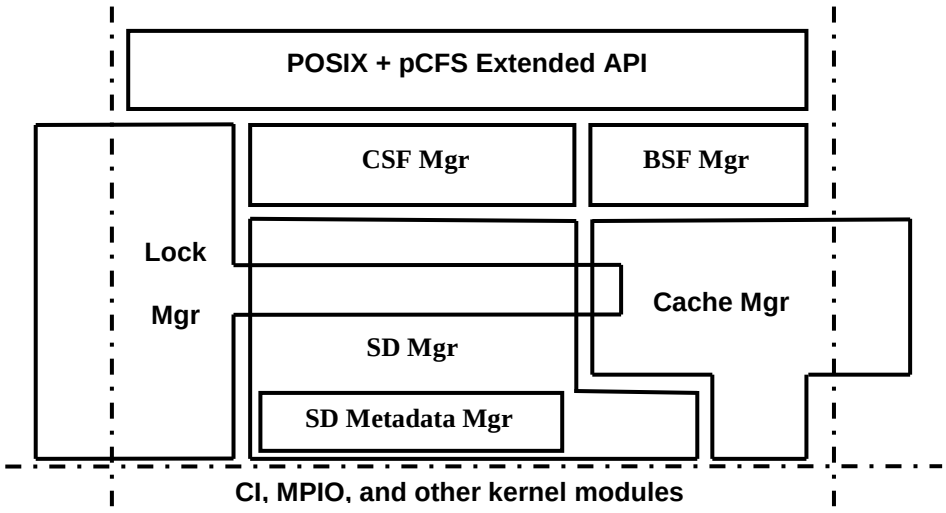


Figure 16.6 pCFS high-level module architecture

The Cluster Infrastructure (CI)[3] is the core building block for a true cluster service (setting it apart from "cluster setups" which are no more than just a collection of nodes), and usually encompasses:

- A membership service that keeps track of which nodes belong to the cluster, taking care of node admission, leave, and eviction (e.g., when a node fails to comply with some basic policy requirement, such as failing to answer a predefined number of heartbeats);
- An inter-node communication service which provides reliable communication among enrolled cluster nodes, gracefully handling dynamic reconfiguration events, such as nodes entering or leaving the cluster (may provide an API for message broadcast and/or multicast support);
- A publish/subscribe database that allows "producers" to register themselves and advertise services or resources they provide, and "consumers" to specify what services and/or resources they need to operate;
- And a failover/failback database that allows the administrator to specify "logical resources" (such services or disks) and rules for the transference/restart of those resources in case of node failure (failover) and resume (failback).

Multi-Path I/O (MPIO), as described here, is an extension for what once was a device--specific concept: that one can aggregate multiple I/O devices of the same class (e.g., LAN interfaces or disk devices) under a common umbrella and use them together to provide higher bandwidth and/or availability. It has been widely used for LAN interfaces, where it is commonly referred to as bonding, link aggregation or trunking. Less known, as it requires either multi-ported disks or a SAN infrastructure (see section 5), is disk-based MPIO; in Linux, disk-based MPIO has long been provided by manufacturers for their own FC drivers; however, recently, it has received enough attention to be regarded as a separate kernel upper--layer module where adapter-specific driver modules should plug-in.

The Shared Disk Manager (SDM) is the module that accesses disk data blocks (requesting them to flow through the appropriate lower-level I/O stack paths) and guarantees coherency among copies of the same shared data blocks when they are kept in different nodes. The level of coherency can vary, as dictated by the usage policy of the layers above the SDM; one can imagine a strict policy where all copies of the same block must be in-sync all the time, or a less restrictive one which, for example, allows data that has been cached but cannot be accessed by a node (e.g., only some part of a block which has not been modified elsewhere, can be accessed) to stay un-coherent for some time; furthermore, it is not obvious that the block is the unit of choice for coherence (see CM, below). The Shared Disk Metadata Manager has detailed knowledge of both on-disk and in-core layouts for metadata structures, and is called when it is necessary to transfer them from/to disk; it also takes care, in the same vein of SDM, of keeping them coherent across nodes.

[3] CI software has been developed to support commercial-grade clusters (e.g., Windows Cluster, Compaq TruCluster, Red Hat Enterprise Linux)

The Cache Manager (CM) is the module that handles data caching; one can devise either a completely separated cache, i.e., one that does not integrate with the operating system page cache (see 15.3.1.1, GPFS implementation on Linux) or the approach followed by most Linux file systems, a tightly integrated one. Here, the issue of coherency must be again tackled; as the element of caching is a page, it is "natural" to promote cache coherency at the page level, a decision which may seem to contradict the block approach previously suggested. In fact, we could have both in place: a page-grained coherency for I/O requests that flow through the page cache, and a block-grained coherency for requests that do not use the page cache.

In a distributed file system the Lock Manager (LM) is of paramount importance, because it is commonly used to implement *both* cache coherency *and* user-level locking primitives (when available). The LM supplies "global locks" that are used, at a cluster-wide level, to lock target objects; e.g., when a request to place a shared global lock over some data structure (residing in the node) is issued, the LM of the requestor node interacts either with a centralised lock server (in a client/server implementation) or with the LM in other nodes (in a truly distributed implementation) to "get hold" of the lock; if successful, i.e., the lock was granted to the requester, the lock is now held at the node (where it protects some data structure). The locked object may be a purely internal file system object, such as a superblock or inode, or the internal representation for a user-level object, such as a file lock.

Finally, the character-special and block-special file managers (CSF and BSF) are used here as mere illustrations of two abstract concepts that represent the two standard UNIX interfaces to access devices: character and block-oriented.

16.7.2 Objectives of the pCFS prototype

Global objectives of the pCFS prototype are:

- New concepts brought in should require a minimum of change in user programming habits, i.e., they should resort to concepts already familiar to programmers, which should not be forced to use new APIs (although new parameter options for commonly used file calls are acceptable).
- Evaluation should be carried out over regular file access, i.e., we do not intend (for now) to improve the speed of memory mapped I/O or metadata operations (e.g., increase the speed of file creation, deletion, directory listings, etc.).

Specific objectives of the pCFS prototype are:

- True file sharing – that is, sharing a file among processes running in distinct nodes for simultaneously reading and writing – operations should have performances which are close to those exhibited in non-sharing situations. Of paramount importance is the situation that arises in typical HPC applications: files often need to be shared across nodes, but each node accesses a file region which does not overlap, at the byte-level, with regions accessed on other nodes.
- Proposed contributions should either be implemented (completely or partially), or else proved to be possible to implement.

16.7.3 Methodology for the pCFS prototype implementation

Developing from scratch a new shared disk file system with the proposed features would be a huge task, unattainable in the realm of this work; conversely, a feature-light implementation would preclude a fair head-to-head comparison against other file systems. Therefore, we decided to build on the work carried out for the proof-of-concept, keeping GFS as the basis for our prototype.

Before we can plunge into Part VII, “pCFS implementation”, we must have a more detailed look at VFS and GFS; the reason is quite simple: GFS is strongly coupled with VFS. Therefore, a description on how VFS provides the fundamental abstractions for different file systems (and how they plug themselves into VFS) as well as an overview on GFS’ architecture (followed by an in-depth look at its global locking mechanisms) is a pre-requisite to understand both the prototype and design decisions that were taken along the path.

Part VI:

Inside the Kernel: VFS and GFS

This Part constitutes a prerequisite to understand the pCFS implementation: in the first section, we discuss the architecture of the Linux VFS and how it is used to integrate specific file systems; in the second section we present an overview of the internals of GFS; and, finally, in the last section we describe, with some detail, how GFS implements locking and uses it to promote clusterwide coherency among nodes.

17 VFS internals

In this section, a very short description of the fundamental abstractions provided by VFS is presented, with an eye on what happens on two important operations: filesystem mounting and opening a file. VFS internals are quite adequately covered in several books, including [Bov+05] and [Rod+05].

17.1 The Linux Virtual File System

The Linux Virtual File System is a layer that captures the commonality between different file system types; it's model is, unsurprisingly, closely related to the UNIX file system model, where a file system instance is described by a superblock, a file by an inode, and a directory is a (special) file that contains names of files and other directories, together with inode numbers. The major VFS objects represented in Fig. 17.1 are:

superblock

Stores information about a mounted filesystem; when a filesystem is mounted, this object is created and gets populated with data retrieved from a filesystem "control block" stored on disk[1].

inode

Stores information about a particular file; gets populated with data retrieved from the file's metadata, stored on disk. Each inode[2] object is identified by an inode number that uniquely identifies the file within the filesystem instance.

file

Stores information about the interaction between an open file and processes accessing it; this is where, for example, the file pointer abstraction is implemented.

dentry

Is the representation of a directory entry in the VFS world; stores information about a file by linking the filename to the file's inode.

For some kernel data structures it necessary to check, very quickly, whether a particular instance exists or not, in memory; this is why slab caches for those structures are further organised into hash tables: inode and dentry instances, for example, are stored in hash-searchable caches. Also, the page cache is used to store file page descriptors – a data structure that points to the "real" file page (notice that previously we have referred to the page cache as if it had the "real data" in it, not pointers…).

[1] Here we are interested in disk based file systems, and our examples assume just those.

[2] Notice that the Linux VFS uses exactly the same terminology – inode – for the VFS and on-disk data structure while, e.g., Sun uses vnode for the VFS structure and inode for the on-disk structure.

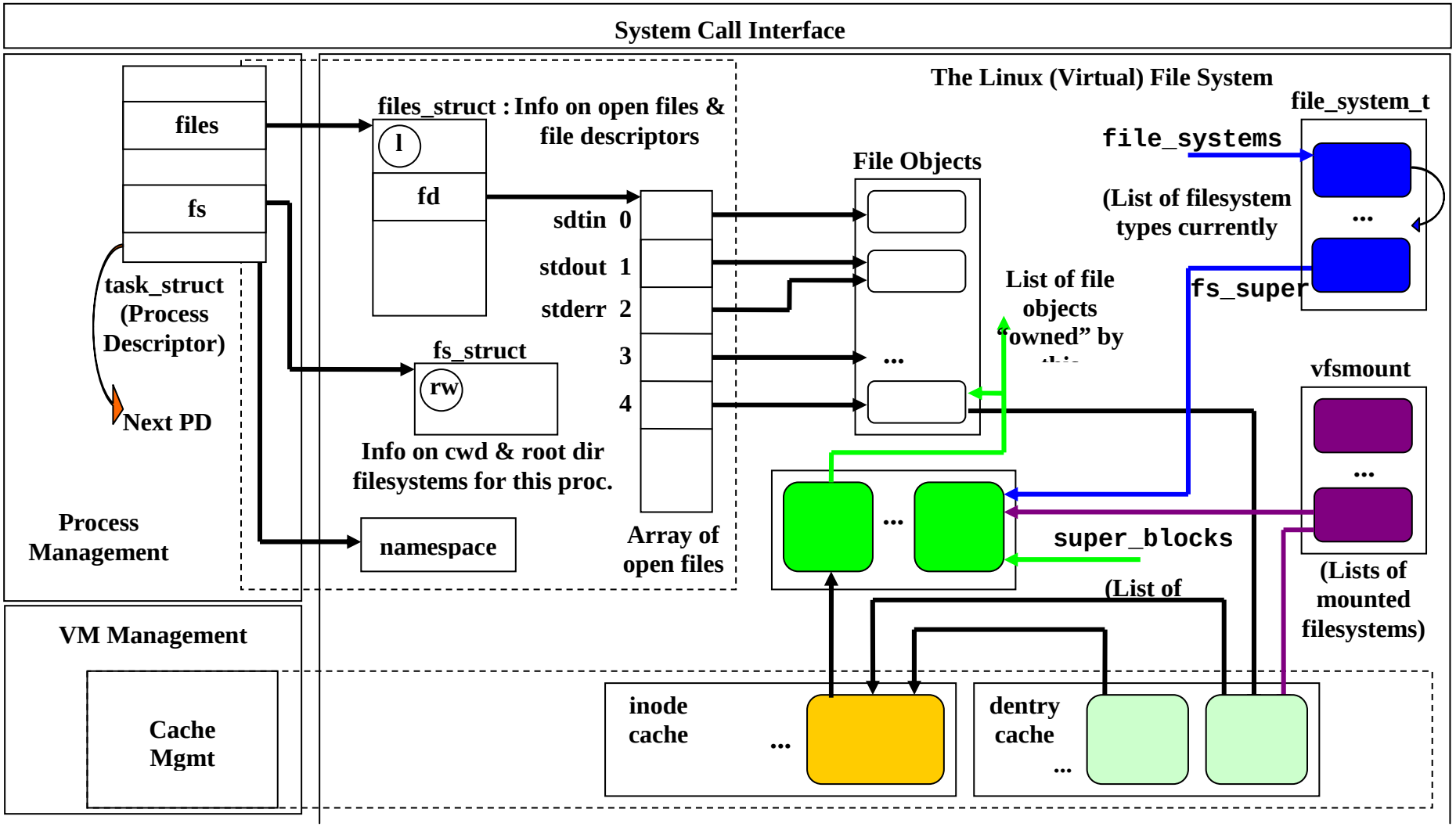


Figure 17.1 Architecture of the Linux Virtual File System layer (and relationships with other layers)

17.2 A closer look at the Linux Virtual File System

17.2.1 Begin in the beginning: file system mounting

The first step for plugging a file system into the VFS happens at the `mount`; we will now briefly describe a subset of the most important VFS objects accessed when a `mount` is performed, as well as their relationships; we also describe how the liaison between the VFS and the specific file system is established, i.e., how VFS generic code ends up calling the file system's specific code.

17.2.1.1 Providing the kernel with the file system implementation module

Whether a module that implements a specific file system is built-in at kernel compile time or dynamically inserted at runtime, it must be register itself with the Linux kernel, an operation that causes a new object (of `struct file_system_type`) to be created and appended to the single-linked list whose head is stored in the `file_systems` global variable (see Fig. 17.2); the object must have been previously initialized with, among other information, the filesystem name (e.g., the object describing an available ext2 file system implementation module would have an "ext2" string on the name field), a pointer to the function that will be invoked on the mount operation to read the filesystem superblock (the `read_super` field), and a pointer to the implementation module (the `owner` field).

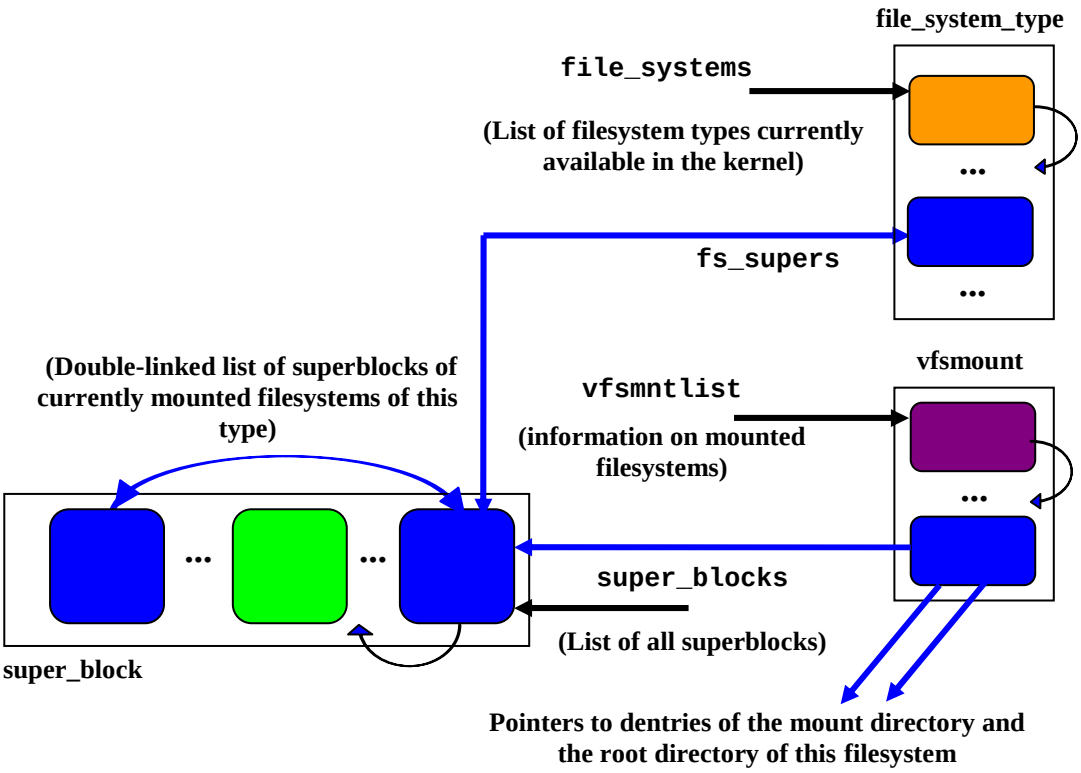


Figure 17.2 VFS objects involved in file system mounting

17.2.1.2 Mounting a filesystem

The simplest (and even so, oversimplified) execution path on a mount operation is:

a) an object (of type `struct super_block`) is created and initialized (for example, a field pointing to the desired device gets filled), and then gets populated with data retrieved from disk when the function `read_super()` is executed (what happens is that the `read_super` field of the appropriate `file_system_type` object is set to point to a filesystem specific function supplied with the module; that function usually resorts either to the "buffer cache" `bread()` or `breada()` to do the actual read);

b) on a successful return, make the superblock object operations pointer `s_op` (see 17.2.1.3 below) point to the `struct super_block_operations` provided by the module implementing the file system – these are the functions implementing operations such as "read inode", "write inode", "write the superblock", etc., that are specific to that file system type (and are called via pointer dereferencing as usual, e.g., `sb->s_op->read_inode()`);

c) the superblock object is appended to the double-linked list whose head is the `fs_supers` field of the `file_system_type` object, which keeps track of all the mounted superblocks of the same file system type;

d) an inode and a dentry are allocated for the "root directory" of the file system, and these objects are then linked to the appropriate VFS data structures;

e) a `struct vfsmount` object is created and filled in, then linked to the appropriate lists and other VFS objects; it stores information about the mount point, mount flags, and relationships between the file system being mounted and other, already mounted, file systems.

In short, the mount operation is the one that, for the specific instance being mounted, "glues" the file system implementation module to the VFS layer, in such a way that the same set of user-level file operations can be used with every file, independently of the specifics of the file system where the file "lives".

17.2.1.3 The superblock object

A shortened listing of the superblock data structure (see `include/linux/fs.h`) is:

```
struct super_block {
  struct list_head         s_list;
  kdev_t                   s_dev;
  unsigned long            s_blocksize;
  …
  unsigned char            s_dirt;
  …
  struct file_system_type  *s_type;
  struct super_operations  *s_op;
  …
  struct dentry            *s_root;
  …
  struct list_head         s_dirty;          /* dirty inodes */
  struct list_head         s_locked_inodes; /* inodes being synced */
  struct list_head         s_files;
  struct block_device      *s_bdev;
  struct list_head         s_instances;
  …
```

```
  union {
    …
    struct ext2_sb_info           ext2_sb;
    …
    struct msdos_sb_info          msdos_sb;
    struct isofs_sb_info          isofs_sb;
    …
    struct udf_sb_info            udf_sb;
    …
    void                          *generic_sbp;
  } u;
  …
}
```

We can easily identify a fixed part, containing several items and corresponding to the VFS superblock object, and a variable, possibly empty union that is used to extend the "base" VFS object with a file system-specific data structure; that structure is, from the VFS point of view, opaque, and should be accessed only by the file system-specific code, not by VFS code. In the union we can find the "add-on" structure pointers for disk-based file systems such as ext2 or MS-DOS, for CD-ROM devices with ISO or UDF formats, as well as a "catch all" for file systems not included in the standard kernel release, in the form of an opaque pointer.

For some of the above mentioned fields, we now briefly describe their usage:

- The `s_list` field is used to attach the superblock to the global linked list of all the superblocks, while `s_instances` is used to attach the superblock to a list of all the other superblocks of the same file system type; `s_dev` and `s_bdev` identify the device where the file system (and thus the superblock) lives, while `s_blocksize` indicates the size (in bytes) of a block on that particular device;
- The `s_files` field is used to build a list of all file objects "in-use" that refer to files living in this file system; this greatly simplifies work needed to get a list of the files currently opened on a particular file system, because all one needs to do is walk through this list;
- If the in-core superblock image has been modified, `s_dirt` is set; when some in-core inode has been modified, it is appended to the `s_dirty` list; then, it becomes easy to update the on-disk images for all inodes, as all we'll have to do is go through the `s_dirty` list updating each inode at a time. All inodes (that belong to this superblock file system) involved in an I/O operation at a particular time are collected in the `s_locked_inodes` list.

Now let's look at the superblock object methods, i.e., the superblock operations. These are "bound" via the `s_op` field. How does one define these operations? We must always remember that the VFS layer is immutable, i.e., it's "structure" and "code" are already written … but with a lot of generic VFS code, as well as pointers to prototype functions not yet defined (`NULL` pointers) – and those are the ones that a file system developer which wants to add a new file system to the kernel must code in the new FS module; to do it properly, he/she must adhere to the VFS predefined superblock (function) operations (also in `include/linux/fs.h`):

```
struct super_operations {
  struct inode *(*alloc_inode) (struct super_block *sb);
  void (*destroy_inode)        (struct inode *);
  void (*read_inode)           (struct inode *);

  void (*dirty_inode)          (struct inode *);
  void (*write_inode)          (struct inode *, int);
  …
  void (*delete_inode)         (struct inode *);
  …
  void (*put_super)            (struct super_block *);
  void (*write_super)          (struct super_block *);
  …
  int  (*statfs)               (struct super_block*, struct
                                statfs *);
  …
  void (*clear_inode)          (struct inode *);
  …
}
```

The developer must implement those functions that are needed to map the new file system into the VFS model; for example, ext2's developers have defined a function to read information from a disk data structure and fill in the VFS `inode` structure, and they called this function `ext2_read_inode()`; then, they went on to define functions to write VFS inodes to their on-disk inode images, to write the VFS superblock to its on-disk image, etc. Then, they defined a structure in the module that implements the "real" (ext2) file system, and "assigned" all (functions) operations to the correct fields of the structure, as in the following code fragment from the `ext2` implementation module (`fs/ext2/super.c`):

```
static struct super_operations ext2_sops = {
  read_inode:     ext2_read_inode,
  write_inode:    ext2_write_inode,
  put_inode:      ext2_put_inode,
  delete_inode:   ext2_delete_inode,
  put_super:      ext2_put_super,
  write_super:    ext2_write_super,
  statfs:         ext2_statfs,
  remount_fs:     ext2_remount,
}
```

Notice that not all functions have to be implemented: one may leave some as `NULL` pointers, while others may point to generic VFS (pre-defined) functions.

Finally, when mounting an ext2 file system one must link the superblock's `s_op` field to the above structure, thus "binding" the VFS abstract operations to the "real" functions implemented in the ext2's module. The last step of this "binding" is performed in the `ext2_read_super()` function, (also in `fs/ext2/super.c`) where we find,

```
struct super_block *sb

…
sb->s_op = &ext2_sops;

…
```

So, when a VFS superblock is being filled by the "abstract" `read_super()` function (bound to the "real" `ext2_read_super()` function which reads the on-disk superblock), at some point of the code's execution the VFS superblock operations pointer is set to point to the "real" functions living in the `ext2` module.

To conclude, the VFS layer has several major "abstract objects" along with their "abstract methods"; some of these methods are already pre-bound to VFS code, while others must be implemented as functions within the filesystem specific module. The explanation of how this is done is a blueprint that can be applied to many other similar VFS objects.

17.2.2 Opening a file

From the set of the most frequently used file operations, open, close, read and write, we will look only at one, the `open()` system call: on one hand, it makes an interesting study because in bridges together two kernel layers, namely Process Management and VFS; on the other hand, studying each call is both tedious and not strictly necessary; the interested reader can refer to [Bov+05, Rod+05].

An `open()` system call is invoked with the following parameters: the pathname of the file to be opened, option flags and access mode flags, and a permission bit mask mode if the file is to be created. If the system call succeeds, it returns a file descriptor – that is, the index assigned to a new entry in the `current->files->fd` array of pointers which will point to the file object,

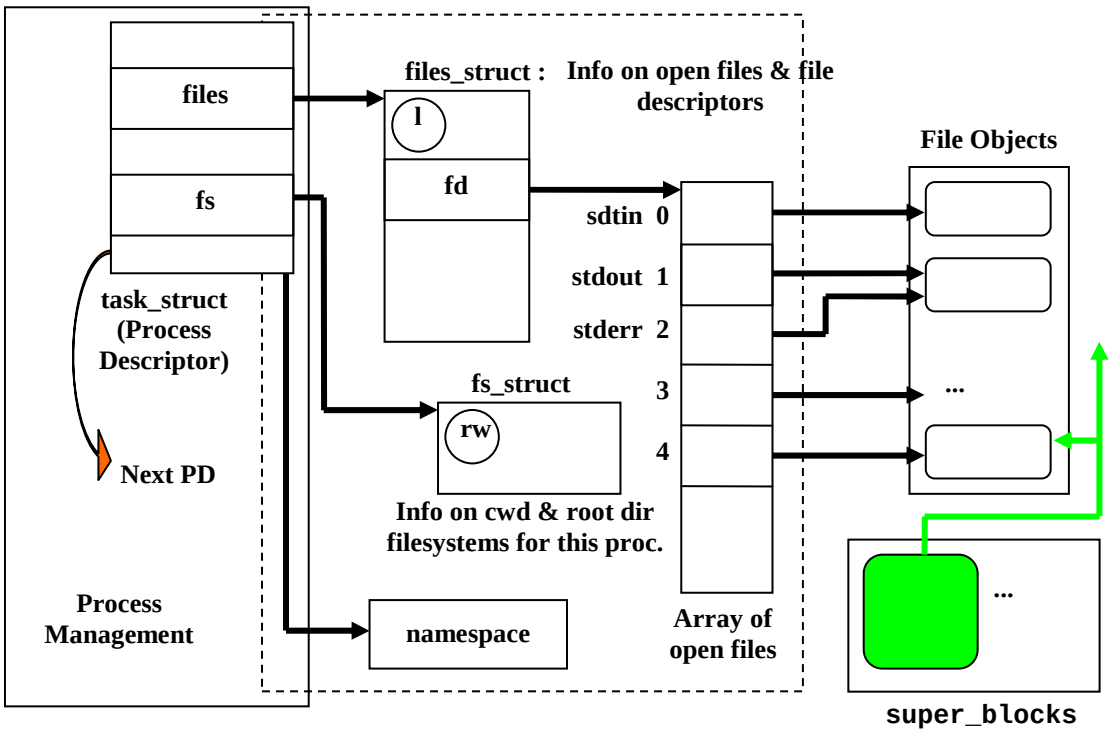


Figure 17.3 Some VFS objects involved in the `open()` system call

The oversimplified (e.g., we will leave out file creation performed through the open call) execution path on an `open()` is:

a) `getname()` is invoked to extract the file pathname from the process address space;
b) `get_unused_fd()` is invoked to find an empty slot in the `current->files->fd` array. The corresponding index (the new file descriptor) is stored in the `fd` local variable;
c) next, `filp_open()` is invoked, with the following parameters: pathname, access mode flags, and permission bit mask. This function, in turn, executes the following steps:
   1- `open_namei()` is invoked to perform a lookup operation, with the following parameters: pathname, access mode flags (encoded in a different way), and a pointer to a local `struct nameidata`; if successful, it returns in its fields `dentry` and `mnt` the addresses of the dentry object and mounted file system objects associated with the successfully looked up file.
   2- `dentry_open()` is invoked, with the access mode flags, and the pointers to the dentry and mounted filesystem objects returned by the lookup operation as parameters. This function:
      a) allocates a new file object, and initializes the fields `f_flags` and `f_mode` according to the access mode flags passed to the `open()` call;
      b) initializes the `f_fentry` and `f_vfsmnt` fields according to the addresses of the dentry and the mounted filesystem objects passed as parameters to the `dentry_open()` call.
      c) sets the `f_op` field to the contents of the `i_fop` field of the corresponding `inode` object; this sets up all the methods for future file operations.
      d) inserts the file object into the list of opened files pointed to by the `s_files` field of the filesystem superblock;
      e) if the `O_DIRECT` flag is set, pre-allocates a direct access buffer;
      f) if the open method of the file operations is defined, invokes it.
d) Sets `current->files->fd[fd]` to the address of the file object returned by `dentry_open()`;
e) Returns `fd`.

Two important things to note are: the file access operations (function pointers) are defined in the file object and are copied from the file's inode object operations, as 2-c) shows; and, a "file system-specific open", if set (i.e., not `NULL`), will get executed almost at the end of the code path, in 2-f).

17.2.3 Closing remarks and GFS preview

VFS code for file operations, as seen above for the `open()` function, allows for calling specific code provided by the target file system; we will shortly see that, when opening a GFS file, `gfs_open()` will be called at the location 2-f), thus bridging the VFS and GFS worlds, and opening the door for file system specific actions, when necessary – for example, in ext2 there are no specific actions to perform, so the function pointer in 2-f) is `NULL`.

18 GFS internals: an introduction

18.1 GFS architecture

Figure 18.1 is an overview of the GFS architecture and its locking hierarchy (adapted from [openGFS]); the left side shows lock module loading/unloading (registering/un-registering with the lock harness) and lock protocol mounting, while the right side shows all other operations. GFS is tightly integrated into VFS, with some GFS objects linked to VFS ones, e.g., VFS superblock's `generic_sbp` points to the GFS in-core superblock image.

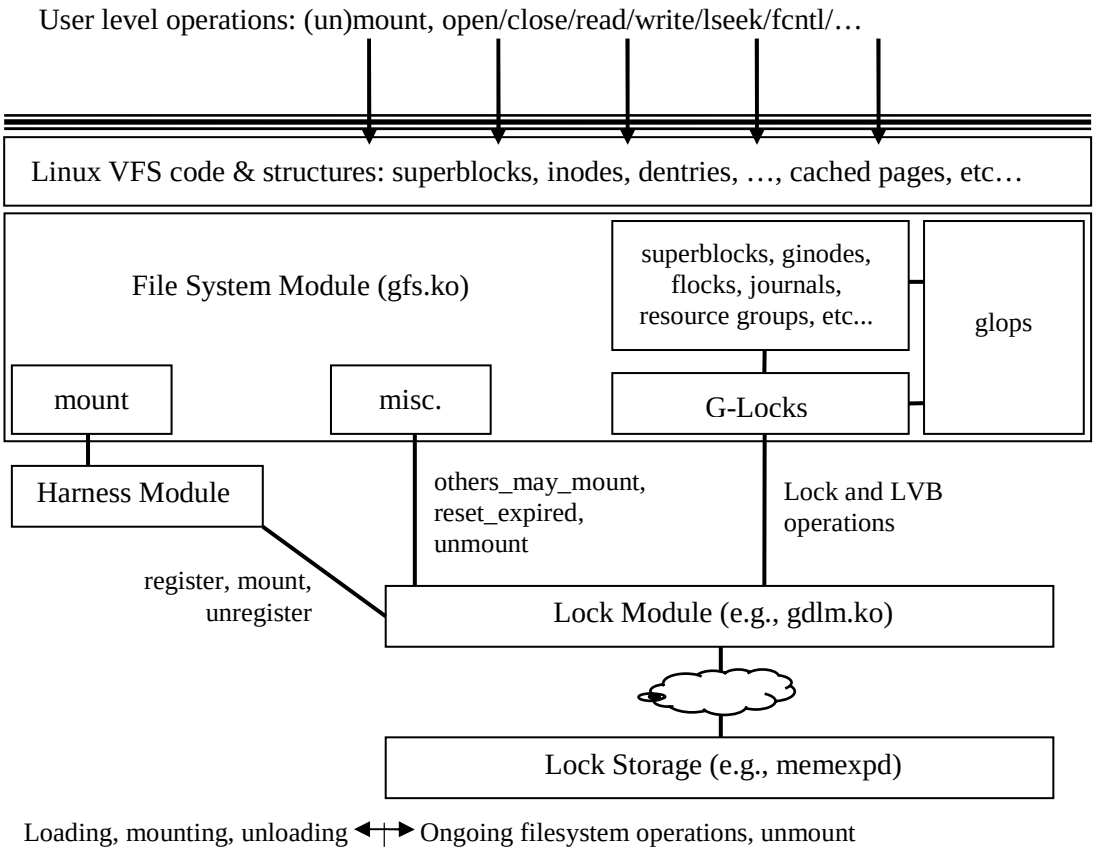


Figure 18.1 GFS fundamental software modules and layers

GFS supports three inter-node lock protocol implementations: the client-server "Grand Unified Lock Manager" (`GULM`)[1]; the distributed "GFS Distributed Lock Manager" (`GDLM`); and the "no lock" (`NO_LOCK`) implementation which allows GFS to be used as a single node local file system. The lock harness serves two simple purposes:

- Maintaining the list of the implementations (e.g., `GULM`, `GDLM`) currently available for filesystem mounting and,
- "Connecting" a selected LM module to a filesystem, at mount time.

[1] Now seemingly discontinued.

As an example, assume we have two distinct locking protocols available: `GULM`, which will allow us to use shared disks across nodes, and `NO_LOCK`, for disks we want to mount locally at a single node, without incurring in inter-node locking overheads. When a locking protocol module is brought into the kernel (issuing, e.g., `modprobe lock_gulm` or `modprobe no_lock`) it registers itself with the harness module; then, when the user performs a GFS filesystem mount, e.g., with `mount –t gfs … -lockproto=no_lock`, the harness module is accessed to extract the appropriate pointers to the lock protocol functions, and all locking operations within GFS will "trigger" the chosen module's locking functions.

The locking modules and lock storage facility take care of:

- Managing and storing inter-node locks and lock value blocks (LVBs, see further down).
- Handling lock expiration (lock request timeout) and deadlock detection.
- Heartbeat functionality[2] (are other nodes alive and healthy?).
- Fencing nodes[2], recovering locks, and triggering journal replay in case of a node failure).

The G-Lock software layer is a part of the GFS file system code. It handles:

- Caching and coordinating locks and LVBs among processes on **this** node.
- Communication with the locking backend (lock module) for inter-node locks.
- Executing glops when appropriate (see below).
- Journal replay in case of a node failure.

The Global Lock (G-Lock, a.k.a. glock) is a fundamental GFS concept, one which will be studied in more detail in the next section; for the moment it is sufficient to say that it is an abstract cluster-wide visible "object" that may be used to support synchronized access to protected resources (e.g., GFS inodes) shared among nodes; access requests may originate from local, intra-node, or global, inter-node, processes; G-Locks also support serialisation of intra-node accesses for correct GFS operation on SMP architectures. The G-Lock operations layer is also a part of GFS file system code, implementing the file system-specific, architecture-specific, and protected-item-specific operations that must occur right after locking or just before unlocking, such as:

- Reading items from disk, or from another node[3] via a LVB, after locking a lock,
- Flushing items to disk, or to other nodes via a LVB, before unlocking a lock,
- Invalidating kernel buffers, once flushed to disk, so that a node can't keep on using them while another node is changing their contents.

Each glock has a type-dependent vector of operations (glops) structure attached to it; this is the key to porting the locking system to other environments, and/or creating different types of glocks, and defining their associated behaviour.

---

[2] In recent revisions, some of these tasks have been, or are in the process of being moved to a new module that implements generic Cluster Infrastructure functionalities.

[3] The only items currently "moved around" from node to node in LVBs are resource group bitmaps.

Finally, an LVB is an opaque data type that is used to carry information across cluster nodes, and is a performance enhancement path for maintaining frequently updated data structures coherent across nodes; for example, it is used to maintain resource group bitmaps in sync across nodes, e.g., a node that updated some bitmap does not need to flush it to disk first so that it can be re-read from disk in other nodes. An LVB is attached to the lock that protects the GFS object it holds, and has its own set of operations; currently, LVB size is fixed, at 32 bytes.

18.2 Lock harness

When a lock manager module is inserted into the kernel, as part of the module initialization it registers itself with the lock harness via the `lm_register_proto()` call; this adds the protocol implemented by that module to the list of available locking protocols in the cluster, ultimately allowing GFS to access to the set of operations it provides.

At the top layer, the lock harness layer offers a set of services to aid in file system mounting (and un-mounting) by performing the remaining part of the LM initialization (or removal); these are: `lm_mount()`, `lm_unmount()`, and `lm_withdraw()`.

To illustrate the use of the functions listed above, let's look at a GFS file system mount operation: at mount time, when asked – via options string, as in `lockproto=gulm` – to use some available lock protocol, the lock harness layer will plug the module's supplied set of operations into the mounted file system GFS in-core "superblock" structure: first, by executing `lm_mount()`[4], which fills in some information within the `sd_lockstruct` structure; then calling the "mount" operation provided by that protocol's implementation module, e.g., `gulm_mount()` for `GULM`; this will, in turn, fill in other information, particularly the `ls_ops` field with the appropriate vector of operations.

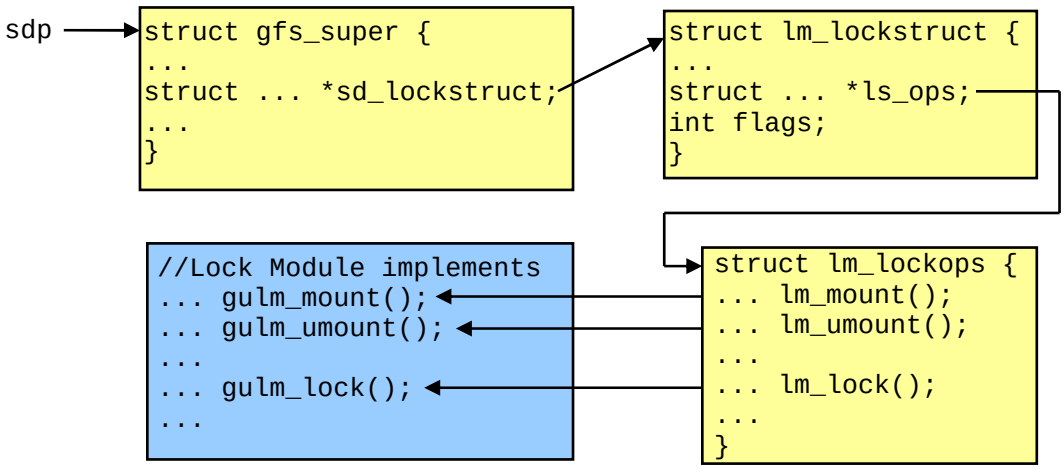


Figure 18.2 Plugging a lock module (in blue) into GFS structures (in yellow)

[4] The calling sequence is: `mount->…->gfs_get_sb->…->gfs_lm_mount->lm_mount()`

Thus, to request a `lm_lock` operation on a lock living in a GFS file system "described" by a superblock pointed to by `sdp`, one may write,

```
sdp -> sd_lockstruct -> ls_ops -> lm_lock(…)
```

which would execute the appropriate lock routine within the protocol used to handle locks in the file system; e.g., the above sequence when applied to the example exhibited in Fig. 18.2 would end up calling the `gulm_lock()` function.

18.3 Lock module

The diagram in Fig. 18.3 below is an overview on how GFS uses lock modules (again, adapted from [openGFS]); it covers calls to the module from all parts of the file system and harness code (where some calls have no functionality if the `no_lock` module is used).

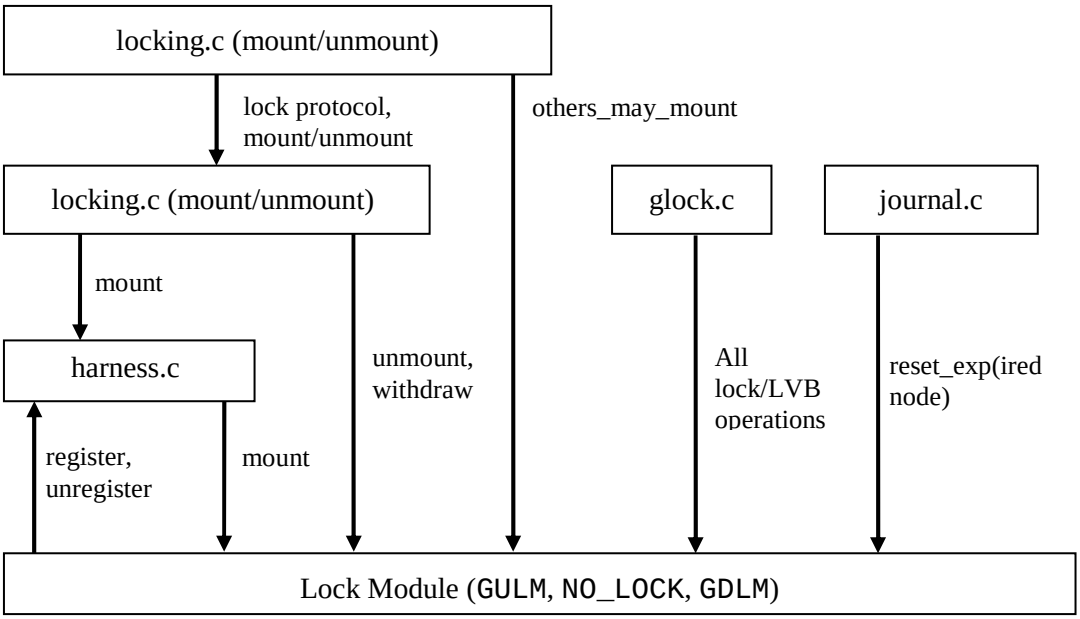


Figure 18.3 GFS usage of Lock module operations

All GFS implementations of a lock protocol must adhere to the same API; the interface[5] is very simple, and defines:

- a lock type, coded as `LM_TYPE_{…, INODE, RGRP, META, …}`,
- a lock state, coded as `LM_ST_{UNLOCKED, EXCLUSIVE, DEFERRED, SHARED}`,
- various lock operations, such as:
  - `lm_get_lock()`, `lm_put_lock()`
  - `lm_lock()`, `lm_unlock()`
  - `lm_hold_lvb()`, `lm_unhold_lvb()`, `lm_sync_lvb()`
- various flags to control the behaviour of lock calls:
  - `LM_FLAG_{TRY, TRY_1CB, NOEXP, ANY, PRIORITY}`
- Other flags to indicate return conditions from `lm_lock()`, coded as `LM_OUT_{…}`

[5] See `harness/lm_interface.h`

- A set of interface operations with Lock Harness and for (un)mount support, such as:
  - `lm_register_proto()`, `lm_unregister_proto()`
  - `lm_mount()`, `lm_unmount()`, `lm_withdraw()`

In brief, a GFS implementation of a lock protocol must be able to create a cluster-wide visible (unlocked) lock, identified by its lock number and lock type; lock it into some allowed state (and also be able to unlock it); support a Lock Value Block (LVB) data type and related set of operations; and, furthermore, support both synchronous (blocking) and asynchronous (non-blocking) operations.

The lock operations are the most important thing on this overview; a brief summary of locking and LVB-related functions called by G-Lock layer follows:

`lm_get_lock` -	find an existing, or allocate and initialize a new `lm_lock_t` (lock module per-lock private data) structure on this node. Does **not** access lock storage, or make lock known to other nodes.
`lm_put_lock` -	de-allocate an `lm_lock_t` structure on this node, release usage of (perhaps de-allocate) an attached LVB (by calling `lm_unhold_lvb`). Accesses lock storage only if LVB action is required.
`lm_lock` -	lock an inter-node lock (allocate a buffer in lock storage, if needed)
`lm_unlock` -	unlock an inter-node lock (de-allocate the buffer in lock storage, if possible)
`lm_cancel` -	cancel a request on an inter-node lock (ends retry loop)
`lm_hold_lvb` -	find an existing, or allocate and initialize a new Lock Value Block (LVB)
`lm_unhold_lvb` -	release usage of (perhaps de-allocate) an LVB
`lm_sync_lvb` -	synchronize LVB (make its contents visible to other nodes)

These “abstract” operations are implemented differently by each distinct lock module, e.g., `GULM` implements them in according to the client/server `GULM` protocol, while the `GDLM` implementation is quite different, and truly distributed with no single point of failure; function names are chosen in order to show up the module’s name, e.g., the `GULM` routines (see `gulm/gulm.h`) are called `gulm_lock`, `gulm_unlock`, etc.

We end this overview of the Lock Module by stressing out that lock creation, as referred in the above summary (in `lm_get_lock`), is purely a node-local operation which does not involve communicating with any other node; only lock and unlock operations (and some LVB operations) do require inter-node messages travelling over the network.

18.4 G-Lock layer

The G-Lock software layer (see Fig. 18.1) provides G-Lock services to the GFS file system code (the top-level interface), uses the services provided by the LM layer (the bottom-level interface), and allows for specialized operations to be plugged in both at the top and bottom-level interfaces. Central to the G-Lock layer are the glock and the GFS holder abstractions; the most relevant operations against the `gfs_glock` structure are (see `gfs/glock.h`),

`gfs_glock_get` - find an existing, or if option allows it, allocate and initialize a new `gfs_glock` structure on **this** node.

`gfs_glock_hold` - increments the glock usage counter.

`gfs_glock_put` - decrements the glock usage counter; if it reaches zero, schedule it for reclaim (it will eventually get destroyed, its memory freed).

while the most relevant operations related to the `gfs_holder` structure are,

`gfs_holder_get` - allocate and initialize a new `gfs_holder` structure.

`gfs_holder_init` - initialize a `gfs_holder` structure in the default way and set its owner, state, flags, and usage counter.

`gfs_holder_put` - get rid of a `gfs_holder` structure, freeing its memory.

Finally, we list some of the lock/unlock operations which are, in fact, a result of enqueueing and dequeueing, as well as promoting and demoting holders onto their associated glocks,

`gfs_glock_nq` - enqueue a glock holder into its glock (i.e., acquire, or lock, the glock; we may get lucky and be granted the lock immediately, or we may have to wait…)

`gfs_glock_dq` - dequeue a glock holder from its glock (release a local process' hold on the glock and service possible waiters; if this is the last holder of the glock – in this node – unlock it, and decide whether to keep it in the glock cache or immediately release it cluster-wide)

`gfs_glock_xmote_th` - call into the Lock Module to lock the glock's LM-lock (which corresponds to placing a lock on the glock), or change an already-acquired lock to a more (promote) or less (demote) restrictive state (other than unlocked – do **not** use it for unlocking).

`gfs_glock_drop_th` - call into the Lock Module to unlock the glock's LM-lock.

Despite having the same prefix, `gfs_glock`, the `_nq` and `_dq` functions act on `gfs_holder` structures, while the `_xmote` and `_drop` act on `gfs_glock` structures. The former, `_nq` and `_dq`, are called in directly by the GFS code layer, while the later, `xmote` and `drop` functions (both top and bottom-halves – not mentioned here) are, as we've seen before, used as a) generic operations for GFS metadata, flock, non-disk and quota-type objects; and b) to link to tailored versions of `xmote` and `drop` functions appropriate for dinode, resource group, transaction, and "general" type objects.

19 Locking in GFS: a in-depth look

19.1 Sharing and locking in local file systems

Even a commonly used local file system, such as ext2 or NTFS, must care about concurrent events that take place in the system: some processes perform actions that need to access, and perhaps concurrently update metadata structures used to manage file system entities such as files and directories, while other processes may be busy sharing – e.g. reading and/or writing – the same file. At the implementation level, steps must be taken to ensure that both file and file system sharing semantics hold, in spite of multiple concurrent operations taking place on in-core data structures; the problem is further exacerbated in operating systems that allow system call preemption, or support parallel execution of kernel threads in SMP environments. These problems are usually tackled with widely known OS synchronization mechanisms – mutexes, spinlocks, semaphores, etc.

The file system model, and particularly its sharing semantics, is important for users because it defines what to expect when active executing entities, such as threads and processes, concurrently access file system "objects"; furthermore, it may provide mechanisms, such as file locks, that can be used to enforce some specific behaviour in the presence of concurrent operations.

19.2 Sharing and locking in distributed file systems

In a distributed file system, where multiple nodes must keep their shared – often extensively cached in memory for performance reasons – copies of file system data and metadata consistent, we need not only to cater for the two intra-node issues previously described, but also a third, new one: how do we solve the shared data consistency problem? An often used solution is to use a global lock, one that could be used to implement synchronized access to these shared data structures, and thus, together with an invalidation protocol that discards other copies when one of them is modified, keep them coherent across the nodes. But how do we implement this global lock?

For non-distributed operating systems (the ones, such as Linux and Windows, we use everyday), global inter-node locks are not an OS provided abstraction and, therefore, we must resort to an add-on software module, a Lock Manager, and modify the code to use the new abstraction to coordinate access to shared data.

19.3 Locking in the GFS world: an overview

In the GFS world, the solution for the three problems previously identified – intra-node and inter-node serialisation, and inter-node coherence of replicated data structures – is subsumed in one concept, the Global Lock (G-Lock): a G-Lock is, as we've pointed out before, a

cluster[1]-wide visible "object" that may be used to support synchronized access (such as mutual exclusion) to protected resources shared amongst participant nodes; requests may originate from local, intra-node, or global, inter-node, processes; G-Locks also support the serialisation of intra-node accesses which are required for correct operation of GFS on SMP architectures. A very short introduction to locking in GFS was recently published in a paper on GFS2 [Whi07].

19.3.1 G-Locks

The G-Lock concept is implemented by the `gfs_glock` (glock) structure and its corresponding set of operations (obviously including some "lock" and "unlock" primitives); we will see more about thus later; for now, it suffices to say that G-Lock usage adheres to the typical lock usage pattern: a) the glock protecting a inter-node shared data structure is locked; b) the desired operation is performed on the structure; and c) the glock is unlocked.

To protect each shared, in-core copy of a particular data structure, a local glock is created in every participant node (i.e., each node that holds the shared structure), and will reflect the node's local view of the global G-Lock abstraction. Some operations on the abstract G-Lock may be purely local, intra-node, operations, while other operations require message exchanges (that may result in queries and/or changes to each local glock state) between the participating nodes and (if properly implemented) result in a coherent view of the G-Lock state among all nodes, each one storing the appropriate state (view) in its local glock structure.

The G-Lock is also abstract in a sort of object-oriented way: when a glock is created, it is assigned to be of some predefined type, one which identifies the kind of object it protects; it has an associated glops vector of operations structure, i.e., a set of functions whose implementation depends on the specific type: a glock created to protect, for example, a GFS inode, is assigned the appropriate set of functions for acting on "inode objects".

A glock is uniquely identified (at inter-node scope) in a GFS cluster by a triplet: lock number, lock type, and lock namespace; we have already seen what the lock type is; the lock number and lock namespace concepts are introduced in the next section.

19.3.2 LM-Locks

As the glock allows us to support two very distinct "usage modes", namely being used at an intra or inter-node scope, the implementers decided to decouple the "local part" from a lighter, more "generic" structure that supports inter-node locking, which we call the LM-Lock (where LM stands for Lock Manager).

Again, this global, inter-node, LM-Lock is a concept; its implementation resorts to a node-local structure of type `lm_lock` (abbreviated lmlock) created in each participant node, much

[1] Our definition of cluster, here, is the appropriate one for GFS: a GFS cluster is a set of SAN connected nodes that share the same "pool" of storage devices (LUNs).

in the same way the G-Lock concept was implemented. The local LM-Lock is implemented by a kernel module, the Lock Manager Module (LMM), in a way that completely decouples it from the rest of the GFS world; the cluster-wide LM-Lock abstraction is implemented by a set of LMMs (as in GDLM) plus additional software, if required (as in GULM).

When a G-Lock is created, a cluster-wide LM-Lock entity must be associated with it; thus, for each node, the LMM must also create a new local lmlock, and attach it to the local glock; i.e., each module-provided lmlock will, when plugged into each node's local glock, "turn" it into a global G-Lock. This lmlock is, similarly to the glock, also a local representative – i.e., will hold this node's vision – of the abstract "cluster-wide lock" maintained by the LM software; for example, if in some designated node the lmlock is held in the "exclusive" state, then we know that all other nodes must have their own local representations (of the same LM-Lock) in the "unlocked" state, and thus only one node effectively holds the "cluster-wide lock". A LM-Lock may be held in one out of four different states: unlocked, shared, deferred and exclusive.

The LM-Lock's "primary identifier" is the lockname, which is a type/number pair; this is also stored in the G-Lock identifier, and thus establishes the relation between these two entities. The LM-Lock lockname "inherits" the type and number from the object protected by the G-Lock it is associated with; e.g., if some G-Lock protects a ginode[2], the lockname structure will hold "type inode" and the block number of the on-disk inode; for structures that do not correspond to existing on-disk entities (e.g., the data structure that holds information about a mount), the lock number is carved on the code, in a ".h" include file.

As we already know GFS supports the coexistence of multiple distinct lock managers that implement different locking protocols, all offering the same functionalities and adhering to the same interface; therefore, they may be used interchangeably – the user should choose the most appropriate for the task at hand. The only added complexity here is that at file system mount time the user must specify which LM protocol instance will be "managing" the GFS file system being mounted, and this constitutes the creation of a new lock namespace. From this point on, all the glocks created within that lock namespace (LM protocol/file system pair) will trigger the creation of lmlocks, by the appropriate lock manager instance. So, the "full identifier" for a G-Lock or LM-Lock is in fact the pair namespace/lockname, which turns out to be the triplet namespace/type/number we've previously described.

Finally, we must point out that another level of decomposition may exist in the implementation of a locking protocol; this may be seen, for example, in the GULM, which is a software-only, client/server implementation of the proposed (but not accepted) SCSI Device

[2] A ginode is a GFS inode structure (an in-core image of a dinode) linked to the VFS vnode structure through its generic_ip field.

Memory Export Protocol (DMEP) [Bar+00][3], where two separate entities are required to implement the protocol: a "server", that implements lock storage (a user-level daemon process, in the `GULM`, or RAM in a disk or disk array supporting DMEP), and clients (the kernel lock modules in the nodes), that access the "server" when needed. This is not the case with `GDLM`, where no "central point" of storage does exist.

19.3.3 G-Lock holder

"Once a node has acquired a glock, it may be shared within that node by several processes, even by several recursive requests from the same process". This statement, extracted from the include file ("`gfs/incore.h`") illustrates the new terminology we will be adopting from now: several nodes may be able to acquire[4] a G-Lock in a shared state (and each of them will have its local glock in the shared state), or one of them may be able to get it in the exclusive state (and others will have their local glock "unlocked").

A process expresses its interest in issuing an operation on a glock by creating a "request packet", the GFS holder structure, that will store both the process identification (in a owner field), and the desired conditions under which the operation is to be successfully granted, e.g., "I want to hold the glock if nobody else has it", or "I want to hold the glock but I'm willing to share with others". The request (holder) is then "linked" to its "target glock" and submitted (enqueued); if it gets immediately granted, it is attached to the glock holders list; otherwise, it will be attached (queued) to a waiters list, awaiting promotion.

A holder is, then, a purely node-local structure that allows us to: a) acquire a glock in some desired state; b) coordinate how it is shared among processes in the node; and c) finally, release it. Thus, it is possible for a glock to have several simultaneous holders: distinct processes (owners) that were able to share it; several compatible "recursive" requests issued by the same owner; or a mixture of both.

19.3.4 G-Lock operations

The G-Lock operations structure allows us to further refine the G-Lock by associating an implementation-specific vector of operations to a G-Lock; as an example, for a glock that protects an inode, the glock's vector of operations "generic" `go_sync()` function maps to `inode_go_sync()`, which triggers a flush of all data and/or metadata associated with an inode when, for example, it is unlocked. However, for a glock that protects a RG, the glock's "generic" `go_sync()` function maps to `meta_go_sync()`, which synchronously flushes all buffered metadata associated with the RG.

---

[3] Which originated as the Device Lock (DLOCK) [Sol97]

[4] To acquire a G-Lock, we need to lock the LM-Lock, so, in a way, acquire (a glock) and lock (its corresponding lmlock) is equivalent; one "drags" the other.

19.3.5 Performance-driven implementation decisions

We will now introduce the topic of performance in this discussion; as usual, it will complicate things a bit but we will hope that, having presented a clear picture before, the reader will not get confused.

When an object such as a ginode, together with all the "companion" data structures it references, is protected by a glock and that glock gets released, data must be flushed to disk; this is a costly, time consuming operation, that gains by being postponed as long as possible (much in the same way the Linux page cache supports write-back).

But how long is "as long as possible"? If a node wants to read data that has been modified in another node, the later must flush it before the glock is acquired by the former; but if a set of processes, all running in the same node, are reading and writing a file, there is no need to flush data, and we would still get more performance if we'd refrain from repeatedly dropping and re-acquiring the file's ginode glock (as these operations must be carried out by exchanging messages across the network). GFS implements these performance enhancement features by tying the flush operation to the drop of a glock, and postponing the (cluster-wide) drop of a unheld glock by keeping it in a cache (with the same status that was stored by the time the last holder was dropped) until it expires, or is forcefully "called back" by another node[5]. So, for "typical" applications, when a process needs to re-acquire a recently released glock it immediately succeeds, getting it from the cache and, as a side-effect (but a major one, for increased performance), data that has been recently accessed still lives in the page cache.

19.4 An example-driven operational overview

We will now try to tie some of the concepts previously introduced, namely lmlocks, glocks and holders, by resorting to a complete example: in a GFS cluster, the file F is, for the first time, opened for reading in node A; then, it is opened for writing in node B. Now, a process in node A starts reading the file, while another process in node B writes to it; for simplicity, let us assume that no user-level locking is involved (and that this does not constitute a problem for the application).

19.4.1 Opening the file

The most relevant (and over simplified) operations for the open are:

1. A pathname transversal (sequence of lookups) is performed; at the end, the file's "inode" number is found from a dentry, and the ginode of the directory which contains the file gets a shared lock on its glock;
2. An new (empty) ginode in-core data structure is created, together with its new protecting glock; the glock is tailored with the appropriate operations for "(g)inode-type objects";

[5] There are situations where this postponing is not possible, but we will not cover them here.

3. A holder is created to allow us to request a lock on the glock; the request will be for a cluster-wide shared lock, but with an exclusive flag set for this node only. The request is then submitted; when granted, the holder is attached to the glock holders list: "I hold a shared lock on this file's ginode"
   3.1. The Lock Manager is called to find an existing lmlock or create a new one; the parameters lock number (drawn from the block number of the file system block that holds the dinode), lock type, and lock namespace are supplied; the inter-node lock request is for "shared".
   3.2. A new LM-Lock must be created (this is the 1st lock on that dinode number); storage for the lmlock object resides, in the GULM case, on the GULM Lock Server, whereas in the DLM case, it may be "duplicated" in several nodes.
4. The in-core dinode field of the ginode object is filled in with data retrieved from the on-disk dinode blocks;
5. The ginode glock's "local exclusive" flag is downgraded to "local shared".

Now, if the file F is to be opened again, but this time on node B, the only difference is on step 3.2: a new "cluster-wide" LM-Lock will not be created by the LM because one (for that type, number and lockspace) already exists; only the lmlock "local structure" will be created, and its status set to indicate it as being currently locked in shared mode; all the other steps will be performed exactly in the same way.

19.4.2 Reading a GFS file

Let us assume node B is not yet writing; the relevant (and, again, over simplified, assuming regular non direct-I/O) operations are:

1. The `read()` call enters the kernel; the normal flow through the VFS layer is performed, i.e., from the VFS file object its vector-of-operations read-function is called:
   ```
   file->f_op->read(…)
   ```
   For a GFS file, this function is mapped into `gfs_read()`;
2. The `gfs_read()` code enters execution;
3. A holder for a shared lock is created and submitted (a.k.a. enqueued) onto the ginode's glock; upon return, the lock is held;
4. VFS level functions, such as `generic_file_read()` are used to get into the node's page cache and retrieve data, or, if needed, submit a low level read to the disk driver; when data is available, it will be copied to the user buffer; (this highlights the tight integration of GFS into the VFS subsystem);
5. The holder is de-queued, unlocking the glock, which then:
   5.1. gets moved to the glock cache, where it stays until it: is requested again; expires and is released cluster-wide; or is released because it was "requested" from another node.
   5.2. if released (because it expired or was forcefully called from other node) a glock operation is performed, and that operation may act on the file's data (e.g., invalidate all cached data) and/or metadata (such as updating the time of last access).

6. The `gfs_read()` returns;
7. The `read()` returns;

To complement the discussion above, it is worth pointing out that if an application has one or more processes that repeatedly read the same file, there is no inter-node traffic, as the glock(s) will stay cached in their nodes, making their lock/unlock purely local, intra-node, operations. Only the first locking operation requires a LM access which will need messages travelling to other node(s).

19.4.3 Writing a GFS file

The flow for the `write()` call is similar to the one above, but much more complex at the GFS level; we omit a lot of details, but the important ones are:

1. The `write()` call enters the kernel; the normal flow through the VFS layer is performed, i.e., from the VFS file object the vector-of-operations write is called:
   `file->f_op->write(…)`
   which, for a GFS file, is mapped into `gfs_write()`;
2. `gfs_write()` enters execution,
3. A holder for an exclusive lock is created and then enqueued onto the ginode's glock; upon return, the lock is held (we will have to wait if any other node has a lock, shared or exclusive, on the glock);
4. (A lot of details omitted here.) The user buffer is copied to kernel space, merged into the page cache; affected pages are marked dirty and linked into the vnode's dirty list; these pages will be flushed regularly by the Linux `pdflush` daemon, or synced on demand;
5. The holder is de-queued, unlocking the glock and, in the same way to what happened with the `read()` above, the glock will stay in the node's glock cache.
6. `gfs_write()` returns.
7. The `write()` returns.

The above description shows why GFS stumbles when, for example, one node is reading a file while another is simultaneously writing it: for every write, the reader has to release its glock immediately (and invalidate the data it has cached so far), as the writer needs it exclusively; so, for every operation, messages are exchanged between the node and the global Lock Manager, and glock caching, as sketched in 4.1-iii above, is useless. To worsen things up, on the next read, the reverse occurs, more traffic travels through the interconnects, again, and the writer must flush all file data and metadata to disk before releasing the glock, as the reader may decide to access data that has been changed by the writer; this requires a string of flush-to-disk operations that, even when a large bandwidth is available from the I/O infrastructure, nevertheless cause a latency build-up that severely degrades each node's sustained bandwidth, as inferred in sections 8 and 9 (e.g., as computed in equation 8.5) and reported in Figs. 16.4 and 28.9.

19.5 Keeping metadata coherent across cluster nodes

When talking about a “coherent view across cluster nodes”, we may separately address two aspects: keeping user-visible (or user-level) objects such as files and directories, coherent, and keeping file system “internal” metadata structures such as inode and data block bitmaps (or other structures used for free/used accounting of blocks and inodes), coherent; failing to provide the first may result in application problems, but failing the second will undoubtedly result in a corrupted file system which, sooner or later, will cease to function.

19.5.1 Coherent file system metadata management in GFS

A coherent view of free vs. allocated blocks is needed across all cluster nodes which have mounted some particular file system, to support correctness in the presence of operations such as concurrent file creation (dinode allocations), removal (possible dinode de-allocations), and writing (which may result in a file size increase and thus require metadata and/or data blocks to support that growth). The problem of a coherent view of block (de-)allocation across cluster nodes is therefore a major issue at the file system level (there’s, of course, more to proper metadata management than just managing bitmaps, as we’ll show in the next section).

As it happens with many other file systems, GFS use the concept of Resource Groups (RG), which are similar to Berkeley’s Fast File System cylinder groups [McK+84] and have been adopted by a legion of followers, including ext2. A RG is a sort of “mini file system” with a superblock, and two regions, one for inode and another for data blocks, along with their corresponding bitmaps. To perform an operation on a specific RG, GFS places a lock on the “RG-type” glock that protects the RG; as expected, allocation and de-allocation operations with RGs’ glocks exclusively held are sufficient to guarantee a coherent view of those RGs across cluster nodes.

19.5.2 Coherent file metadata access in GFS

Guaranteeing that every node has a coherent view of free and allocated resources, such as disk blocks, however, is not enough, as it could lead to situations quite similar to the problem of lost updates, but now with the file’s metadata; as an example, two processes in distinct nodes could be “filing in” sparse holes, each one in its respective (non-overlapped, even at the page level) region; if both were modifying the same metadata portion, *e.g.*, distinct pointers in the same indirect block, the last writer would superimpose stale data over some part that had already been modified, and flushed, by the other node.

Obviously, concurrent access to file metadata structures must also be properly carried out, and GFS has the right mechanism for the job: while executing the `gfs_write()` the ginode glock is exclusively locked, and other nodes cannot keep any data or metadata from that file in their caches.

Part VII:

pCFS Implementation

In this Part we describe how pCFS is implemented, through the addition of two kernel modules, a user-space daemon, and slight modifications to GFS code; the modified GFS code distributes information about clusterwide open files and active regions, and implements cache coherency without resorting to expensive disk flushing and cache invalidation operations.

20 Prototype Implementation: introduction

20.1 Overview

The term “prototype” clearly states we’re not aiming either a full or a production-quality implementation; our primary objective is to show that pCFS, while retaining GFS’ strengths, can efficiently support HPC applications, so the prototype specifically target regular files accessed through the usual read/write and other POSIX calls, eschewing direct I/O and memory-mapped operations. Furthermore, no modifications were made in code paths that handle directories, special files, journaling, etc., i.e., no attempt was made to speedup metadata operations (such as file creation, deletion or lookup).

20.1.1 Rationale

Some discipline was imposed on the prototype implementation; important restrictions we wanted to enforce were: i) no modifications to Linux API (no new *syscalls*) or changes to the VFS layer were allowed; ii) modifications to GFS code should be kept to a minimum, even at the expense of having to duplicate GFS code into pCFS-specific modules. Benefits from (i) are clear: pCFS will run on any distribution that supports GFS (currently, Red Hat Enterprise Edition), and existing applications may run unmodified, while (ii) means that it should be easy to keep pCFS in sync with new GFS releases, as burden is confined within pCFS’ own modules (where it should be easy to manage); furthermore, it should be possible to execute “near native” GFS, which will greatly simplify debugging and pCFS-to-GFS benchmarking.

20.1.2 Implementation strategy

We’ve chosen to incrementally develop the prototype; in the first phase, we concentrated on delivering support for high performance I/O for those situations were no data or metadata allocation (*e.g.*, indirect blocks) was required – i.e., an existing file is simply read or rewritten; in the second phase, we handle cases where metadata allocation is required.

20.1.3 Dealing with adversities and uncertainties

As time went by we faced several obstacles; some were just plainly annoying, but others forced us re-evaluate our initial goals. The first class includes the Linux kernel internals, quite undocumented in the file system layer (VFS *et al*). The second is a consequence of internal changes in the interfaces, which happen quite often and across minor releases, too; and, consequently, some particular software becomes strongly tied to a particular kernel release, while another one only works with a different release, making it difficult to use them together.

As an example, we were planning to implement pCFS’ cooperative caching with Kerrighed containers [Lot01, Mor+04]; however, this was not possible because the version of the containers module depends on a particular release of the Transparent Inter-Process

Communication (TIPC) protocol [TIPC] which was not compatible, at that moment in time, with any kernel version which would support GFS. Therefore, we are trying to bring it all together, and believe that, afterwards, a full implementation of pCFS' cooperative cache may be easily achieved.

20.2 pCFS: architecture and operation overview

Most pCFS code is split between two kernel modules, pCFSk and pCFSc, and a user-level daemon, pCFSd; furthermore, a very small amount is delivered as patches to GFS.

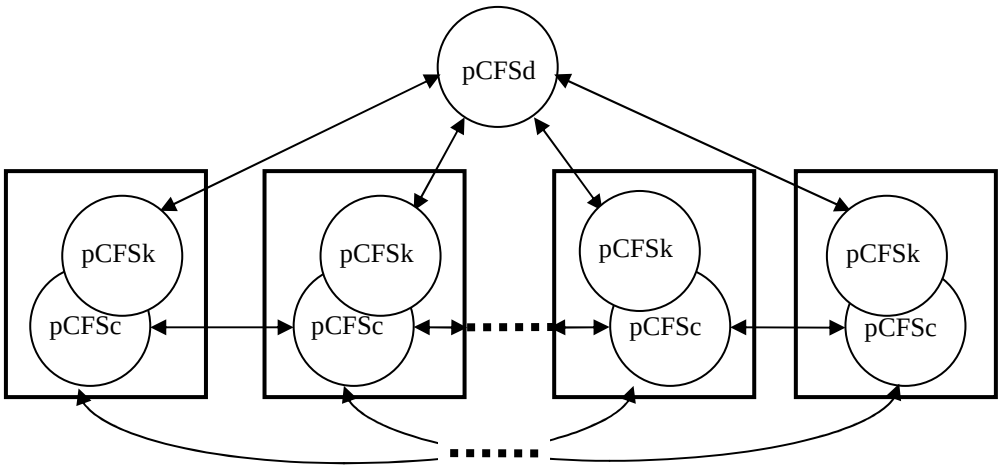


Figure 20.1 pCFS architecture and module interconnections

Each node has both a pCFSk and a pCFSc module, while there is a single pCFSd instance per cluster; a brief description of each component follows:

- Each pCFSk maintains a "local database" that stores information about per-node relevant data structures, *e.g.*, pCFS inodes (*i.e.*, those corresponding to files opened with one of the pCFS flags), and, for each file, the list of "active regions". Each pCFSk opens a TCP stream against the pCFSd, which handles in a separate thread.
- pCFSd maintains a "global database" of cluster-wide relevant data structures, a sort of "union" among the structures pCFSk maintains at each node; when necessary, pCFSd sends invalidation messages to the pCFSc modules in selected target nodes.
- Each pCFSc maintains per-node "in-flight" data that must be shipped to/received from other nodes and then updated into the VFS page cache. Furthermore, pCFSc maintains coherence by flushing out and/or invalidating selected pages from the node's page cache.

To give the reader a brief introduction to the pCFS operation, we start by stating that, for pCFS-modified file calls, when the user process performs a file operation, *e.g.*, a `read()` on a "pCFS file", a GFS path that leads to a pCFSk call is taken; if call processing can be handled locally in pCFSk, it "immediately" returns to the GFS regular code; otherwise, pCFSk will exchange information with pCFSd, and will either return to GFS code (for local data access), or take a different path, fetching/delivering data from/to a remote node.

20.3 Phase 1: High performance R/W with no metadata allocation

HPC applications usually share a file in a way that processes running in different nodes access disjoint, non-overlapping sections of the file; this implies that data coherency is not, in general, an issue. In the first phase of the prototype implementation we devised a way to easily improved bandwidth by explicitly requiring participant processes to define regions before they access a file, releasing them when they are no longer needed; to be used in isolation (i.e., not complemented with other approaches), however, it requires that there are no major metadata changes to the file, i.e., its size must be kept constant and, in the event the file is sparse, no "holes" should be "filled in".

20.3.1 Overlapping vs. non-overlapping file access operations.

But, even when not overlapping at the byte-level, regions may well overlap when larger units – blocks or pages – are considered, as shown in Fig. 20.2 below.

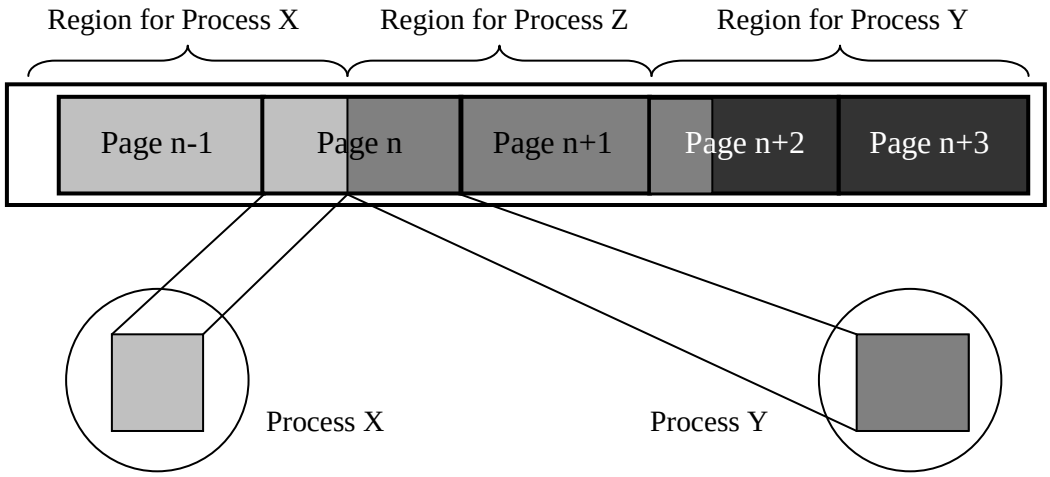


Figure 20.2 False sharing and lost update (last writer "wins")

As previously shown (section 11.2), this may raise coherency issues. Given that the unit of caching at the file system level (at the page cache) is the page, we identified the following issues due to false sharing:

- **Un-harmful**: There is only one writer node. Even if a reader node has an out-of-date page cached, one where a portion written in the writer node is not up-to-date, pCFS processes in the reader node are not allowed to access that portion, so consistency issues do not exist.
- **Harmful**: There are multiple writer nodes. This may trigger lost updates, as follows (Fig. 20.2): a writer in node X writes into its "side" of the cached page; another writer in node Z does the same to its "side" – notice that byte level overlapping does not occur; then, it does not matter which node is the first to write out "its" data (page n, in the figure) to disk: others (e.g., Y) may not be able to retrieve the first node's (e.g., X) updated portion of the data from the moment the second node (e.g., Z) writes out its image (of page n, again) on disk.

20.3.2 pCFS major data structures

The two structures depicted in Fig. 20.3 are the foundation of pCFS' improvements over GFS: `pCFS_inode`, which records data about an open pCFS inode, and `pCFS_region`, which records data about an active region placed by some process over a file.

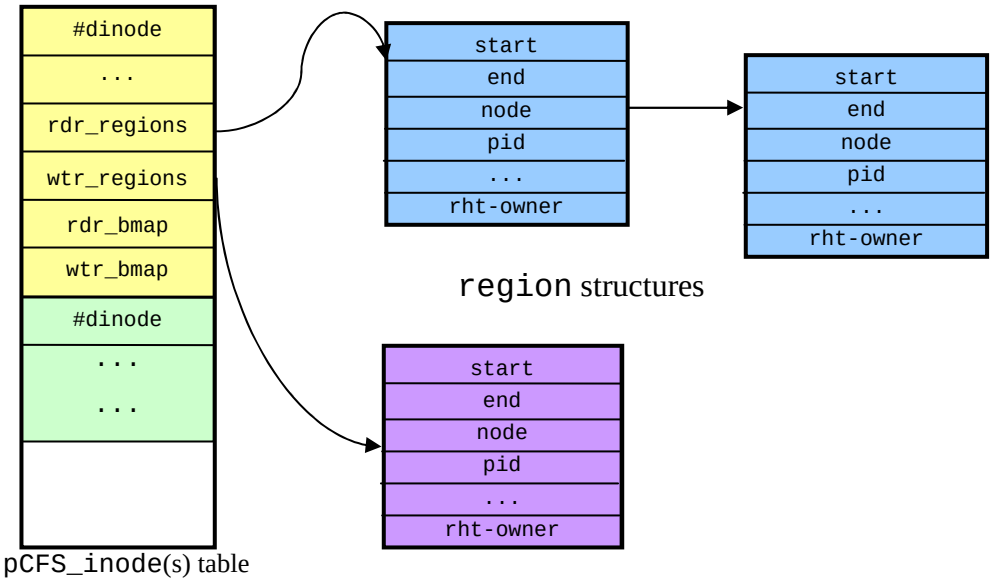


Figure 20.3 pCFS major data structures and their relationships

There are two hash tables[1] of `pCFS_open_inode` structures: a "local database", in pCFSk, for files opened in the node; and a global database, in pCFSd, for files opened across the cluster. The structure of a `pCFS_open_inode` is:

```
struct pCFS_open_inode {
  uint64_t              dinode;
  unsigned int          count;
  unsigned int          mode;
  unsigned int          owner;
  unsigned int          fwrdrs;
  struct pCFS_region_l  * rdr_regions;
  struct pCFS_region_l  * wtr_regions;
  unsigned long         rdr_bmap;
  unsigned long         wtr_bmap;
};
```

where

`dinode`	Identifies the on-disk (and in-core, as they are the same) file inode.
`count`	Number of outstanding opens.
`mode`	Reserved (currently unused).
`owner`	If non zero, identifies the file owner.
`fwrdrs`	If non zero, there are owner(s) (for boundary pages) in the region lists.
`rdr_regions`	List of regions laid out by reader processes.
`wtr_regions`	List of regions laid out by writer processes.
`rdr_bmap`	Bit map of nodes reading this file.
`wtr_bmap`	Bit map of nodes writing to this file.

[1] Named `pCFS_opens`, and currently implemented as fixed-size arrays of `pCFS_open_inode` structures.

As for a region, its structure is:

```
struct pCFS_region {
  loff_t        start;
  loff_t        end;
#ifndef __KERNEL__
  node_t        node;
#endif
  pid_t         pid;
  node_t        ownerL;
  node_t        ownerR;
  unsigned int  flags;
};
```

where,

`start`	Marks the byte offset at the start of the region.
`end`	Marks the byte offset at the end of the region.
`node`	(Only for pCFSd regions) Identifies the node that laid out the region.
`pid`	Identifies the pid that laid out the region.
`ownerL`	Accesses to the left boundary page must be forwarded to this node.
`ownerR`	Accesses to the right boundary page must be forwarded to this node.
`flags`	Consistency checking: must be either `F_RDLCK` or `F_WRLCK`.

20.3.3 A brief look at the major file operations

We now look at how pCFS changes were introduced in the GFS code, and follow them with an execution scenario for a typical application which uses the most relevant file operations: we start with an `open()`, followed by an "insert region" with `fcntl()`, then we perform a `read()` and a `write()`; before terminating the application, we lift the region with another `fcntl()`, and finally `close()`.

20.3.3.1 `open()`

Each time a process in node performs an open of a "GFS file" with a pCFS option flag or'ed in, when control reaches `gfs_open()`, a pCFSk function is invoked to:

1 Check if the file is already open in the node (find a `pCFS_inode` with a matching dinode)

1.1 If found, a check is made for the presence of `O_CLSTXOPEN` (cluster-wide exclusive) or `O_NODEXOPEN` (node exclusive) – an error situation, in both cases, and we return.

1.2 Else, a message is sent to pCFSd to check on the global database that the open does not conflict with other outstanding opens of the same file; if it does, take an error return.

2 Otherwise, entries may be created and/or updated at the local and/or global level; this includes incrementing the inode's usage count and updating reader/writer bitmaps at both "sites".

We end up either allowing or rejecting the open, and both the local node and the pCFSd tables are updated accordingly. Notice that some fields may exist in pCFSd "versions" of the structures, but not in pCFSk – an example being the node field which exists in the pCFSd `pCFS_region` structure, but not in the one for pCFSk.

20.3.3.2 Region processing: laying out a new region with `fcntl()`

When a user process, calling `fcntl()` in the same way it does to place a POSIX advisory lock, lays out a new region over a pCFS file,

1 Normal VFS first, then GFS pre-processing are carried out;

2 GFS sends out a request to the Lock Manager asking for a shared (for a read lock) or exclusive (for a write lock) clusterwide LM-lock with a "POSIX lock" tag as key. (Notice that GFS calls the LM layer directly, not the G-Lock layer);

3 If successful, we know the lock is clusterwide valid, so we call pCFSk code to:

3.1 Build up a new `pCFS_region` structure, storing the start and end of the file region (byte offsets), the pid of the requesting process, the lock flag (`F_RDLCK` or `F_WRLCK`) and inserting it in the appropriate order (key: start, end, node, pid) in the list (either `rdr_regions` or `wtr_regions`, depending on the flag).

3.2 Send a message to pCFSd with region information (including the node id), in order to get the region placed in the global "pCFSd database"; this is where the bitmap structures, `rdr_` or `wtr_bmap`, depending on the flag, get the node bit updated; in the reply packet, the bitmaps are sent from pCFSd to pCFSk, where they are used to update the node's knowledge about which nodes are currently using the file, either for reading or for writing.

3.3 We return to GFS code.

4 GFS code returns to the VFS code, which places a "POSIX lock object" into the appropriate inode (vnode) list and returns to the user.

20.3.3.3 `read()`

The code is quite similar to the one already sketched in 19.4.2 except for the patches, which we include here bracketed by "pCFS begin" and "pCFS end":

1. The `read()` call enters the kernel; the normal flow through the VFS layer is performed, i.e., the file object's vector-of-operations function is called: `file->f_op->read(…)`. For a GFS file, this function is mapped into `gfs_read()`;
2. The `gfs_read()` code enters execution
   2.1. `/* pCFS begin */` If accessing a pCFS file, we call pCFSk code to assert that read boundaries are within a valid pCFS region; otherwise, we'll bail out with error, and skip to 6. `/* pCFS end */`
3. …
6. The `gfs_read()` returns;
7. The `read()` returns;

Notice that we do not modify anything else; in particular, we do not care if we're reading from a page that has some portion of stale data – which we **can not access** because it is located outside our region, and we would have take an error return in 2.1 above.

Verifying that the file "is a pCFS file", locating its inode entry and performing the validity check in the region list are pure intra-node operations carried out at the pCFSk module and

are, therefore, very fast; so, the overhead introduced is negligible, as reported in [Lop+08] where pCFS reads were within 1% of those of GFS.

20.3.3.4 `write()`

As the user process writes, when the `write()` code path reaches `gfs_write()` it will execute new code introduced to check if a pCFS file is being accessed (if not, regular GFS processing continues) and, in that case, if the request is within a valid region (if not, an error is returned). *The pCFS difference is that*, now that the access has been verified and granted, *we can*, even for a writer node, *ask for a shared glock against the file inode*, and resume regular GFS code (assuming no false sharing problems and/or metadata allocation), which will access data either from the page cache, or from disk, using the SAN infrastructure.

To prevent a reader in a node from reading data which has been modified by a writer in another node, that data has to be either flushed to disk or moved through an interconnection infrastructure; flushing data to disk is important because it makes it permanent, but should not slow down other nodes' file access operations – something that "mainstream" CFSs such as GFS can't do. *Our strategy for flushing does not slow down other nodes' file access operations* because it does not require (in the absence of metadata allocation) either exclusive access – locking the ginode glock in exclusive mode, as shown in 19.4.3 and which would trigger invalidations sent to other node's caches – or "frequent" (as in per-call) flushing.

20.3.3.5 `fcntl()` again: region removal and data flushing

As the writer process removes ("lifts") the region, we must guarantee that modified data is committed to disk before a process in another node may read it *from disk*. We opted for the easy solution: a synchronous flush in the moment the region is removed; delaying the flush up to the moment where the access is needed by the other node is too complex, unless we are running a SSI operating system, which offers a page flush/invalidation mechanism "for free". So, when lifting a region, with `fcntl()`,

1 Regular VFS first, and then GFS pre-processing is carried out;

2 GFS processing includes sending out a Lock Manager request asking to drop[2] the cluster-wide LM-lock with a "POSIX lock type" tag. In case of a successful return, we call pCFSk code to:

2.1 Perform the flush.

2.2 Remove the `pCFS_region` structure, from `wtr_regions` and free its memory.

2.3 Send a message to pCFSd with the region information so that it can also remove that region; but, before doing it, (i) pCFSd uses `rdr_bmap` and `wtr_bmap` to get the list of the nodes currently using the file and, (ii) sends out *invalidation messages to pCFSc* modules on those nodes, noting the region start and end – we need this because in reader nodes operating "close" to the regions' borders VFS code could have been reading-ahead into another node's region, and

[2] We're assuming a write lock here, so there is only one!

if we didn't invalidate that data, once the node laid out a valid region over it, it could access stale data[3].

3 GFS returns to VFS code, which removes the "POSIX lock object" from the appropriate inode (vnode) list and, finally, returns to the user application.

In a typical HPC application regions are often quite large (tens of MB or more) so while a process is writing, file system/OS flushing does occur from time to time, triggered by memory pressure and/or cache expiration; our lifting mechanism just guarantees that all writes are flushed before "letting another node in" that same byte range.

20.3.3.6 `close()`

Currently, pCFS close processing is very simple, because we choose to return an error if a close is attempted on a file that has outstanding active regions (*i.e.*, we currently require the programmer to lift every region that was laid out before closing the file). If ok to close, we just decrease the `pCFS_inode` entry count field and, if it reaches zero, de-allocate (currently we just clear it) the structure. Although we already provide a pCFSd reply message informing a node that it is the last node in the cluster that is reading/writing/closing a particular file, we do not yet take advantage from that piece of information.

20.3.4 Forwarding: using the LAN to solve the "lost update" problem

To solve the lost update problem (and others which will shortly be discussed) we've added the left and a right owner fields to the `pCFS_region` structure; they are used as follows (using Fig. 20.2 as guidance, assuming all processes are writers and `X` lays out its region first, then `Y` and finally `Z`):

- When a process lays out a **writer** region, a check is made to see if it has **writer** neighbours and if their respective boundaries overlap at a page level; if they do, we signal its neighbour as the page owner. For example, page `n` is shared between processes `X` and `Z`, while page `n+2` is shared between processes `Y` and `Z`. When the `Z` writer finally lays out its region, the check to see if there are pages shared between `Z` and a writer region "left-neighbour" (here, `X`) and/or a writer region "right-neighbour" (here, `Y`), returns true for both, so the left-owner field of `Z`'s `pCFS_region` is set with `X`'s node id while the right-owner is set with `Y`'s node id.
- For each `write()` we check if the write will touch the regions' first and/or last pages and, then, whether left and/or right owners exist; if they do, a pCFSk function is called to forward that data to the neighbour's pCFSc, which then inserts it into the node's page cache.

Furthermore, nodes that "forward" data from the file to other nodes (a non-zero in the `pCFS_open_inode frwdrs` field) should not be allowed to keep data which belongs to those shared boundary pages in their page caches; this implies that reads are also affected, and

[3] Notice that we could disable read-ahead, but this would be, as a rule, detrimental to performance.

if a non-owner writer needs to re-read data “sitting” on a boundary page, that data will be fetched (through the interconnect) from its neighbour’s cache.

20.4 Phase 2: pCFS support for coherent metadata management

We have previously, in 19.5, mentioned that there are two separate aspects on coherency: those related to user-level visible objects, such as files and directories, and those related with file system “internal” metadata structures, such as inode and data block bitmaps. Then, we saw that GFS handles coherent management of file system metadata structures using two different approaches: for resource groups, GFS uses the RG-specialised glocks (19.5.1); but for everything else – and this includes both data (file) and metadata (inodes, index blocks) structures, GFS uses the same strategy to enforce coherency: a per-inode global lock enforced both in `gfs_read()` and `gfs_write()`.

In this subsection we look at new ways used in pCFS to promote coherency without severely degrading, as GFS does, file system performance.

20.4.1 Resource group handling in pCFS

As previously pointed out, to perform an operation on a specific RG, GFS places a lock on the “RG-type” glock created to protect that RG; as expected, allocation and de-allocation operations require the RG’s glock to be held exclusively. For pCFS, we found that this does not, in general, degrade bandwidth and, therefore, we kept the standard GFS operations for resource group handling.

20.4.2 Coherent block allocation/de-allocation at the file level

Two operations may result in major changes to a file structure and, consequently, to the file system where it lives: `truncate()` and `write()`. GFS handles `truncate()` through an exclusive glock, and we are not interested in pursuing a different path; it is not, after all, a common operation – and, anyway, if we truncate a file we must invalidate all file’s data and metadata cached across nodes. As for the write operation, it may trigger major changes when data blocks, index blocks, or both, must be allocated, either as a result of an increase on the file’s size, or because holes in a sparse file get “filled in”.

To support coherent block allocation across nodes sharing the same file, the region mechanism is not enough because, if we relied only on regions, we could end up in a situation similar to lost updates, but now with file’s metadata; as an example, two processes in distinct nodes could be “filing in” sparse holes, each one in its respective (non-overlapped, even at the page level) region; if both were modifying the same metadata portion, *e.g.*, distinct pointers in the same index block, the last writer would superimpose stale data over some part that had already been modified, and flushed, by the other writer, and we could end loosing a big amount of data, e.g., if we lost the “head” of those newly allocated index blocks.

We have three alternatives to handle coherence:

- Select, whenever there is a possibility of major changes in the file's metadata, GFS standard behaviour through a "regular" `open()`, loosing the performance we could achieve with pCFS;
- Use the pCFS approach for all I/O which does not require block allocation and, when needed, temporarily revert to the standard GFS behaviour;
- Select a "master node" and have all the others ship data to/get it from that node.

20.4.2.1 On mixing GFS and pCFS opens

In the above list, the first alternative, resort to plain GFS, although possible is undesirable and should not be used, for two reasons: first, it may be quite difficult for the application programmer not only to assert if metadata will be changed, but also to rewrite the application in a way it will either execute a GFS or a pCFS `open()`, and then follow the exact path with the appropriate "programming style" for that choice; but, more importantly, in the current pCFS prototype one should not concurrently open the same file using both GFS and pCFS "styles", because that will lead to data and metadata inconsistencies across nodes.

20.4.2.2 Handling pCFS metadata coherency through lock promotion

For the current prototype, we have implemented the second alternative as follows: if code execution in the `gfs_write()` takes a path which leads to block allocation, the file's ginode glock which, per pCFS changes, was acquired in the shared state, is re-acquired in the exclusive state; this guarantees that, before the exclusive state is granted to the writer, all other nodes will invalidate both metadata and (unfortunately) data pertaining to that file from their caches and, on subsequent accesses, they will get fresh copies – either from disk, or from the writer's cache. We expect that, in those situations where block allocation is an infrequent event, this strategy of "last minute" promotion of the glock to an exclusive state will not result in a sizeable performance slowdown[4].

20.4.2.3 Implementing the data shipping approach

For those cases leading to what would be a very high number of repetitions of the pattern "region-in; read (and/or) write; region out" for small buffer sizes, or for those cases where there is a high number of operations that require allocation of data and/or metadata blocks, shipping data to a single owner may be the best solution, from the performance point of view.

The data shipping approach is an extension of the forwarding technique introduced in 20.3.4 to overcome false sharing, and can be implemented through the left and/or right-owner fields in the `pCFS_region` structure. A situation where all nodes ship to a single "master" node can, therefore, be easily implemented – we just need to assign the master node id to both owner fields in the file's region structure for every node but the master.

[4] Unfortunately there are some stability problems with this feature, leading to FS crashes...

In the current prototype, the following features, although useful, are not yet implemented; sorted from easiest to the more difficult to implement, they are: user-assigned master (at open time, with a flag, or at runtime, with a call; probably `ioctl` or `fcntl`); master re-election (required if the current master retires); automatic resizing of regions (based on file access pattern discovery, "the system" could automatically lay-out/remove regions so that a route to pCFS' performance features could be offered to an application that does not use regions – see below).

20.4.3 pCFS access without regions

As explained in 16.3, to perform file access with pCFS the user (programmer) may define non-overlapping regions for each process; when he/she chooses not to specify regions in a file, data shipment is used to perform file access. To prepare for data shipment, in the ongoing implementation, the **first** node to perform a **write** is elected the master node; a region covering the whole file is created automatically (the region also covers file growth, as the region end is set to infinity). Subsequent nodes accessing the file also have regions covering the whole file and having the owner fields pointing to the master automatically created for them. Non-master nodes do not, of course, cache data.

21 pCFS kernel modules

21.1 Introduction, function naming and implementation notes

As shown in Fig. 20.1, the current pCFS prototype is built around a set of two kernel modules per-node (pCFSk and pCFSc), plus a single user-level demon for the whole cluster. For ease of reference we'll insert a shortened description of each module's purpose:

- Each pCFSk maintains a "local database" that stores information about per-node relevant data structures, and opens a TCP stream to pCFSd, which is handled there by a separate thread.
- pCFSd maintains a "global database" of cluster-wide relevant data structures; when necessary, pCFSd sends invalidation messages to the pCFSc modules in selected target nodes.
- Each pCFSc maintains a per-node buffer for data that must be shipped to/received from other nodes. Furthermore, when requested to do so, pCFSc maintains coherence by flushing out and/or invalidating selected pages from the node's Linux page cache.

Code inserted (patched) into GFS' kernel module bridges GFS with pCFS, as it calls pCFSk functions which, in turn, interact with pCFSd (which may then interact with pCFSc). In brief, naming rules are:

- Code patched into GFS is referred to as the "pCFSm layer", and macros and functions will bear the `pCFSm_` prefix.
- Functions exported to GFS will be prefixed with `pCFSm_`; if a function interacts with other modules (including pCFSd) it will be also tagged with the `clst_` prefix.

21.2 Patching GFS: pCFSm code

Currently, the pCFSm layer implements two macros:

```
#define PCFSm_IS_FILE_PCFS(file) \
          ((file)->f_flags & (O_CLSTXOPEN|O_NODEXOPEN|O_CLSTSOPEN))

#define PCFSm_IS_GLOCK_PCFS(glock) \
          pCFSk_is_ginode_pCFS((&((struct gfs_inode *) \
                                  ((glock)->gl_object)) ->i_num)->no_formal_ino)
```

Their purpose is:

`pCFSm_IS_FILE_PCFS()` Tests whether the VFS `file` object refers to a pCFS file.

`pCFSm_IS_GLOCK_PCFS()` Tests whether the GFS `glock` (is attached to a `ginode` that) refers to a pCFS file.

21.3 The pCFSk module interface

The list of pCFSk module's exported functions is:

```
int pCFSm_clst_open(uint64_t dinode, unsigned int o_flags);

int pCFSm_clst_prepare_close(uint64_t dinode);

int pCFSm_clst_commit_close(uint64_t dinode);

int pCFSm_clst_region_in(uint64_t dinode, loff_t start, loff_t end,
                            unsigned int flags);

int pCFSm_clst_region_rm(uint64_t dinode, loff_t start, loff_t end,
                            unsigned int flags);

int pCFSm_clst_region_vrfy(unsigned int rw, uint64_t dinode, loff_t offset,
                              loff_t len, int *ownerL, int *ownerR);

int pCFSm_clst_region_segments(struct file *file, size_t size, loff_t *offset,
                                   int retval, loff_t segment[]);

size_t pCFSm_clst_shipFrom(uint64_t dinode, const char *buf, size_t size,
                              loff_t *offset, int owner);

size_t pCFSm_clst_shipTo(uint64_t dinode, const char __user *buf, size_t size,
                            loff_t *offset, int owner);

int pCFSm_is_ginode_pCFS(uint64_t dinode);
```

Most of the names are self explanatory; however some do deserve further description:

`pCFSm_clst_region_vrfy()` checks a pCFS `dinode` for the existence of a region which will cover a read (`rw == 0`) or write (`rw == 1`) access starting at `offset` and with a length of `len`.

`pCFSm_clst_region_segments()` is called after `_region_vrfy()` to compute the data ranges that must be handled by the local node and/or its left and/or right neighbours.

`pCFSm_clst_shipFrom()` retrieves from the `owner` node an amount `size` of data stored in the file specified by `dinode`, starting at `offset` and move it to the user's buffer pointed to by `buf`.

`pCFSm_clst_shipTo()` forwards to the `owner` node an amount `size` of data stored in the user's buffer pointed to by `buf`; the data will be written in the file specified by `dinode`, starting at `offset`.

`pCFSm_is_ginode_pCFS()` checks if `dinode` represents a pCFS file (i.e., a file that was opened with a pCFS option flag).

21.4 The pCFSc module interface

The pCFSc module doesn't export functions to GFS; it handles messages from other nodes (in the current prototype forwarded via pCFSd) to perform cache coherency invalidations and data shipping operations.

21.5 The pCFSd daemon architecture

The overall architecture and major processing steps performed by the pCFSd daemon are depicted below: after establishing connections with all nodes, it's up to each thread to communicate with its partnering pCFSk through its TCP channel (`genCltSkt` array); when required, a thread may send invalidation requests to pCFSc modules (through TCP channels in the `invCltSkt` array) in selected target nodes. Access to global shared data is infrequent, and serialised through a single mutex.

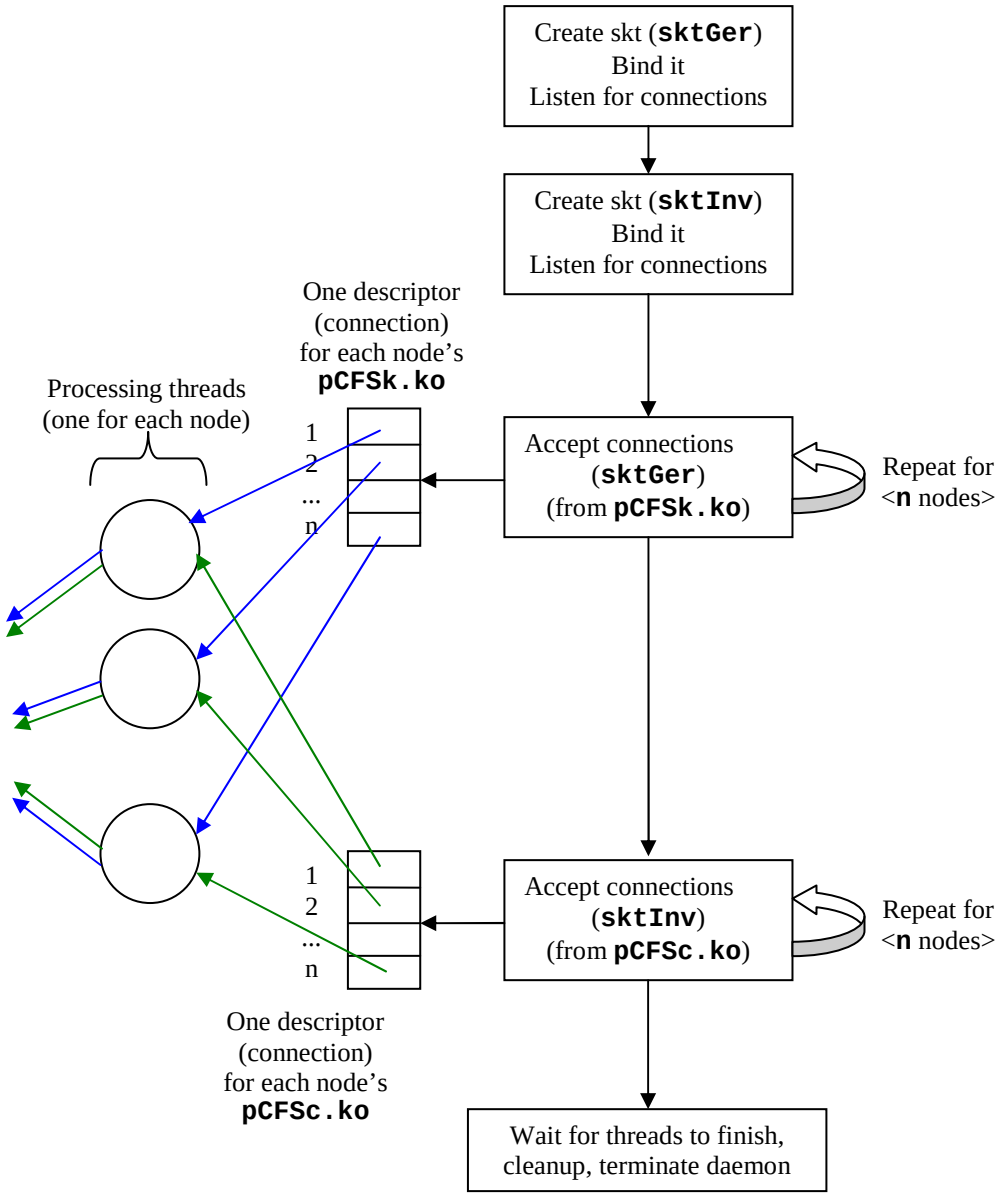


Figure 21.1 pCFSd daemon architecture

21.6 Selected examples of interaction among pCFS' components

For each function we now briefly introduce some information on the interaction between pCFSk, pCFSc and pCFSd; we also flag any inconsistencies with POSIX on our error returns. For more details about the wire protocol (tags, packet structures, etc.) see section 22, further down.

21.6.1 Opening a pCFS file

```
int pCFSk_clst_open(uint64_t dinode, unsigned int o_flags);
```

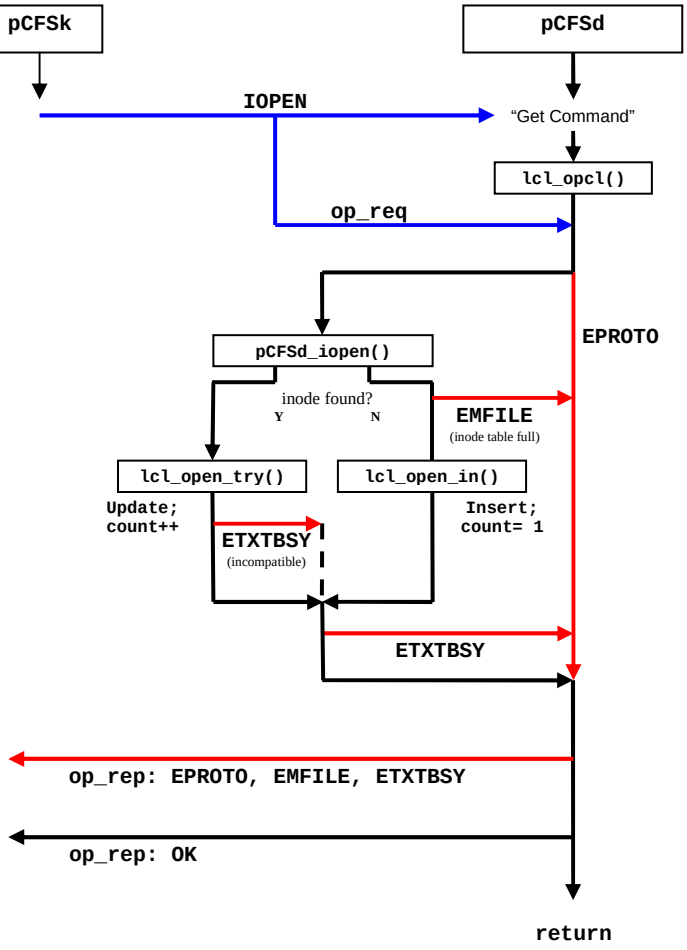


Figure 21.2 Opening a pCFS file

Processing overview: In the node issuing the `open()`, a check is performed at the `pCFS_opens` table to see if the file was already open (found a matching dinode id in the table); if found, verify that the requested open is compatible (check `o_flags` against the stored mode), else drop out with `ETXTBSY`. If this is the first open, check for available space in the table; if full, return with `EMFILE`. If no error has occurred, forward the request to pCFSd and await a reply. At the daemon, the request is processed, and its global `pCFS_opens` table is searched for a dinode id match; if none found, a new entry is created (if the table is full, `ENFILE`); if a compatible entry is found processing continues, else, `ETXTBSY` is returned to the client. If no error has occurred, a reply containing the current owner and sharer bitmap sets is sent back to the node. Back at the node's pCFSk, if an error has occurred return it

to the user, therefore denying the `open()` else update the local entry and return zero. **Note:** To flag an incompatible open, we have resorted to `ETXTBSY`, which is used in standard POSIX to signal an attempt to remove an executable file while it is being executed; and we resorted to `ERPROTO` to indicate either communication protocol errors or inconsistencies between the daemon and the kernel modules (which indicate bugs, as they should not arise).

21.6.2 Insert a region in a pCFS file

```
int pCFSk_clst_region_in(uint64_t dinode, loff_t start, loff_t end,
                         unsigned int flags);
```

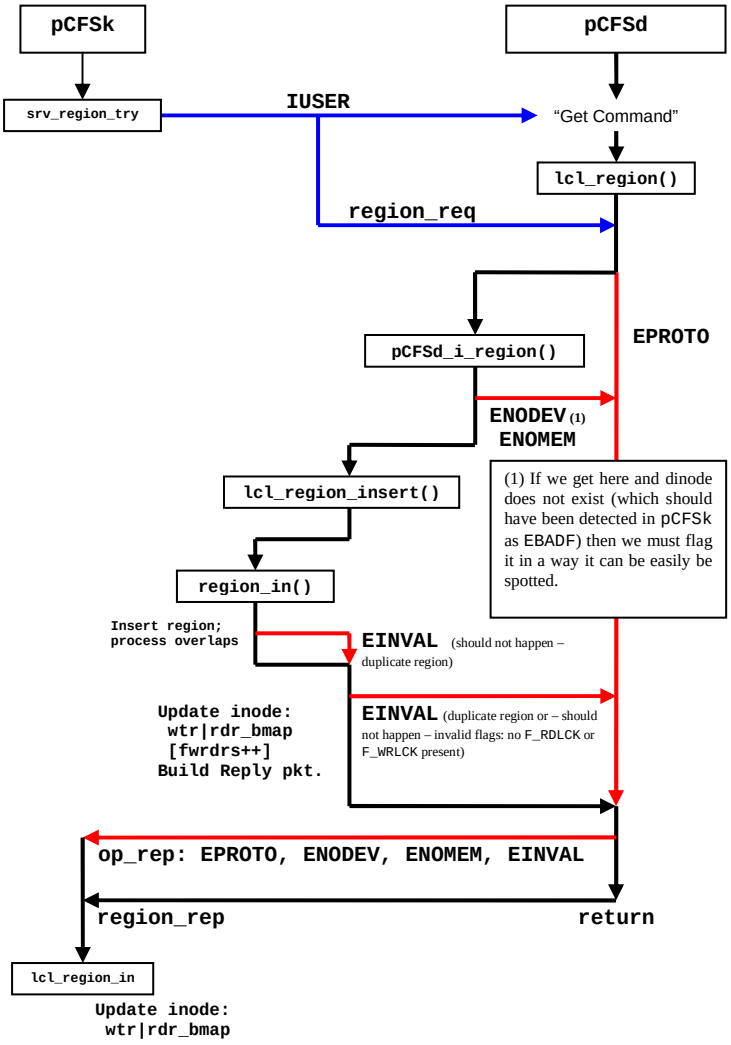


Figure 21.3 Insert a region in a pCFS file

Processing overview: In the node issuing the `fcntl()`, (1) check for the existence of the dinode entry in the `pCFS_opens` table, else – this should never happen – bail out with `ENODEV`. (2) Search the appropriate list (`rdr_regions` or `wtr_regions`, depending on the flags) to locate the place to insert the new region. Bail out if inappropriate flags returning `EINVAL`, or if incompatible with current regions, pend or return `EAGAIN`. Else, forward the request to pCFSd, and pend, awaiting a reply. At the pCFSd daemon, perform (1) and (2) as above; if there's an error, report it back to the pCFSk client;

else, insert the region structure and report back the current file owner and sharer bitmaps. Back at the node's pCFSk, if an error has occurred return it to the user, else update the local entry with the updated information provided by the daemon and return. **Note:** As previously, we resorted to `ERPROTO` to indicate communication errors and inconsistencies between the daemon and the kernel modules (which indicate bugs, as they should not arise); however, we decided that if an attempt to insert a region in a non-open file has reached this level (it should have been detected at VFS or GFS layers), it should be reported as `ENODEV`, used in POSIX to signal an attempt to access an inexistent device.

21.6.3 Remove a region from a pCFS file

```
int pCFSk_clst_region_rm(uint64_t dinode, loff_t start, loff_t end,
                         unsigned int flags);
```

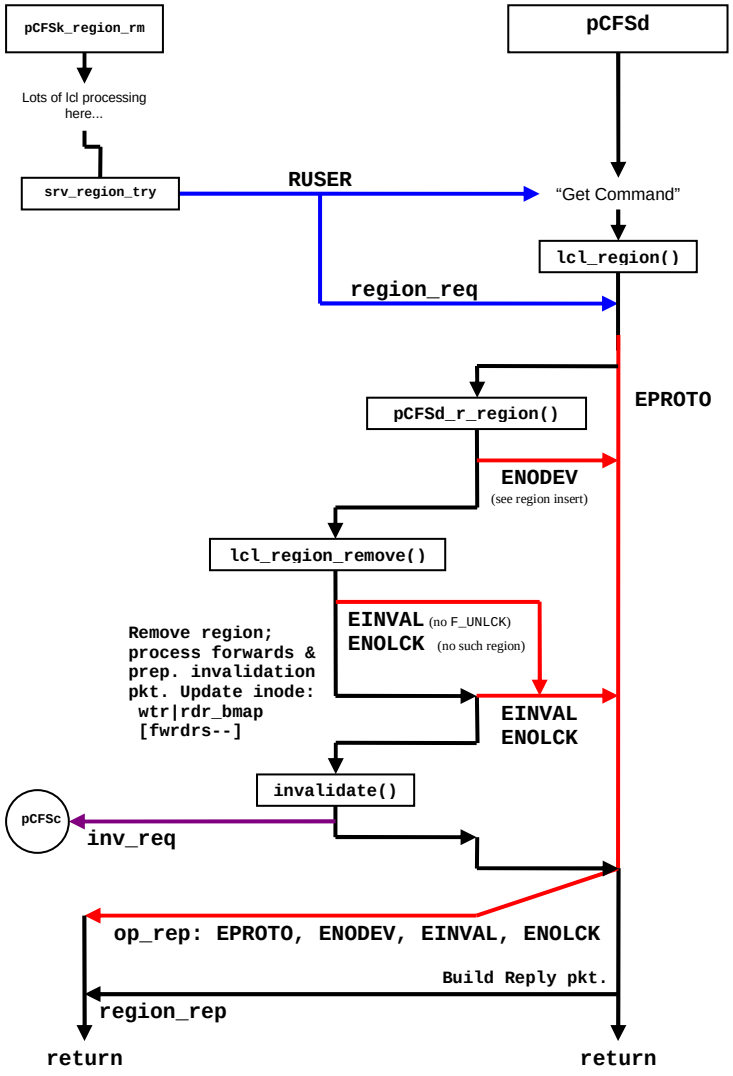


Figure 21.4 Remove a region from a pCFS file

Processing overview: In the node issuing the `fcntl()`, (1) check for the existence of the dinode entry in the `pCFS_opens` table, else – this should never happen – bail out with `ENODEV`. (2) Search the appropriate list (`rdr_regions` or `wtr_regions`, depending on the flags) to locate the place to remove the new region, looking for a full match with (3) `{pid, start, end}`. If a region was not

found, return ENOLCK; if inadequate flags were used, return EINVAL. Otherwise, flush out any data to disk, forward the request to the pCFSd and pend, awaiting the reply. At the daemon, perform (1) and (2) as above but, for lookup, use {node, pid, start, end}. If not found, a consistency error is logged and reported back; else, remove the region structure and prepare a successful return packet, together with the updated owner and sharer sets. If the client pCFSk has told us (pCFSd) that data has been modified, we send inv_req messages to all sharers to invalidate any bytes in this region they may have cached with read-ahead. Back at the node's pCFSk, if an error (other than inconsistency) has occurred it is reported to the user; inconsistency errors are logged, but normal processing continues: the region structure is removed. A return code signals what the closing process was: zero, a reader; one, a writer.

21.6.4 Close a pCFS file

```
int pCFSk_cluster_close(in: dinode, node, pid)
```

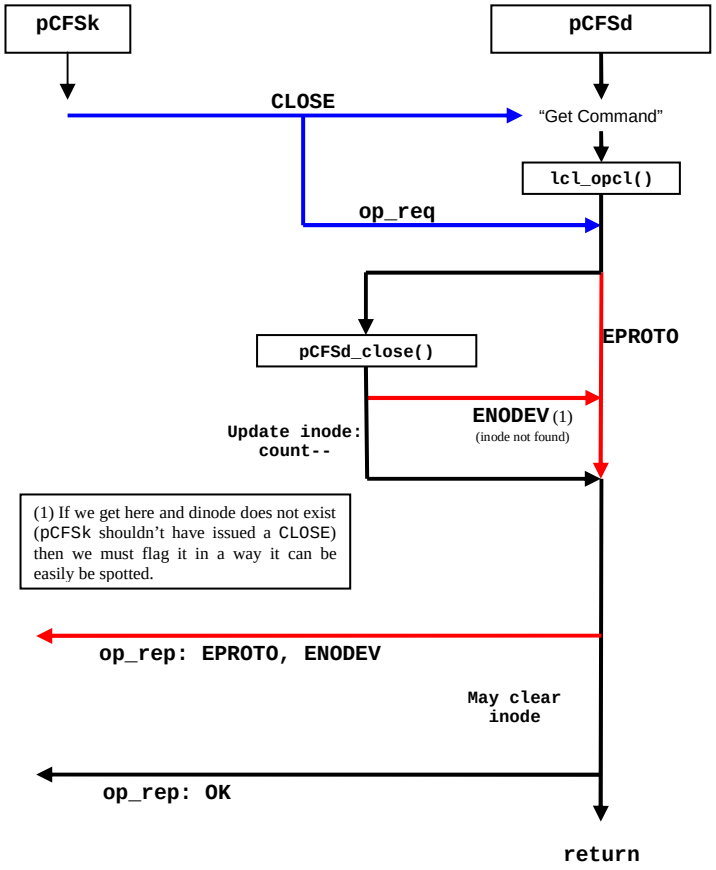


Figure 21.5 Close a pCFS file

Processing overview: In the node attempting to close the file, (1) find the entry for dinode in the pCFS_opens table (if not found return with ENODEV) and (2) assert that its region lists do not contain entries for this process. If (2) fails, report a "must remove regions before closing" error (currently not implemented; we will opt for the standard approach of automatically remove all regions for that process when closing a file). If ok, the close request is forwarded to the pCFSd and we pend, awaiting the reply. At the daemon, perform (1) and (2) as above; if failed return the appropriate error

reply packet. If this wasn't the last user of the file, the current owner and sharer sets are reported back to the issuing node's pCFSk; else, the entry is removed from the daemon's global `pCFS_opens` table. Back at the node's pCFSk, if errors have occurred, they are logged, but normal closing continues; if this was the file's last user either in the node or clusterwide, the entry is removed from the node's local `pCFS_opens` table.

21.6.5 Shipping data to/from an owner node

```
size_t pCFSk_clst_shipTo(uint64_t dinode, const char *buf,
                         size_t size, loff_t *offset, node_t owner);

size_t pCFSk_clst_shipFrom(uint64_t dinode, char *buf, size_t size,
                           loff_t *offset, node_t owner);
```

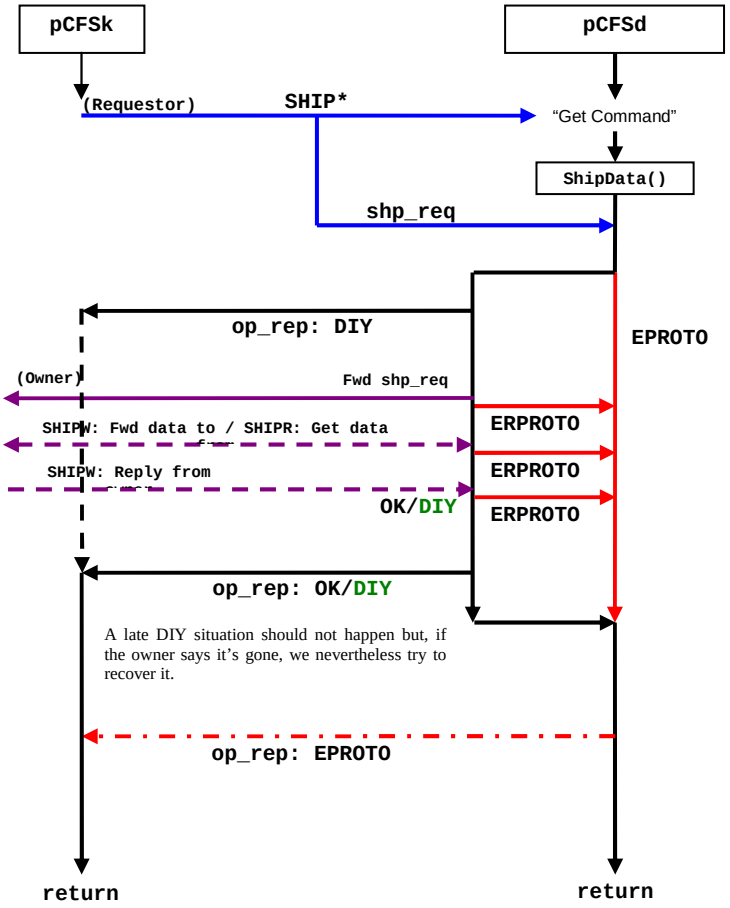


Figure 21.6 Shipping data to/from an owner node

Processing overview: (1a) In the requestor node, build a packet specifying we want to ship size bytes to/from node owner, to be stored at/retrieved from offset, and send it to intermediary pCFSd. (2a) At the daemon, the global `pCFS_opens` table is searched for a dinode match; if not found, an error is logged and ENODEV is returned; if found but we know that owner "is gone", `DIY` is returned; else, `OK` is returned. (1b) Again at the node's pCFSk, if `ENODEV` return it. If `DIY` return 0 (subsequent code at the pCFSm layer will perform the write/read locally). For the `OK` return, build a second packet, this

time a data packet and send it to pCFSd. (2b) At the daemon, send the packet to its final destination: the pCFSc module in the node `owner`. (3) The pCFSc module at the node: a) for a `SHIPW` receives the data from the pCFSd and merge it into the node's Page Cache, linking it to the inode's dirty list; or, b) for a `SHIPR` retrieves the data from the file (using VFS functions that either get it from the cache or force a disk read), packs it into a data packet, and sends it to the intermediary pCFSd. (2c) Pass the reply, `OK` or `DIY`, back to the pCFSk requestor, which will either return the number of bytes processed, or zero, to flag the pCFSm layer to process the request locally.

21.6.6 Closing remarks

Among the details we've chosen to omit, we include communication and protocol errors: currently, no attempt is made to recover from communication errors – if a TCP connection aborts, for some reason, we do not try no re-open it; and any protocol error among two parties, e.g., pCFSk and pCFSd, is flagged with `EPROTO` and may be propagated up to the user – but processing may continue, in some cases. Finally, the total amount of code (comments and blank lines included) for this set of independent modules is 4470 lines (coded in C); its per-module breakdown is reported below:

include files			pCFSc	pCFSk	pCFSd
user-level	kernel-level	common (to user & kernel)	707	1430	1848
N.A.	229	256			

Table 21.1 Lines of code breakdown for each module

22 The pCFS wire protocol

22.1 Introduction

The wire protocol refers to data formats used in "conversations" among pCFSk, pCFSc, and pCFSd. It is quite simple and includes a set of one-byte commands that are used to tag packets, and three request packet structures: one for inode operations, another for region operations, and a third one for coherency and shipping operations. Two reply packet formats are used: one for region operations and another for every other case.

22.2 Wire protocol for pCFS inode table management

The "operation request" structure, `op_req`, is used in requests sent from pCFSk to pCFSd for inode operations; although there are only two functions in the pCFSk interface, one for the open and another for the close, at the wire protocol there are two separate "open operations": `IOPEN`, for the very first open, and `UOPEN`, for subsequent opens of the same file (inode).

The structure of the operation request packet, `op_req` is

```
struct op_req {
  char          cmd;
  uint64_t      dinode;
  unsigned int  mode;
  unsigned int  node;
};
```

and the valid “commands” (tags) are

```
#define IOPEN  'I'
#define UOPEN  'U'
#define CLOSE  'C'
```

where

`cmd`	Tags the packet for open, update or close, as defined above.
`dinode`	Identifies the on-disk (and in-core, as they are the same) file inode.
`mode`	Reserved (currently unused).
`node`	If used, serves only for “double-checking” purposes as the TCP stream already identifies the intervening node(s).

The operation reply packet structure is

```
struct op_rep {
  char          cmd;
  uint64_t      dinode;
  int           mode;
  node_t        owner;
  unsigned long rdr_bmap;
  unsigned long wtr_bmap;
};
```

where

`cmd`	Tags the packet as a reply for an open, update or close.
`dinode`	Identifies the on-disk (and in-core, as they are the same) file inode.
`mode`	Zero for “no error, acknowledge”; positive when information is being returned; negative for error codes.
`owner`	If non-zero, carries the id of the inode’s owner.
`rdr_bmap`	Carries the bit map of the node ids of read sharers for the file.
`wtr_bmap`	Carries the bit map of the node ids of writer sharers for the file.

In the reply, `cmd` and `dinode` fields are used for double checking, only. In the prototype every interaction (message-reply) is synchronously run to completion, and cannot be overlapped with other messages, so there is no need, strictly speaking, of a `cmd` and `dinode` fields in the reply packet, as there is no need for sequence numbers.

22.3 Wire protocol for region management

The region request structure, `region_req`, is used in requests sent from pCFSk kernel modules to the pCFSd daemon for region insertion and removal, while the `region_rep` structure is used in replies sent back from pCFSd to pCFSk. The structure of the region request packet, `region_req` is

```
struct region_req {
  char                cmd;
  uint64_t            dinode;
  struct pCFS_region  region;
};
```

And the `pCFS_region` structure is

```
struct pCFS_region {
  loff_t        start;
  loff_t        end;
  node_t        node;
  pid_t         pid;
  node_t        ownerL;
  node_t        ownerR;
  unsigned int  flags;
};
```

Valid commands are (where “user” is a synonym for region):

```
#define IUSER  'i'
#define RUSER  'r'
```

where

`cmd`	Tags the packet for insert or removal, as defined above.
`dinode`	Identifies the on-disk (and in-core, as they are the same) file inode.
`start`	Byte offset where the region starts.
`end`	Byte offset where the region ends.
`node`	If used, serves only for “double-checking” purposes as the TCP stream already identifies the intervening node(s).
`pid`	The pid requesting the region.
`ownerL`	Not used in requests.
`ownerR`	Not used in requests.
`flags`	The flags argument in the user `fcntl()` call.

The reply packet structure for region operations is

```
struct region_rep {
  char          cmd;
  uint64_t      dinode;
  node_t        ownerL;
  node_t        ownerR;
  unsigned long rdr_bmap;
  unsigned long wtr_bmap;
};
```

where

`cmd`	Tags the packet as a reply for a region insert or removal.
`dinode`	Identifies the on-disk (and in-core, as they are the same) file inode.
`ownerL`	If non-zero identifies a left owner for the first page of the region inserted.
`ownerR`	If non-zero identifies a right owner for the last page of the region inserted.
`rdr_bmap`	Carries the bit map of the node ids of read sharers for the file.
`wtr_bmap`	Carries the bit map of the node ids of writer sharers for the file.

22.4 Wire protocol for coherency management and data shipping

The `cc_req` request packet may be used for two different purposes: to send invalidation requests from the pCFSd daemon to a subset of nodes via their pCFSc kernel modules; and to perform data shipment operations, i.e., moving data from one node to another.

The `cc_req` structure is

```
struct cc_req {
  char          cmd;
  uint64_t      dinode;
  node_t        node;
  loff_t        start;
  loff_t        end;
};
```

Valid commands are:

```
#define INVPG  'X'
#define SHIPR  'R'
#define SHIPW  'W'
```

where

`cmd`	Tags the packet for cache invalidation or data shipping, as defined above.
`dinode`	Identifies the on-disk (and in-core, as they are the same) file inode.
`node`	Used only in data shipping operations to identify the target node.
`start`	Byte offset where the operation starts.
`end`	For invalidations, byte offset where the operation ends; for shipping operations, amount of data to be shipped.

The structure for the reply packet for cache and data shipping operations is the same `op_rep` structure used for pCFSk/pCFSd interaction, already described in 22.2.

23 pCFS changes to GFS code

23.1 Introduction

We now list some GFS functions and the modifications we have introduced to implement the pCFS behaviour, using some of the pCFSm functions previously described. The list is, obviously, not complete; it includes a subset we believe is relevant to give the reader a better understanding of the prototype implementation. And keeping that in mind, we've chosen to present them in a particular order, starting with the `gfs_write()`.

When reading the code, one should never forget how it enters execution: as the user calls, e.g., a `write()`, the flow of execution enters the kernel in `sys_write()`, then flows through the VFS layer code until it reaches GFS, in this case in the `gfs_write()`.

23.2 Selected code fragments

23.2.1 Writing to a pCFS file

The original `gfs_write()` function is very simple, just

```
/**
 * gfs_write - Write bytes to a file
 * @file: The file to write to                    @buf: The buffer to copy data from
 * @size: The amount of data to write             @offset: The current file offset
 *
 * Outputs: Offset - updated according to number of bytes written
 *
 * Returns: The number of bytes written, updates offset; errno on failure
 */
static
  ssize_t gfs_write(struct file *file, const char *buf, size_t size, loff_t *offset)
{
     return(__gfs_read(file, buf, size, offset, NULL));
}
```

This is, indeed, a very simple piece of code and does not even allow us to show one of the major changes of pCFS, namely the one were the exclusive lock on the inode is replaced by a shared one, as pointed out in sections 19.4.3 and 20.3.3.4. In fact, that particular change is buried very deep into GFS code. But, as we will see below, this simple function has, nevertheless been extensively changed… Modifications to support coherent writes across write shared “frontier” pages (see Fig. 20.2) – which, fortunately, also provide us with a simple way to support data shipping – turn `gfs_write()` into a more complex function:

```
static
  ssize_t gfs_write(struct file *file, const char *buf, size_t size, loff_t *offset)
{
     struct inode *inode = file->f_mapping->host;
     struct gfs_inode *ip = get_v2ip(inode);
     uint64_t dinode;

     loff_t segment[3]= {0,0,0};
     int ownerL, ownerR, retval, retcode; int skew = 0;

   /* Take the normal GFS path */                                              (1)
     if ( !IS_FILE_PCFS(file) )
            return(__gfs_write(file, buf, size, offset, NULL));

   /* Downwards for pCFS file with region locks or in D-S mode */              (2)
     dinode= (&ip->i_num)->no_formal_ino;
     retval= pCFSm_clst_region_vrfy(FLOCK_VERIFY_WRITE,
                        dinode, *offset, (loff_t)size, &ownerL, &ownerR);

     if (retval < 0) return retval;

   /* If we don’t have neighbours, process it locally */
     if (!retval)                                                            (3)
            return(__gfs_write(file, buf, size, offset, NULL));

   /* We are D-S or have neighbours */
     if ( retval == O_DATA_SHIP )
       segment[0]= size;
     else
       pCFSm_clst_region_segments(file, size, offset, retval, segment);      (4)
```

```
   /* Left Owner? failure, try recovery through local GFS write */
     if (segment[0]) {                                                          (5)
          retcode= pCFSm_clst_shipTo(dinode, (const char __user *)buf,
                                     segment[0], offset, ownerL);
             if (retcode != segment[0])
                    segment[1]+= segment[0];
             else {
                    skew= segment[0];
                    *offset += segment[0];
             }
     }

   /* Local write? If failure, try recovery through local (GFS) write */
     if (segment[1]) {                                                          (6)
             retcode= __gfs_write(file, buf+skew, segment[1], offset, NULL);
             if (retcode != segment[1]) {
                    PCFS_INFO("Failure in __gfs_write");
                    return retcode;
             }
             skew+= segment[1];
     }

   /* Right Owner? If failure, try recovery through local (GFS) write*/
     if (segment[2]) {                                                          (7)
        retcode= pCFSm_clst_shipTo(dinode, (const char __user *)buf+skew,
                                   segment[2], offset, ownerR);
        if (retcode != segment[2]) {
            retcode= __gfs_write(file, buf+skew, segment[2], offset, NULL);
            if (retcode != segment[2]) {
               PCFS_INFO("Failure in __gfs_write recovery");
               return retcode;
            }
        }
     }
     return size;
}
```

Comments to the modified `gfs_write()` code:

(1) The overhead of the modifications to the GFS regular write is, as intended, negligible: it costs a few variable assignments and the evaluation of the if statement and its macro, which accesses local variables.

(2) For pCFS files, we check with `pCFSm_clst_region_vrfy()` that the write was executed in data shipping mode or within a valid region. We get a zero or positive return: zero indicates we have no neighbours owning pages that we want to access; we get a 1 if there is a owner for the leftmost (lowest index) page in our region, and a 2 if there is a owner for the rightmost (highest index) page in our region; finally, we get a 3, if we have both left and right neighbours owning “our” frontier pages. This function is executed against purely local data – it does not access the daemon.

(3) If we have no neighbour owners, we perform the local, GFS regular write.

(4) When we have neighbours, `pCFSm_clst_region_segments()` – again, executed against local data – is used to break up the size into a maximum of three portions: one to be shipped to a left owner, another to be handled by the local node, and the remaining to be shipped to a right owner (of course any – but not all – of the above mentioned portions may be zero). The writes will be handled at (5), (6) and (7), below. An attempt is made to recover any failed shipping with a local write.

(5) We ship `segment[0]` bytes directly from the user buffer to the left neighbour, thus skipping the page cache *in the local node* (in the owner node, data is injected into its page cache); if successfully, we update the pointer to the user buffer; else, we try to recover by adding the amount of data we should have written to the next segment's duties.

(6) We take `segment[1]` bytes from the user buffer and perform a local write; upon failure, we return the error to the user.

(7) We ship `segment[2]` bytes from the user buffer (again, skipping the page cache) to the right neighbour; upon failure, we try to recover with a local write and, if we fail again, we return the error to the user.

23.2.2 Reading from a pCFS file

The original `gfs_read()` function ("header" comments removed) is also very simple:

```
static ssize_t gfs_read(struct file *file, char *buf, size_t size, loff_t *offset)
{
     return(__gfs_read(file, buf, size, offset, NULL));
}
```

Interaction among writers and readers, even when they share non-overlapping portions of the same page is guaranteed by the invalidation mechanism, as explained in 21.3.1; so, we should not need to change the `gfs_read()` function. Change is, in fact, required, but not to support interactions among readers and writers; it is necessary to support sharing among neighbour writers, as the solution adopted for the `gfs_write()` above skips the local node's page caches for file segments that are shipped. Therefore, to support reading of up-to-date data in these frontier segments, a node may have to request it "back" from the owner.

The majority of the code is quite similar to the one in `gfs_write()` and could be obtained just replacing calls to write with calls to read; we choose not to duplicate it here, but instead focus on one important difference: a read can take place against a read (`F_RDLCK`) or write (`F_WRLCK`) region, so we have to check for both. The (rather) stripped down code is:

```
static ssize_t gfs_read(struct file *file, char *buf, size_t size, loff_t *offset)
{
     ...
     int rw= FLOCK_VERIFY_READ;


     /* Take the normal GFS path */
        if ( !IS_FILE_PCFS(file) )
          return(__gfs_read(file, buf, size, offset, NULL));

     /* Downwards for pCFS file with region locks or in D-S mode */
        dinode= (&ip->i_num)->no_formal_ino;

retry:                                                                    (1)
        retval= pCFSm_clst_region_vrfy(rw, dinode, *offset, (loff_t)size,
                                       &ownerL, &ownerR);
        if (retval == -ENODEV) {
          retval= 0;
          PCFS_ERROR("pCFSm_clst_region_vrfy: dinode not found");
        } else if (retval == -ENOLCK) {
```

```
          if (rw == FLOCK_VERIFY_WRITE) {
            return retval;
          } else {
            rw= FLOCK_VERIFY_WRITE;
            goto retry;
          }
        }

   /* If we don't have neighbours, process it locally */
        if (!retval)
          return(__gfs_read(file, buf, size, offset, NULL));

   /* We have neighbours */
        pCFSm_clst_region_segments(file, size, offset, retval, segment);

        skew= 0;

   /* Left Owner? If failure, try recovery through local (GFS) read */
        if (segment[0]) {
          ...
        }

   /* Local read? If failure, try recovery through local (GFS) read */
        if (segment[1]) {
          ...
        }

   /* Right Owner? If failure, try recovery through local (GFS) read*/
        if (segment[2]) {
          ...
        }

        return size;
}
```

The only comment to the modified `gfs_read()` code fragment above is that the "retry loop" is executed as follows: to verify the read against a valid region, we first assume that a read region has been laid out and, therefore, execute a `pCFSm_clst_region_vrfy` with a `FLOCK_VERIFY_READ` search option; if we don't find a matching region, we "upgrade" our option to `FLOCK_VERIFY_WRITE` and retry the search; only a second failure will lead to the conclusion that no valid region exists and the read must be aborted.

23.2.3 Removing a region from a pCFS file

As described before (see sections 21.3.3.2, 21.3.3.5, 22.6.2), pCFS regions can be laid out and removed using the POSIX lock options of `fcntl()`. As it happens with other user calls, `fcntl()`l drops through `sys_fcntl()` and, along the way, executes the GFS function `gfs_lock()`, sketched below:

```
static int gfs_lock(struct file *file, int cmd, struct file_lock *fl)
{
      struct gfs_inode *ip = get_v2ip(file->f_mapping->host);
      struct gfs_sbd *sdp = ip->i_sbd;
      struct lm_lockname name = { .ln_number = ip->i_num.no_formal_ino,
                                  .ln_type = LM_TYPE_PLOCK };

   /* pCFS begin */
      struct gfs_glock *gl = ip->i_gl;
      struct gfs_glock_operations *glops = gl->gl_ops;
      int retcode;
   /* pCFS end */

   /* Check for conflicts on local node and possibly wait */
      ...
```

```
    if (!IS_FILE_PCFS(file)) {    (1)
            if (IS_GETLK(cmd))
                    return gfs_lm_plock_get(sdp, &name, file, fl);
            else if (fl->fl_type == F_UNLCK)
                    return gfs_lm_punlock(sdp, &name, file, fl);
            else
                    return gfs_lm_plock(sdp, &name, file, cmd, fl);
    }

  /* pCFS begin */
    if (IS_GETLK(cmd))                                                    (2)
        return gfs_lm_plock_get(sdp, &name, file, fl);
    else if (fl->fl_type == F_UNLCK) {                                    (3)
        retcode= gfs_lm_punlock(sdp, &name, file, fl);
        if (!retcode) {
            retcode=pCFSm_clst_region_rm(ip->i_num.no_formal_ino,
                      fl->fl_start, fl->fl_end, (unsigned int) fl->fl_type);
            if (glops->go_sync) && (retcode == WTR)                       (4)
                glops->go_sync(gl, DIO_DATA);
            else
                PCFS_ERROR("REGION Syncing, but no glops->go_sync");
        }
    } else {
        retcode= gfs_lm_plock(sdp, &name, file, cmd, fl);                 (5)
        if (!retcode) {
            retcode=pCFSm_clst_region_in(ip->i_num.no_formal_ino,
                                         fl->fl_start, fl->fl_end,
                                         (unsigned int) fl->fl_type);
            }
    }
    return retcode;
  /* pCFS end */
}
```

Comments to the modified `gfs_lock()` code:

(1) The original GFS code is bounded within this if, for non-pCFS files.

(2) For pCFS files, we check for a “get region” command, flagged with `F_GETLK`, using the standard GFS code, as in (1).

(3) When removing regions from pCFS files, after the `gfs_lm_punlock()` we trigger `pCFSm_clst_region_rm()` to a) remove the region from local and global pCFS “databases” and b) send invalidation messages to other nodes.

(4) Then, we force a flush of the file (inode); this is the final step to guarantee consistency with other nodes: as they access bytes within this (removed) region, they will be forced to get them from disk (or from owner nodes that get them from disk). Notes: a) the `go_sync()` is, for regular files, mapped to `inode_go_sync()` and tests if the inode is dirty and, after flushing, clears the dirty flag; b) this version was not tested against metadata changes (and does not flush them).

(5) Similarly, when inserting regions into pCFS files we start by using the regular GFS function, i.e., `gfs_lm_plock()`, and then `pCFSm_clst_region_in()`, to insert the region into local and global pCFS “databases”.

23.2.4 Closing remarks

The amount of code which has been added (and/or modified) to the GFS "main" module is quite small; the current version, which has a fair amount of lines used for debugging and/or are commented out waiting for its inclusion in newer revisions, has an excess of 470 lines when compared with GFS' original sources. The breakdown is as follows:

	Original GFS version	pCFS modified version
Number of code files (.c)	39	Unchanged
Number of include files (.h)	42	+ 1
Total number of lines (.c)	33425	+ 410
Total number of lines (.h)	5523	+ 60

Table 23.1 Breakdown of the pCFS changes to GFS code

Part VIII:

Benchmarking pCFS

In this Part we benchmark pCFS against “plain” GFS and other well know file systems such as NFS and PVFS (where both the “regular” configuration, with internal disks, and the highly available configuration, with disk volumes provided by an external disk array, were benchmarked); these benchmarks go beyond the usual set of bandwidth metrics and also account for CPU consumption.

24 Characterising the infrastructure

24.1 The test bed infrastructure

The infrastructure used for the tests was already portrayed in Fig. 16.2 and is reintroduced again for ease of reference.

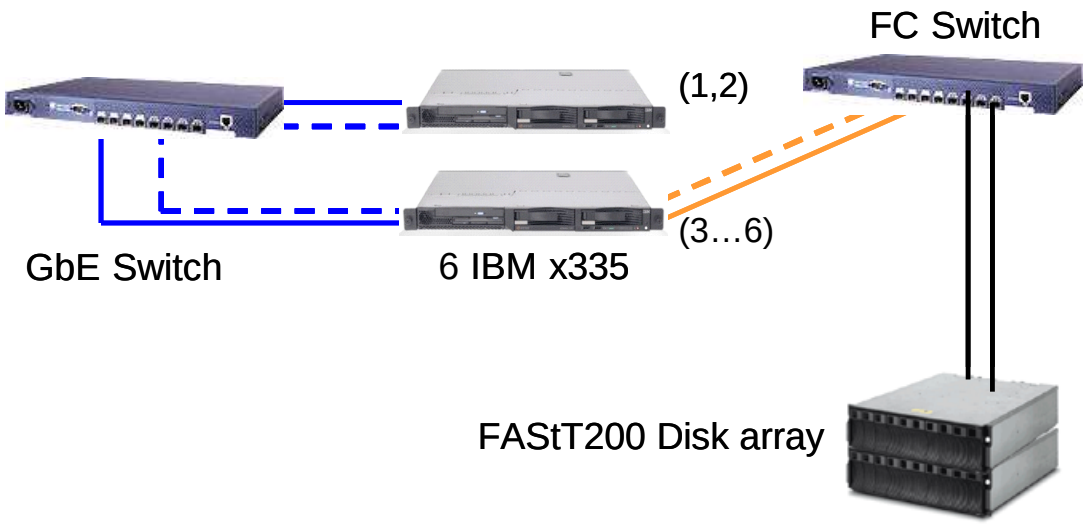


Figure 24.1 Test bed infrastructure

The infrastructure was completely isolated and dedicated to testing; nothing else was running in the nodes except for the Munin [Munin] data collection agents (`munin-node`) which were configured to gather just the information needed for the reports; each node was polled once every minute, and reporting/graphing was carried out in another node, not represented in the figure, so the load introduced was negligible. When reading the CPU usage graphs, these are 2-CPU nodes with hyper-threading on, therefore Linux counts 4 CPUs per node; thus, if the value reported for, say, "system time" is 20% it should be adjusted to 10%.

24.2 Networking: the LAN infrastructure

Network testing focused on determining the highest bandwidth available from the hosts' integrated Broadcom 5703 NICs, and checking if the SMC 8624T Gigabit Ethernet switch would be able to support all ongoing TCP streams without undue contention; tests were carried out with the `netperf` network performance benchmark[1] as follows:

- We configured each even numbered node as a server, and each odd numbered node as a client.
- Each client's bandwidth was separately measured; then, it was again measured while other clients were also concurrently accessing their servers.
- Each test ran for 10 minutes, and was repeated three times. Message size was 16 KB (the Linux version of `netperf` does not allow this parameter to be changed).

[1] http://www.netperf.org

The set of figures below is self explanative, but we nevertheless add a few comments: first, and foremost, Munin-reported results are within 5% of the values reported by `netperf` (we used `netperf -c -C -l 600 -H` *hostname*) so we decided to include Munin graphs and dispense the `netperf` output.

In summary, we have, for the "slow" (2.66 GHz) nodes a TCP bandwidth of 975.5 Mb/s, a CPU usage of 37.6% (system: 19.81, softirq: 55.36, after adjustment to 2 CPUs), and a rate of 16.2 k (thousand) interrupts per second issued by the NIC (eth0). For the "fast" (3.06 GHz) nodes (not shown), both the TCP bandwidth, at 975.4 Mb/s, and the interrupt rate, at 16.9 k interrupts per second, are quite similar, the difference being the CPU usage, at 27.9 %.

Figure 24.2 TCP bandwidth testing with netperf

We decided to experiment with the so called Jumbo frames, and we configured the nodes with 9000 bytes of MTU; the results were impressive: Munin reported a bandwidth increase to 999.04 Mb/s, and CPU usage decreased to 11.22% (system: 12.87, irq: 1.18, softirq: 8.38); the largest drop is, clearly, in the softirq usage. Interrupt rate at the controller, as expected, decreased to 11.5 k/s.

To conclude, movement of data across a gigabit interconnect may be fast but quite expensive in CPU: the client alone can consume about 40% in a 2 CPU node; adding both the client and the server will easily double that figure. We note that testing all nodes concurrently showed no degradation introduced by the SMC 8624T Gigabit Ethernet switch at MTU 1500, and a very slight decrease at MTU 9000 (Munin reported 995.92 Mb/s).

24.3 Storage: the FC infrastructure and the disk array

Storage infrastructure testing focused on determining the highest bandwidth available from the FAStT-200 storage array subsystem and, while doing it, assessing if the Brocade Silkworm FC switch would be able to support all six FC streams (6 FC adapters on 4 hosts "connecting" to two FC ports on the disk array) without undue contention; recall that the FC infrastructure uses the lowest rate available, at 1 Gb/s per FC port.

To test the array, it is fundamental to understand its internal architecture; Fig. 24.3 shows the architecture of an entry-level Dell/EMC array, which is quite similar to the FAStT-200 array we're using, a dual-processor configuration.

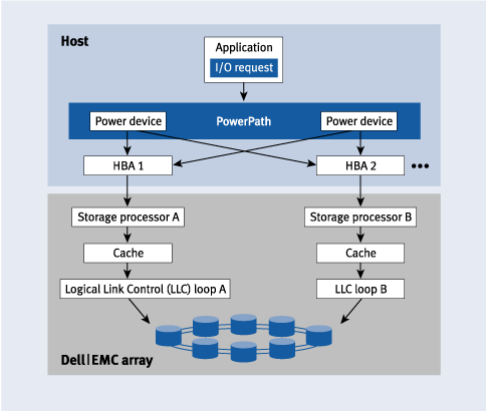


Figure 24.3 Entry level, dual storage processor disk array architecture

In disk arrays, identical physical disks are usually grouped together in a larger virtual RAID volume; in entry-level disk arrays all disks that form a group are owned by a single storage processor (SP) – that is to say, only that processor can issue commands and transfer data to/from those disks (if that SP fails, then the other "takes over" the disk group). Thus when an application issues I/O requests targeting a RAID volume, requests may follow different routes, but they must reach the SP that owns the volume.

Aggregation, at the array level, of disks into a RAID group usually increases bandwidth (BW) in the disks/cache/SP path (a disk is the "weakest link" in the chain, delivering a sustained BW which is clearly below the cache and/or SP's capacity) and results in increased bandwidth to the host. However, the second SP is idle, and cannot be used. A common solution that allows both paths to be used in parallel is to aggregate devices at the host using "storage virtualisation" software such as Linux LVM; as an example, we could aggregate into a larger virtual LUN two RAID groups, one owned by "SP A" and another by "SP B".

For our array we want to assess several configurations, trying to get the best "base level" one to supports the typical HPC environment – large files, often accessed sequentially or in segmented mode (different processes accessing different regions). Tests were carried out with a program we have developed ourselves because widely used file benchmark applications

such as IOzone[2], did not provide the features we needed, such as the ability of using direct I/O on raw devices. Our objectives were to find out:

- The size of the cache for a storage processor (there are two in our FAStT-200) and consequently, its maximum bandwidth – achievable when accessing data cached in the processor.
- If concurrently accessing both storage processors would degrade the above result.
- The sustained bandwidth when reading from a disk (not from cache).
- The CPU usage at the host.

Our application performs as follows: a) it starts by sequentially reading 32 MB from a raw file (e.g., /dev/sdb) opened with `O_DIRECT` to bypass the Linux page cache; b) for each data size, a cache-fill run is executed – and this also touches the page-aligned pages in the user buffer, preparing it for the next page fault free runs; c) the file is re-read with a given "record size" – typically starting at 4 KB and going up to, at least, 1 MB – and each run is separately timed; finally, size increased by 1 MB (or 2 MB for larger file sizes) and the above steps are repeated.

24.3.1 Single storage processor / Single drive tests

The graph below was taken with a run against a single disk drive owned by one storage processor; it shows that although we can read at 75 MB/s with a 16 MB record size, this only happens for data sizes smaller than 46 MB; therefore we conclude that the size array cache seems to be around 45 MB (which is puzzling because the array's product brief quotes a cache size of 88 MB). We also conclude that the sole disk drive used in the test is able to sustain sequential reading at 45 MB/s.

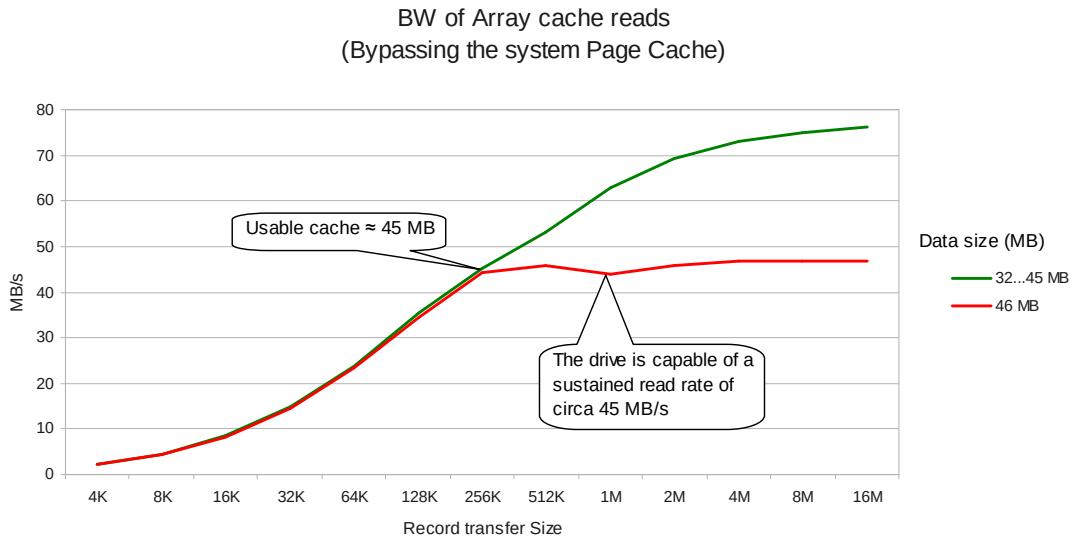


Figure 24.4 Cache size and the sustained read bandwidth (1 processor, 1 drive)

[2] http://www.iozone.org

A set of Munin graphs was taken, and the `iostat` graph, showing the read rate in I/O blocks per second, is now our primary source for information. Notice that the CPU usage is circa 1% (adding system: 1.55, irq: 0.03, and softirq: 0.69, and then adjusting for 2 CPUs); note – `iowait` signals the amount of CPU that was not used because the process was waiting for I/O. Test results show (Fig. 24.5) that a maximum of 50k blocks were read per second and these triggered 440 interrupts per second in the Qlogic FC adapter (QLA-2200F).

Figure 24.5 CPU and interrupt usage, and blocks/s in the array cache read test

24.3.2 Dual drive tests

To try to increase the array's performance, two new configurations were tried: the first one with an array-based RAID-0 volume built from two disks – the volume was then assigned to one of the array storage processors, which was responsible for handling all I/O[3]; and a second configuration, where both SPs were used, each one owning a single drive – and, at the host level, these drives were aggregated into a single RAID-0 volume with the LVM software – therefore creating the opportunity for using both storage processors (and both disks) in parallel, in an attempt to increase the performance.

[3] High-end (expensive) disk arrays do exist where more than one storage processor can issue I/O requests for the drives that make up a RAID volume; we do not know of entry level (inexpensive) disks arrays, such as the FAStT-200, that are capable of doing it.

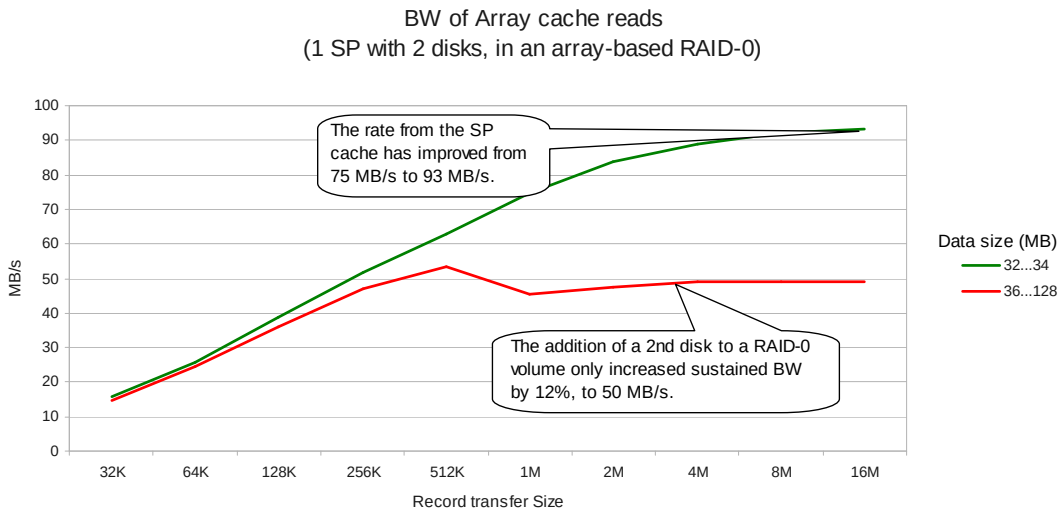


Figure 24.6 Read bandwidth for 1 SP, 2 disks in RAID-0

The first configuration enables us to check the array's ability to aggregate bandwidths of individual disks that make up an array-based RAID volume; the result is quite poor from the perspective of the sequential read test – when compared to the single disk case in Fig. 24.4, the bandwidth increased by a mere 12% to 50 MB/s. However, the I/O rate increased from 50k to 70k blocks/s which is an indication that it may perform better in random read/write testing (graphs not included).

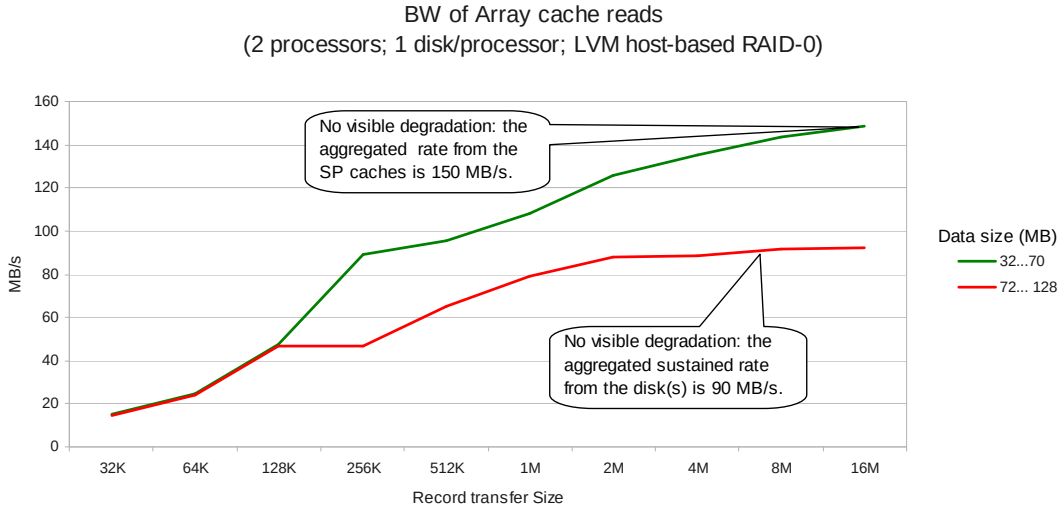


Figure 24.7 Read bandwidth for 2 SPs, 1 disk/SP. LVM stripes them in RAID0

The above graphs confirm that the array does not degrade its bandwidth when using both SPs in parallel; in fact, recalling from Fig. 24.4 that each SP is able to deliver a maximum of 75 MB/s from its cache, and each disk contributes with a sustained bandwidth of 45 MB/s, the array's total is fine at 150 MB/s when reading from its cache(s) and a sustained 90 MB/s when reading from both disks in parallel. Notice that the graphs in Fig 24.8 below show the

CPU usage has increased slightly to 1.6% (system: 2.38, irq: 0.03, softirq: 0.82, halved for 2 CPUs), while a maximum of 50k blocks read per second was reached for each disk drive.

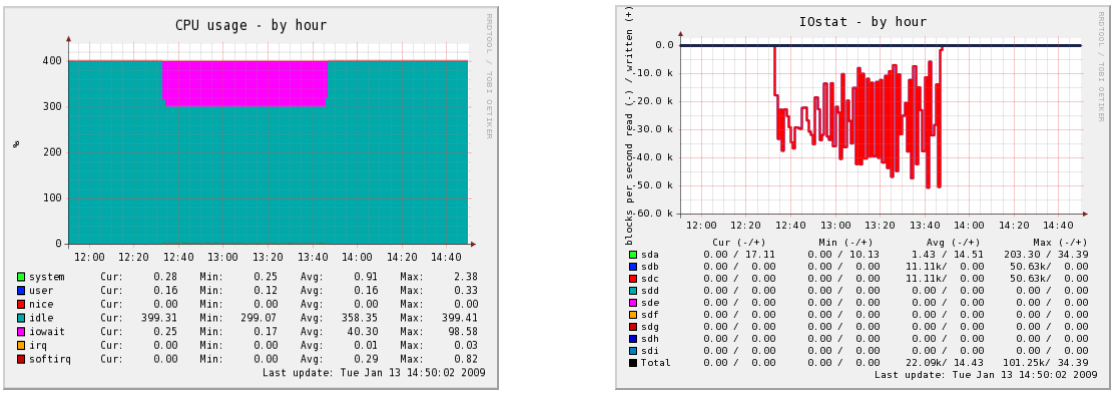


Figure 24.8 CPU usage and I/O statistics for 2 SPs, 1 disk/SP and RAID0 LVM

25 File System testing

25.1 Introduction and rationale

The rationale for the set of tests we will perform is the following: in a very crude statement, this work is about *data* (file) *sharing* among processes accessing “file services” that either run in the node, or in “remote” nodes; and furthermore, these services are geared towards performance, in a HPC-way. Therefore, tests will have to specifically target this environment.

Carrying out short, easily reproducible, and yet meaningful tests is therefore of primary importance; but, regrettably, popular I/O benchmarking applications cannot be used here; as an example, we refer two widely used ones: Bonnie++ and IOzone; the former was designed to test file system performance of single node architectures. However, IOzone can be used on multiple nodes, and furthermore has an option, `-W`, briefly referred in the documentation as “lock files when reading or writing” [Cap+03]; unfortunately, looking at the program’s source code, we found that it uses `fcntl()` calls with arguments to lock/unlock the file as a whole, so it is worthless for us.

Another option is to use real applications; for example, an MPI application such as one we have developed in-house to process tomography images [Cad+08]: it accesses the image file in big, disjoint regions, for reading and writing. However, we cannot use it (yet) for pCFS testing, as usage of MPI over pCFS requires a new ROMIO driver (to cater for pCFS open extensions, etc.). We think that this is probably a small project (if documentation on the ROMIO internals does exist), but not doable within the timeframe of our work.

25.2 The “benchmarking application”

So the only remaining option left was to develop our own benchmarking application, and that’s just what we have done. It is a fairly simple application, composed of a controller and a set of exerciser programs.

The `controller` runs in a node and accepts a string as its sole parameter; the string is a sequence of characters, S and P, which specifies that an exerciser should be fired (along with others) either sequentially (S), or in parallel (P). A few examples are: SSS, where three exercisers will be fired in sequence; SPPS, where a first exerciser will start and, when it finishes, two will be fired in parallel; then, when they are both done, a fourth one will be run.

An exerciser is an I/O program that reads or writes; it accepts as arguments the file size, the buffer size, the total number of exercisers that will be used in the test, and its id number. There are six versions of the exercisers; we’ll just show the reader’s list, the writers being symmetrical to this one:

- `rdr`, a simple reader
- `rdr-lck`, a reader which performs full region locks before it starts reading
- `rdr-sml-lck`, a reader which performs a per-record lock/read/unlock sequence

When an exerciser is started, it registers with the controller and computes the offset where it will start accessing the file (using its id and the file length); then it `lseek()` there and, if that’s the case, locks the region with a standard, *byte-range* `fcntl()` call; finally, it waits for the controller’s command to enter the I/O loop. Upon termination, it reports to the `controller` that its work is done, and waits for the termination command.

This benchmark can be used to exercise a broad range of situations, such as modelling I/O behaviour from parallel applications; for example, when a MPI application performs I/O over NFS, the ROMIO library uses a per-call lock/read/unlock sequence that we can accurately reproduce with the `*-sml-lck` exercisers. We can also, to some extent, simulate multiple file access by streaming over file regions that are very far from each other (the minus is that simulation over a single file does not properly exercise the metadata part: for reading, it may profit too much from metadata caching, while for writing there will be too much locking contention); however, in our tests, we do not try to simulate accesses to multiple files.

25.3 Local file system testing: ext3 performance

We also briefly tested local filesystem performance – namely, ext3 – as this is one of the most utilised local file systems, and the one we’re going to use to support both NFS and PVFS testing. We use a 32 GB ext3 filesystem on top of the best configuration we found from previous tests, i.e., a 64K striped LVM RAID-0 created with 2 physical disk partitions, where each disk was attached to a different storage processor; all tests were run over a 16 GB file.

25.3.1 Single process testing

Our first experiments where conducted with a single process; this setting, particularly when reading, allows us to get a rough figure on the storage system's I/O latency, one that can help us to understand single-process benchmarks that we will perform later on, such as when reading from a single NFS or PVFS client. Sustained performance was tested both for reading and writing, with both buffered access, through the page cache, and direct I/O; we also compared write-though (using the `O_SYNC` flag option on the open) with a `fdatasync()` flush triggered at the end of the write loop. Array-based write caching was disabled.

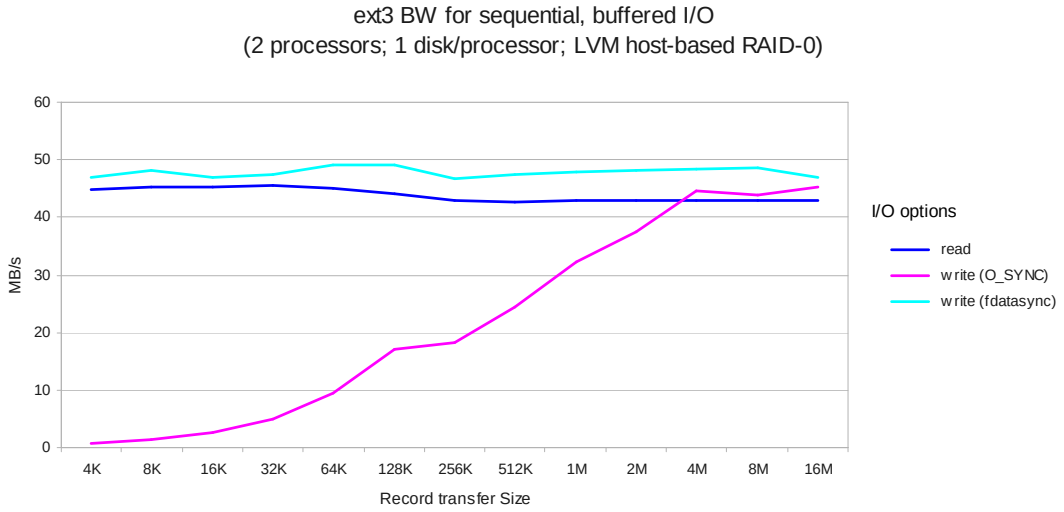


Figure 25.1 Buffered I/O in the ext3 striped volume.

We can clearly see that the strategy Linux devised for buffered reads aims to deliver a smooth performance over a broad range of transfer sizes; this is a result of reading 8 pages (32 KB) for each new request, as configured in `/proc/sys/vm/page_cluster`[4], of read-ahead policies, and of fragmenting large reads; the net result is around 45 MB/s over the whole range. A write-through policy for each `write()` call is definitively too expensive except for very large buffers, and periodic flushing with `fdatasync()` seems a good compromise as it allows for write-combining of several pages[5].

On the other side Fig. 25.2 shows that for direct I/O no optimizations are attempted, so small-sized requests result in very low bandwidths, but very large requests do extract, at 85 MB/s for reads, almost 100% of the sustained bandwidth available from the array.

It is a disappointment that the highest bandwidth we could get from the ext3 filesystem (Fig. 25.1) is 50% below the measured array's sustained performance (Fig. 24.7); we switched to ext2 and got the same results, so we looked for possible causes. VFS and the VM

[4] Increasing this value brought no sizeable benefits.
[5] There has been some going forth and back in different kernel versions on whether flushing calls should wait that everything is committed to disk or immediately return after triggering the flush...

subsystem policies for the page cache, as mentioned above, do contribute for this decrease, but other possible explanations include the fact that ext2/3 implementations have been reported on several online sources to be below what can be achieved with other better performing file systems available for Linux, such as XFS [Chi+06].

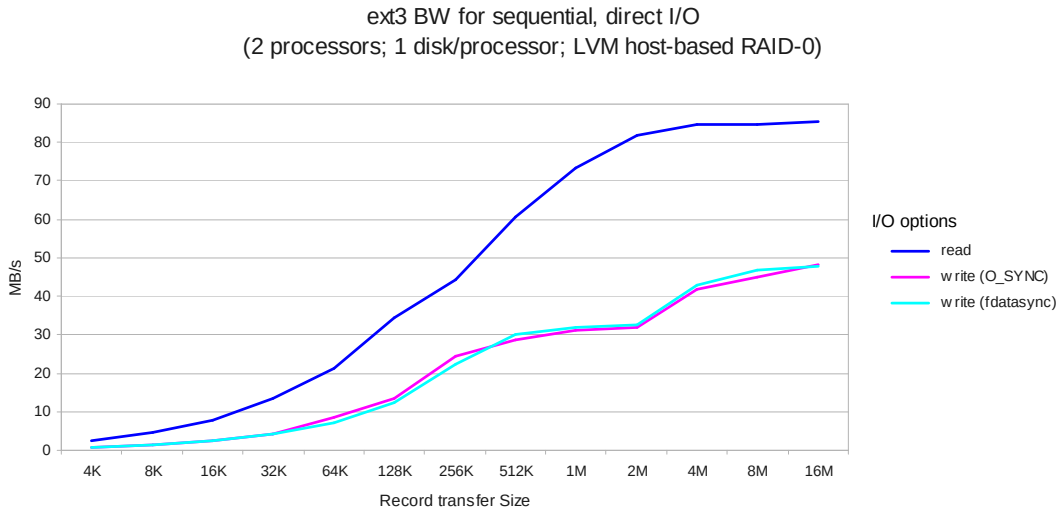


Figure 25.2 Direct I/O in the ext3 striped volume.

25.3.2 Multi-process experiments

A set of experiments involving regular buffered I/O with multiple executing processes running on a single node was then performed, the main objective being the characterisation of the node's behaviour when, e.g., the node is used as a file (NFS or PVFS) server and has to serve multiple concurrent requests – omitting the "network" and the DFS parts, just to see how the local file system and storage subsystem do perform.

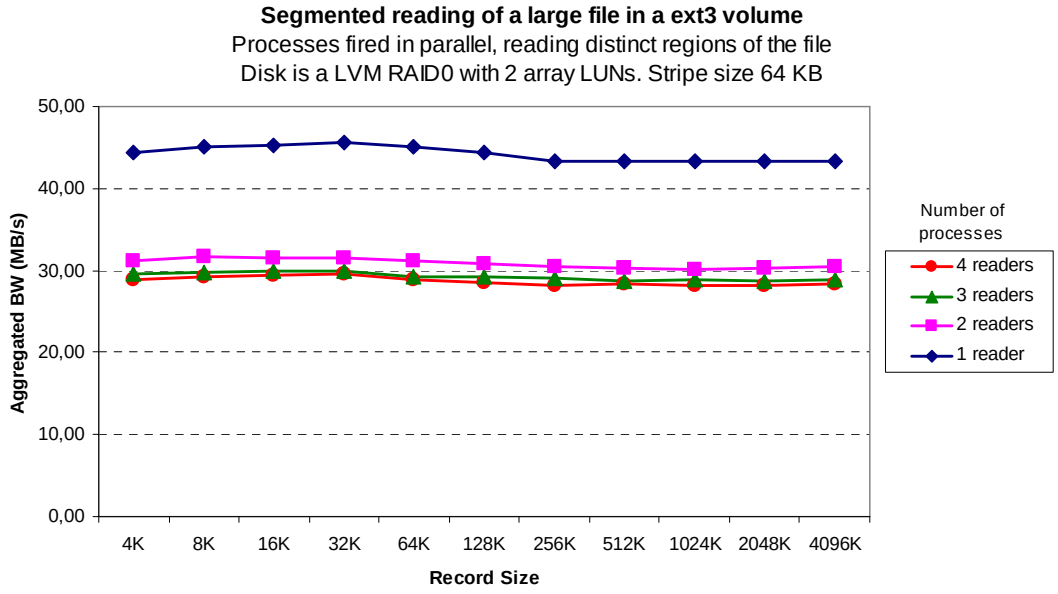


Figure 25.3 Segmented reads for increasing number of concurrent readers

Three sets were run: multiple readers (Fig. 25.3), multiple writers (Fig. 25.4), and sharing a file among a single writer and multiple readers (Fig. 25.5), all accessing distinct, non-overlapped ranges within the same file; as the number of actives processes is increased, so is the size of the "region" under access, in order to force each client to access a minimum of 4GB to avoid any cache effects; for example, with one and two processes we use a 8 GB access range; with three processes, a 12 GB access range; and, finally, with four processes, a total of 16 GB are accessed.

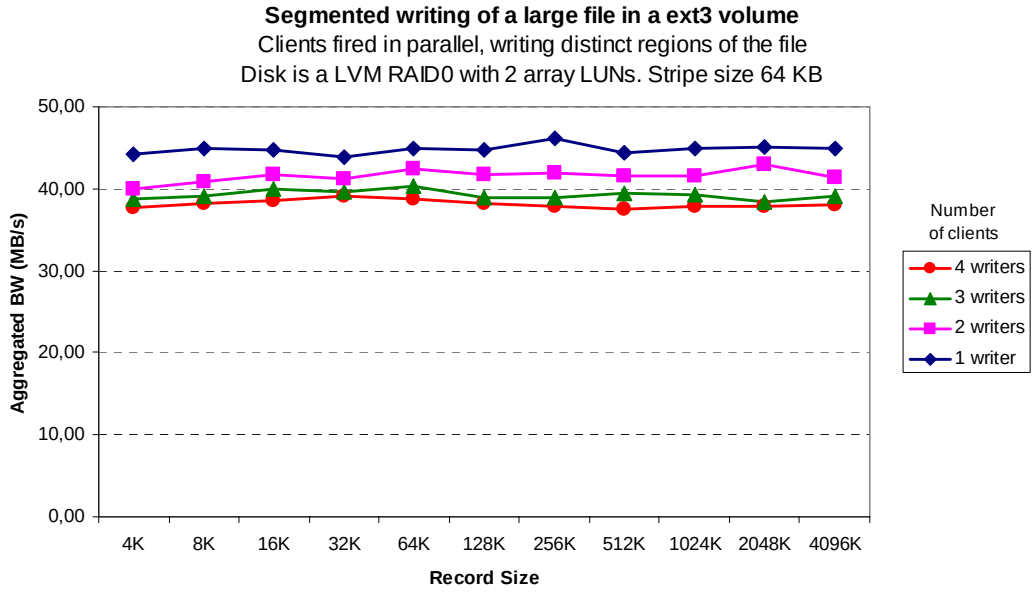


Figure 25.4 Segmented writes for increasing number of concurrent writers

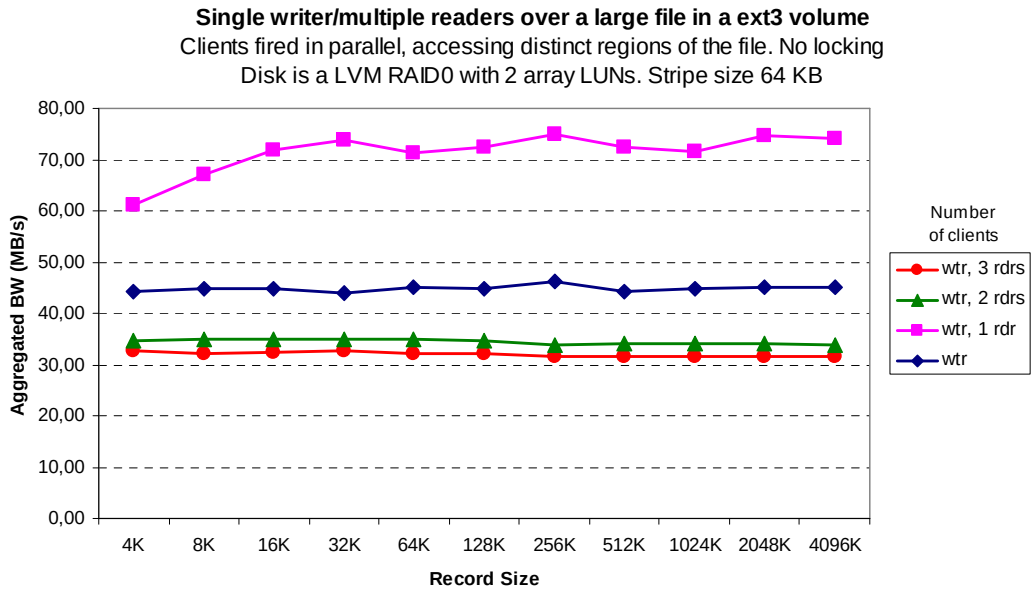


Figure 25.5 Single writer / multiple readers, non-overlapping regions

In short, every segmented test shows that tried configuration with a single logical disk made up from an LVM based RAID-0 with two disk drives, one per controller – which fared well under the sequential tests – cannot, in general, cope with the demands of the segmented, large seek inducing access pattern, as exercised by these tests. With a single exception – the shared single writer/single reader test in Fig. 25.5 – all tests show that aggregated bandwidth decreases as the number of "tasks" (processes, here) increases.

Explaining the anomaly, i.e., the single writer/single reader test faring much better than the rest is not something that we will pursue, as it requires a much deeper investigation (one cannot, however, fail to notice that in this test data is moved across a full duplex link in opposite directions). We are satisfied to get a baseline of a single node (in this case, a node with two HBAs) in order to get a better understanding of multiple node tests.

26 NFS tests

26.1 NFS test infrastructure

For the NFS tests we have used the best configuration we could get from experiments carried out in the previous section: for the (single) server we used a node with two FC adapters, 4 GB memory and two 3.06 GHz Xeons; the disk array was configured with one disk per storage processor, and the disks were striped with LVM to create a single volume that is accessed through both adapters in parallel; the volume was formatted as an ext3 filesystem and a single 24 GB file was created; finally, for the Gigabit adapter, we could not use Jumbo (MTU 9000) frames, as NFS simply hanged, so we had to resort to regular sized frames (MTU 1500).

For all tests we used NFS v3; at the server the ext3 volume was mounted with `noatime` and exported with the `async` option while, for clients, read and write sizes of 32 KB over a TCP client/server channel were used (`rsize=32768, wsize=32768`); furthermore, unless otherwise noted, all tests were run against 8 `nfsd` daemons, and were performed three times to get averaged results (except when taking Munin CPU and other statistical data, where a separate single run was taken in order to get simple, uncluttered graphs).

26.2 Reading from the server's cache

Full file scan tests were carried out to determine the bandwidth available to (seen by) clients when sequentially reading a file; first, we explored buffer sizes from 4K to 4 MB in a single client test reading from the server's cache; the result is a bandwidth of 116 MB/s, quite *close to the value we've predicted in section 9.3* (eq. 9.8) and to the GbE maximum, as measured with `netperf`; we use it in Fig. 26.1 to denote the "upper limit" in bandwidth for our configuration. Keeping the amount of data accessed small enough to fit in the server's

cache and increasing the number of clients results in an increased aggregate bandwidth, but the bandwidth seen by each individual client drops in proportion (not shown).

26.3 Segmented reading

The next test was to have each client accessing a distinct segment of the file: each one was given a different starting offset, and then proceeded sequentially reading its segment (all were disjoint from each other); tests were carried with cold client caches and with data either fully cached at the server (keeping the maximum offset accessed below 1 GB), or un-cached (forcing each client to access a 4 GB region – an exception is the test with only one client, where 8 GB were accessed). Fig. 26.1 reports our previous finding for the "small" 1 GB file fully contained in the server's cache, plotted as an upper limit for bandwidth under NFS, together with tests accessing 8 GB (1 and 2 clients), 12 GB (3 clients) and 16 GB (4 clients).

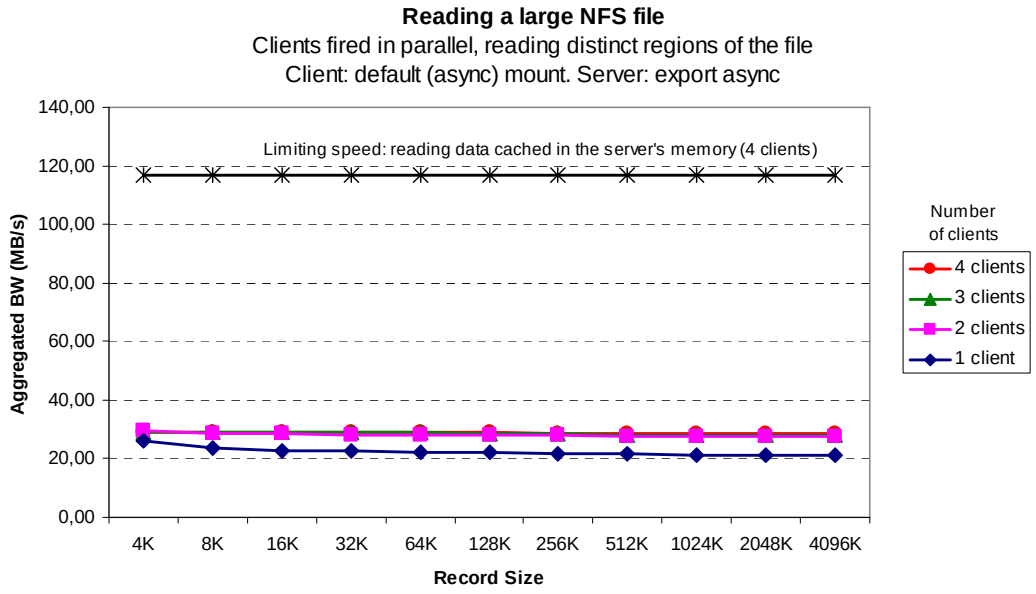


Figure 26.1 Read scalability for segmented reads over a large file

The graph above highlights two problems: the first one is the bandwidth for a single client which, at 26 MB/s, is circa 40% below the file system's bandwidth capability, at 45 MB/s; the second one is that adding more clients, in this case with a segmented access pattern, results in very small improvements with diminishing returns every time.

Loosing bandwidth with a single sequential reader is a consequence of both the application usage of synchronous reading (even taking into account the kernel's read-ahead), i.e., a new read request is only submitted after data for the previous one has been delivered, and of the increased latency introduced by NFS over-the-network request/response. However, the application can be modified to break its synchronous read behaviour either through the use of

asynchronous (or non-pending) reads or via multithreading, which will usually result in an improvement in the per-client bandwidth.

Increasing the number of clients will result in more aggregated bandwidth only if the server's storage subsystem is able to withstand the client's request rate – which, in this segmented access test, it doesn't, as it is not able to deliver the number of IOPS (I/O operations per second) required to sustain the client's request rates for this access pattern, because the array is already at its limit, 31 MB/s for local ext3 segmented accesses, shown in Fig 25.3. This may, however, be mitigated with an interposed global "intelligent" scheduler between clients and the server, such as aIOLi [Leb06]. aIOLi serialises, recombines, and reorders client requests in a way that, in the end, it will hopefully result in a more effective request sequence being delivered to the NFS server. However, aIOLi does not seem to be designed for situations where files are write shared between clients, and we could not find if it does handle request "re-combination" in the presence of file locks – something that has to be done if one uses NFS for shared file access, even in HPC applications – see below.

26.4 Segmented writing

26.4.1 Safe file sharing in NFS

NFS writing by multiple clients raises several issues on data coherency. For ease of reference, we reproduce here a fragment from 11.3.3: "the only way we can guarantee strong cache consistency in NFS (versions 2, 3 and 4) is through the use of record (also called byte-level) locking. Use of file locking in NFS requires some knowledge of its interactions with caching, otherwise the expected behaviour may not materialise". The weak cache coherency model of NFS and the fact that MPI doesn't provide user level locking primitives is the reason why, *when accessing data with a MPI application over a NFS filesystem client nodes should be configured for synchronous writing with no data or attribute caching* [Tha+04].

From the synchronicity point of view (to keep it simple and discuss only NFS v3) there are four possible combinations as we "configure" the client/server pair, ranging from both configured for asynchronous behaviour, to both being synchronous. On the server side (on `/etc/exports`) we may use either the asynchronous option (`async`) which immediately replies to clients as data is received on the server, leaving to the local filesystem/kernel the decision on when to flush data out, or use the synchronous (`sync`) option which will only reply to the client after having committed the data to disk[1]. On the client side, we can request synchronous behaviour either globally, by specifying the "no attribute caching" (`noac`) option on the mount command, or for selected files only, using the `O_SYNC` option on the file open. We have not tested for synchronous writing on the server side; it is well know that it

[1] Things may be a little bit confusing, as to guarantee a true end-to-end synchronous operation one should also mount the server's local filesystem with the `sync` option.

leads to a large drop in performance, and we feel that its use is difficult to justify on the grounds of "protecting against data loss": a typical application uses several related files making it difficult to recover when all but the one which was being written at the time of failure were successfully committed to disk; it is usually simpler to restore all files.

Therefore, tests were carried out always with the server's `async` export option. In the next set of tests we investigate NFS' write scalability by increasing the number of clients which are concurrently writing to the server; and we test both for the best possible performance case (but an unsafe one which my lead to lost updates) where each client caches data and metadata at will and writes asynchronously, and for the "correct" (safe) case, where we use "no attribute caching" (`noac`) option together with locking.

26.4.2 Unsafe file sharing: searching for maximum performance

From the HPC point of view, write file sharing is not an infrequent case; as such, we will try to determine what we can achieve as "best case" in performance terms when writing a large file; we start from a situation where clients cache data and metadata at will and asynchronously write to the server (which also caches data and metadata, flushing it at will). As depicted in Fig. 26.2, a single client writes at 35 MB/s, i.e., using only about 1/3rd of the available GbE link bandwidth; increasing the number of clients results in minor variations in bandwidth usage, with three clients better than a single one, but both two and four clients performing worse than just one. Of course, one can only use this configuration when applications do not concurrently share files for writing (the presence of a single writer is enough to trigger coherency issues); in this case our multiple writers test is grossly unsafe, possibly suffering from lost updates.

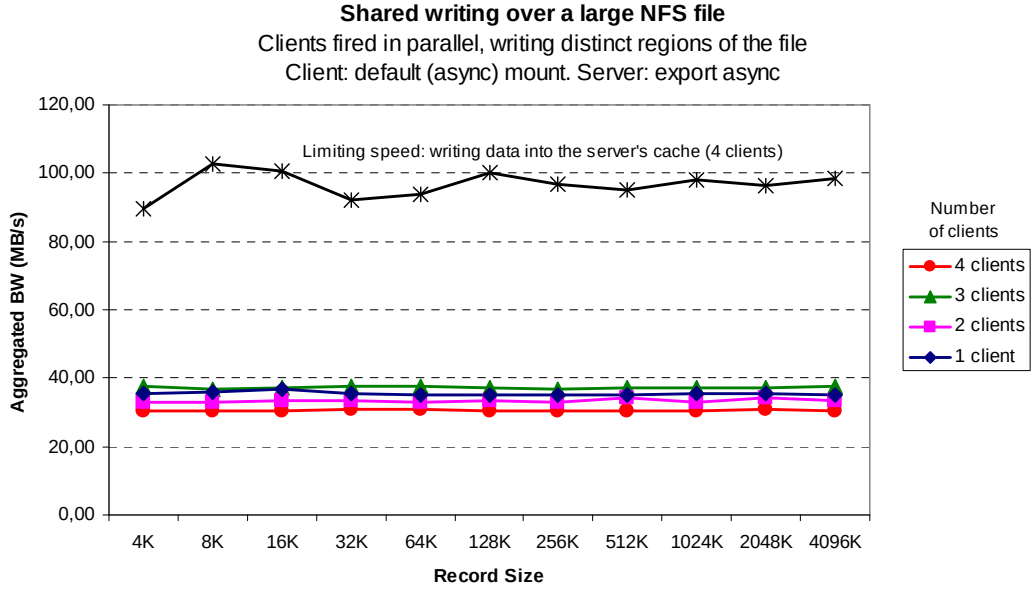


Figure 26.2 Write performance: best values with an "unsafe" configuration

We have another anomaly, now in the test with three clients; again, we will not try to explain it, one reason for it being that these configurations are not usable in “real life” situations, as they do not guarantee proper file coherency.

26.4.3 Coherent file sharing: client locks its entire region, writes, and then unlocks it

Now we look at two different ways of using NFS to share a file among writers running in distinct clients without introducing coherency problems; in both cases clients mount the NFS filesystem with the `noac` option and the applications use `fcntl()` locks.

Our first case looks at the performance we can get if clients access non-overlapping regions of the file in the following way: first, every client, using the standard `fcntl()` call, locks the entire region that it will access; then, it sequentially writes over it. Our findings are reported in Fig. 26.3; the first thing we notice is that single client performance is 15 MB/s, a drop of almost 60% when compared to the single writer in Fig. 26.2, and a consequence of the combined action of locking and `noac` resulting in a write through behaviour. As clients are added, aggregated bandwidth does increase, reaching a maximum of about 26 MB/s, a drop of about 30% from the “unsafe case” and a feeble usage of a Gigabit Ethernet link.

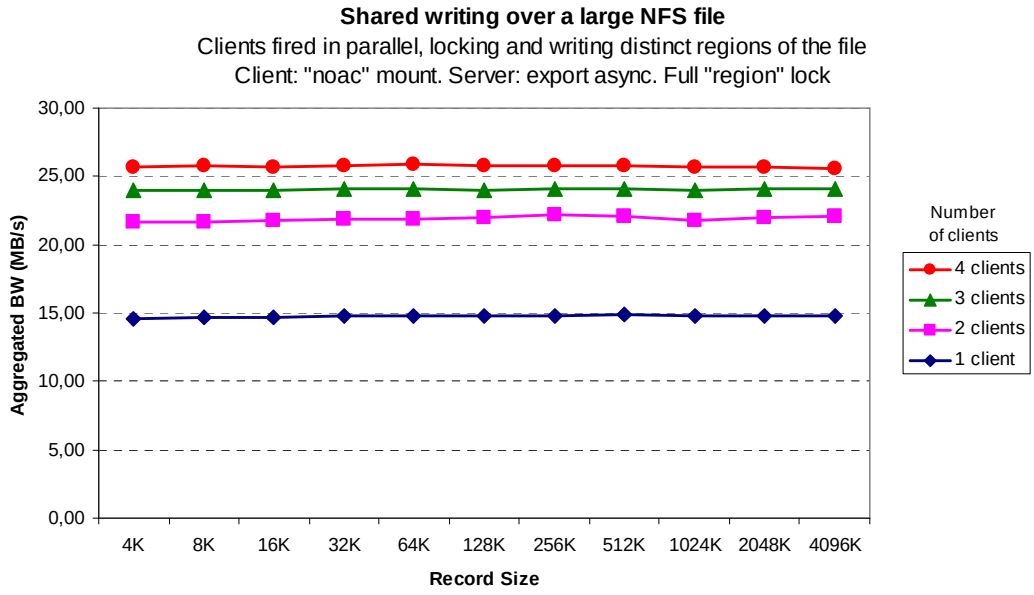


Figure 26.3 Write performance with region locking

26.4.4 Coherent file sharing: per record lock/write/unlock

Our investigation on NFS’ performance continues with a simulation of what would happen when an MPI application writes over a NFS shared file – we keep the `noac` option, use “regular” non-MPI processes (clients) which lock just the bytes they are going to write into, write, and finally remove the lock, as performed by the ROMIO driver for NFS.

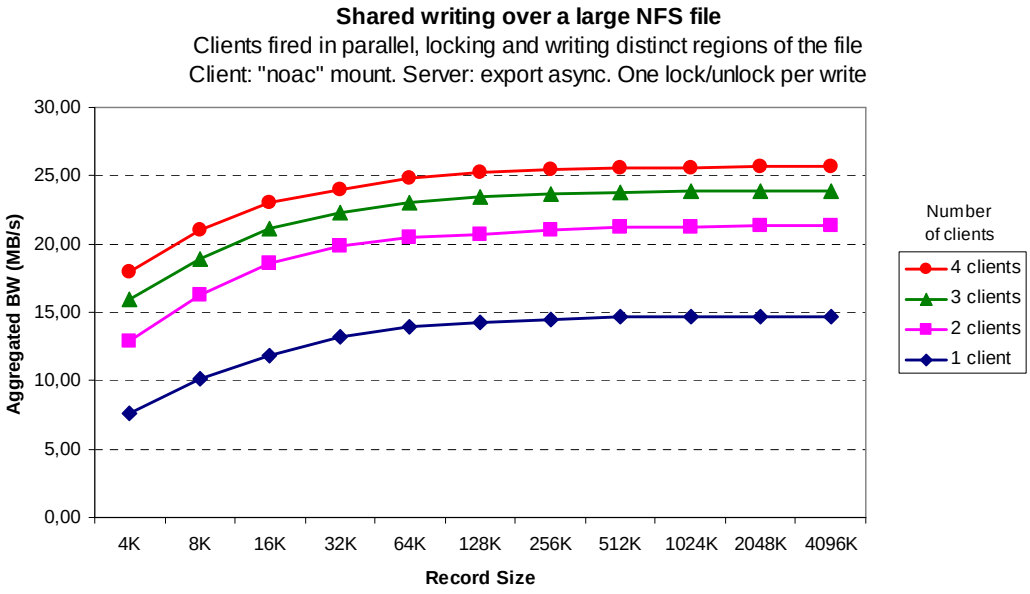


Figure 26.4 Write performance with record locking

The overhead of the locking protocol becomes quite clear when we look at Fig. 26.4: for small writes, the latency of the request-reply traffic exchanged with the server when the client asks for a lock (and releases it) becomes an important factor in the overall performance drop, particularly if multiple clients are involved; however, for record sizes above 64 KB the overhead becomes less important *vis-à-vis* the time necessary to complete the write, so the bandwidth is just slightly below the value we've got in the previous "big region lock" experiment on Fig. 26.3.

26.4.5 File sharing with a single writer/multiple readers

We conclude with a last experiment, one where we deal with a scenario that can be found in several parallel applications: file sharing among a single writer and non-overlapping multiple readers. It is an interesting test, as it *may*, under the right circumstances, be performed without forcing clients to use both locking and synchronous behaviour together (even if readers have stale data cached, they won't access it); however, it requires the use of an invalidation protocol, one that would trigger invalidation of cached stale data – and this is something that does not exist in NFS. Therefore, we start with a full region lock/access/ /unlock test similar to the one reported in Fig. 26.3; the difference, now, is that tests are performed with a single writer and an increasing number of readers. As before, the use of `fcntl()` locking and filesystem `noac` mounting at the client guarantees correct behaviour at the expense of reduced performance.

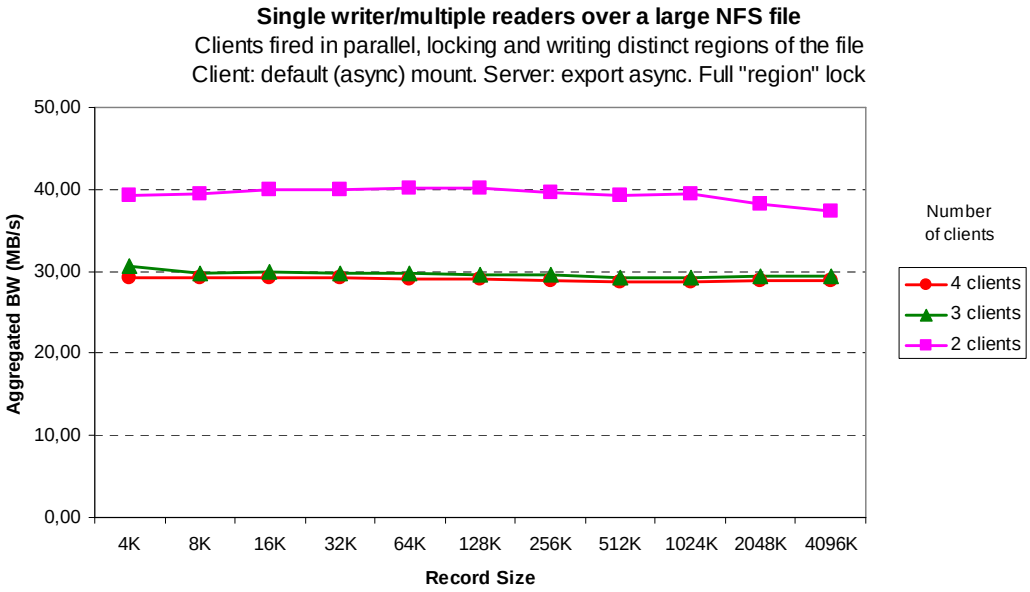


Figure 26.5 Non-overlapping 1 writer/N readers with region locking

26.5 Resource usage

The last step of this investigation on NFS usage to support HPC-like file sharing is a set of measurements both on the clients and in the server, including NFS statistics (server), disk access statistics (server) and, both on clients and server, the CPU, Ethernet bandwidth, and interrupt usage - all taken for a single run of the four client writers test of Fig. 26.4.

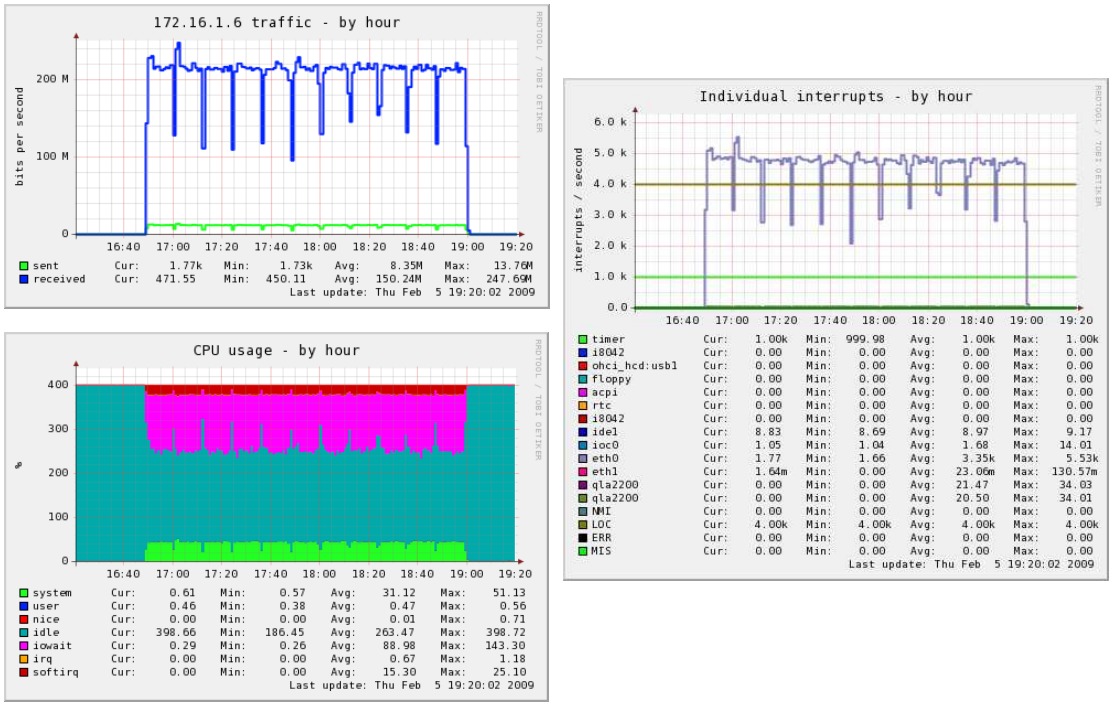
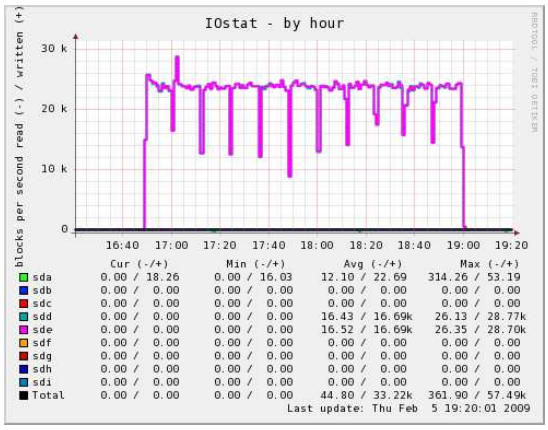


Figure 26.6 (a) Resource usage at the server: LAN, interrupts and CPU usage

Resource usage at the server shows us that the link is at 22 MB/s, about 1/5th of its full capacity, even with four concurrent clients; this is a consequence of the write-through policy applied when both `noac` and locking are used. However, CPU usage is already at about 40%, i.e., 1/5th of the two CPUs in the server has already been consumed; recalling that `netperf` used 40% just to move data across the GbE, this roughly indicates that if more clients are added and/or a more benign access pattern is used (and the server can increase its debit) CPU will probably become a bottleneck before the server's link bandwidth is exhausted.



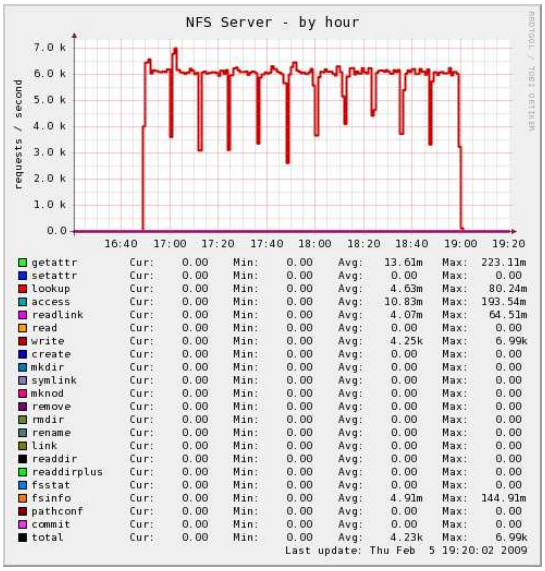
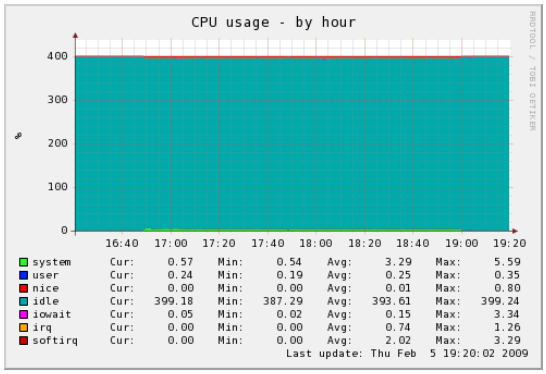
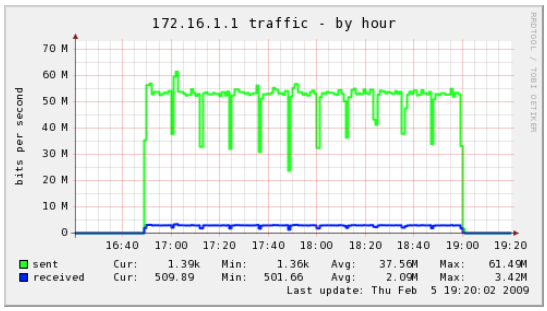


Figure 26.6 (b) Resource usage at the server: disk and NFS request rates





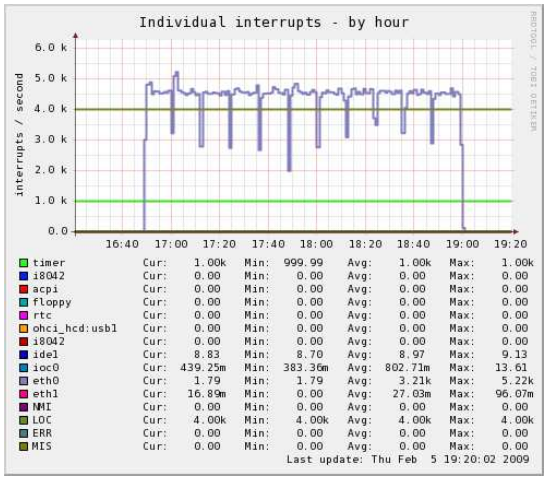


Figure 26.7 (a) Resource usage at 2.6 GHz clients

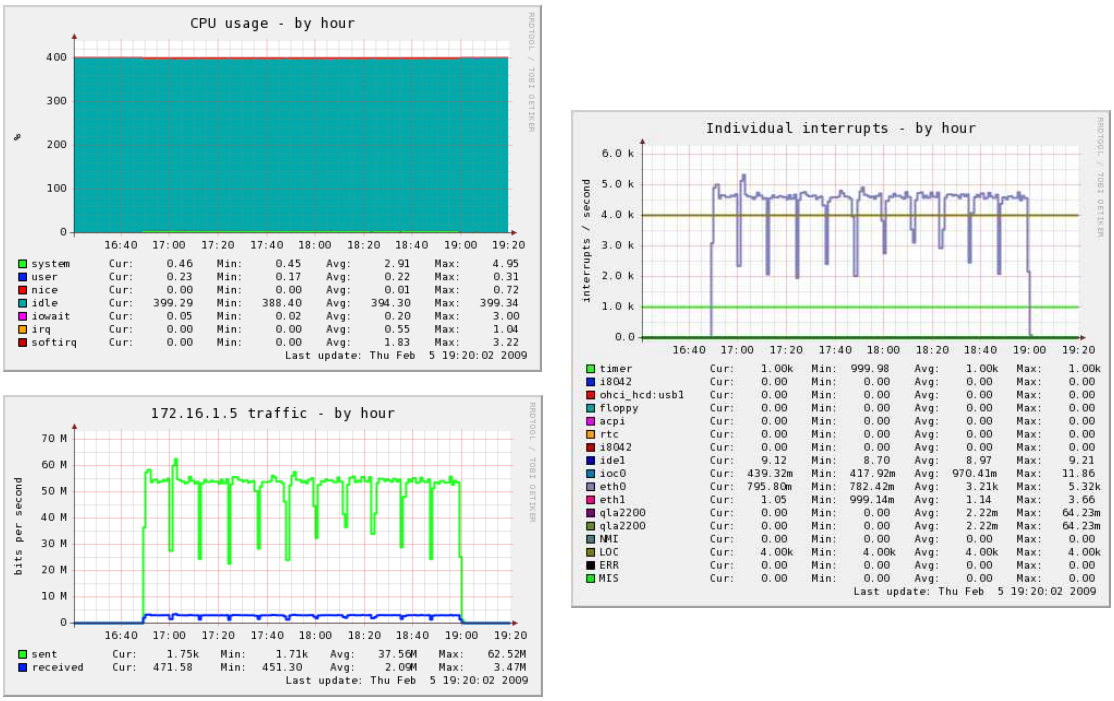


Figure 26.7 (b) Resource usage at 3.06 GHz clients

Figure 26.7 shows client resource usage; only two clients are shown, one representing nodes with 2.6 GHz CPUs while the other represents nodes with 3.06 GHz CPUs. It is obvious that, at a low rate of 5.5 MB/s (per client), CPU consumption is already becoming relevant to applications, at about 22 % – the exact amount depending on clock speed.

26.6 Summing up NFS results

Table 26.1 below summarises the NFS results; for shared write tests, only results with no client caching and “big region” locking are included. CPU usage represents the maximum usage over the whole 4K-4096K range, and occurs in the test with 4 writers.

	Readers				Writers				1 Writer/ N readers			CPU usage	
KB	1	2	3	4	1	2	3	4	1	2	3	Clients	Server
4	26.1	29.5	28.8	29.0	14.6	21.7	24.0	25.6	39.2	30.7	29.3		
8	23.5	28.7	29.0	29.1	14.7	21.7	24.0	25.7	39.4	29.8	29.3		
16	22.6	28.6	29.0	29.0	14.7	21.8	24.0	25.6	40.0	29.9	29.2		
32	22.5	28.2	29.0	29.0	14.8	21.9	24.0	25.8	40.0	29.8	29.2		
64	22.3	28.2	28.9	29.1	14.8	21.9	24.1	25.9	40.2	29.8	29.1		
128	22.1	28.1	28.6	28.9	14.8	22.0	24.0	25.8	40.1	29.6	29.0	45.2	39.3
256	21.8	27.9	28.5	28.8	14.8	22.2	24.1	25.8	39.6	29.6	28.8		
512	21.5	27.7	28.3	28.6	14.9	22.1	24.1	25.8	39.3	29.3	28.7		
1024	21.3	27.8	28.3	28.6	14.8	21.8	24.0	25.7	39.4	29.2	28.7		
2048	21.3	27.7	28.3	28.5	14.8	22.0	24.1	25.7	38.3	29.4	28.9		
4096	21.4	27.7	28.3	28.6	14.8	22.1	24.1	25.6	37.4	29.4	28.9		

Table 26.1 Summing up NFS results

26.7 Concluding remarks

Quite surprisingly, NFS testing was a nightmare; we had a NFS server problem with two kernel versions – Scientific Linux 5 2.6.18-8.1.15.el5 and CentOS 5.2 2.6.18-92.el5 – and, to fix them, we had to install version 2.6.18-92.1.18.el5. The problem was related to NFS writes: performance with a single writer was 2 MB/s before, and went up to 35 MB/s (Fig. 26.2) after the upgrade. Then, we had to abandon the single writer/multiple read tests with small locks as, when the reader client had already read about the same amount of data as the node's memory size, the Linux kernel would sometimes invoke the kernel OOM (out-of-memory killer) and start killing processes, sometimes even hanging or crashing the system. Another problem we've found with the NFS server was that sometimes, after a client crash, it did not drop the locks left out by the client.

When we changed the client kernels to the newer version (2.6.18-92.1.18.el5) we've re-run the tests of figure 26.4 for four clients, and found differences within 3%, which we deem not relevant; so all NFS client tests reported here use the older kernel version (2.6.18-8.1.15.el5) while for the NFS server we've used the newer version (2.6.18-92.1.18.el5).

27 PVFS tests

27.1 PVFS test infrastructure

For the PVFS tests we defined a configuration with 6 nodes: two I/O servers, one metadata server (doubling as client), and four clients. For I/O servers, we tested two alternative configurations: one where I/O servers have internal disks (one disk per server) as shown in Fig. 27.1; and another where I/O servers use LUNs provided by the disk array, each server mounting its private volume, as shown in Fig. 27.2.

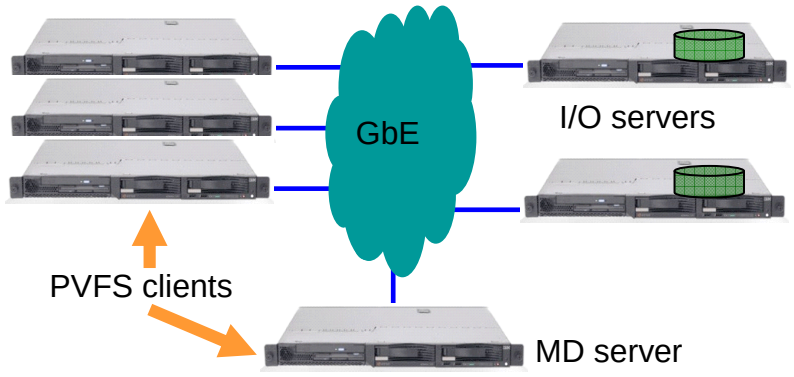


Figure 27.1 PVFS test configuration: I/O servers with internal disks

The reason behind the configuration with external disks, hereafter referred as HA-PVFS, is that I/O servers with internal disks cannot withstand node failures: if an I/O node fails, the file system becomes unavailable; with HA-PVFS, a "spare" node mounts the LUN "left over" by

the crashed node, and restarts the PVFS daemons; clients can, after a brief pause, resume access to the file system.

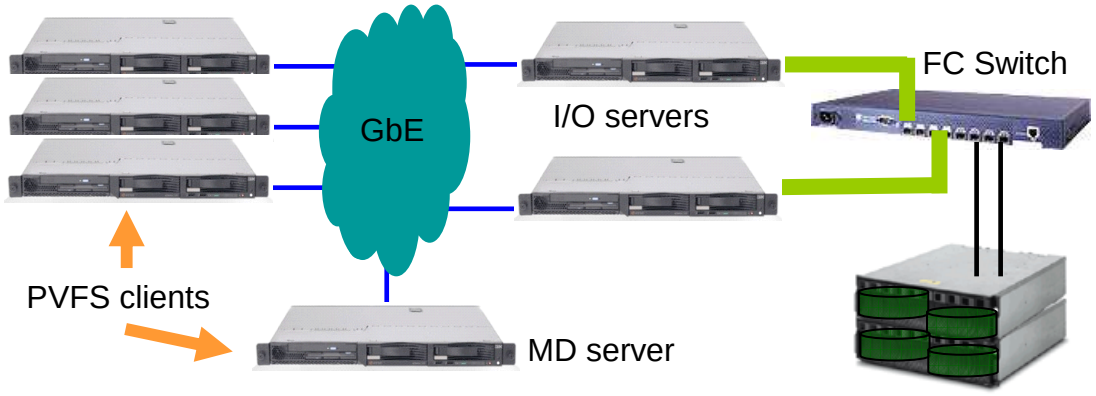


Figure 27.2 HA-PVFS test configuration: I/O servers with external disks

For the I/O servers that access external disks, we have chosen “fat” nodes: each node has two FC adapters, two 3.06 GHz Xeons, and 4 GB memory; the disk array was configured as before, with one disk per storage processor, and the disks were striped with LVM to create a single volume that is able to be accessed through both adapters in parallel. The difference, now, is that we use one logical volume per server, so we are using a total of four disk drives. PVFS data stores were formatted as ext3 file systems and, over the PVFS filesystem, a single 18 GB file was created with PVFS’ defaults: a 64K stripe and a round robin distribution which places every other stripe in a different I/O server. On the Gigabit adapter, we used regular frames so we may do a fair comparison against NFS. Testing was performed using version 2.7.0 and the POSIX interface; this allows us to reuse the same applications – with locking calls disabled; this decision (as explained before in 25.1) does, of course, leave out untested one major aspect in PVFS: its integration with MPI.

27.2 PVFS I/O servers with internal disks

27.2.1 Read-only tests

This set of tests characterises PVFS reading behaviour when accessing large files, ones that cannot be fully held in the I/O nodes’ caches; for that reason we always access 16 GB, to stick to the general rule stating that one should access at least the double of amount of RAM (which, when both I/O servers are accounted for, is 8 GB).

27.2.1.1 Full file scanning

Fig. 27.3 reports the aggregated bandwidth as the number of clients is increased and the whole file is sequentially scanned; accessing a file small enough (2 GB) to be fully contained in the servers’ cache allows us to plot PVFS’ upper bandwidth limit.

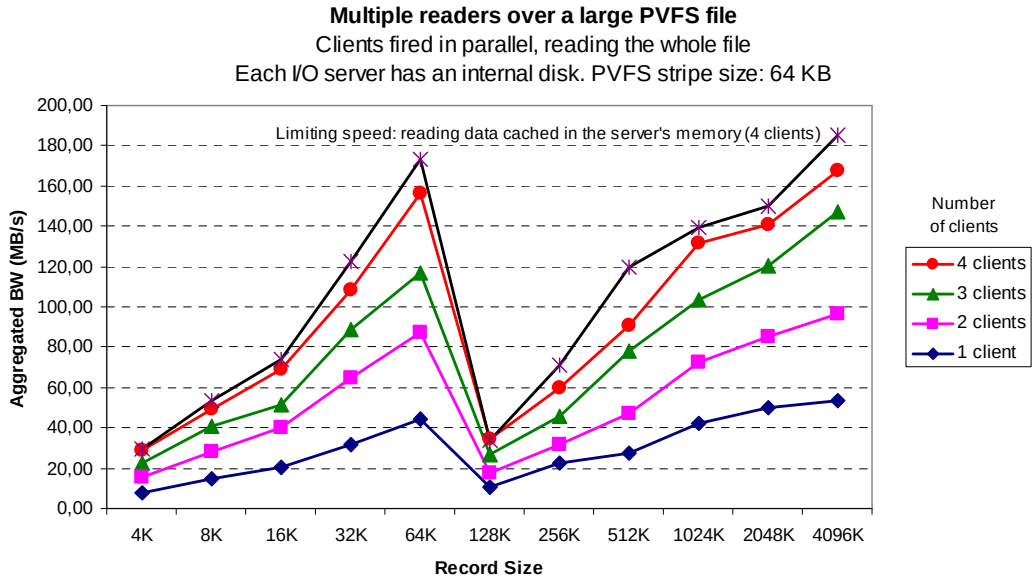


Figure 27.3 Read sharing a large file, sequential access (internal disks)

27.2.1.2 Segmented file access

The set of tests performed by PVFS clients is similar to those previously performed to evaluate NFS. Fig. 27.4 reports the aggregated bandwidth for the segmented reading tests as the number of clients is increased; as before, we access a file section fully contained in the servers' cache (2 GB) to plot the PVFS' upper bandwidth limit.

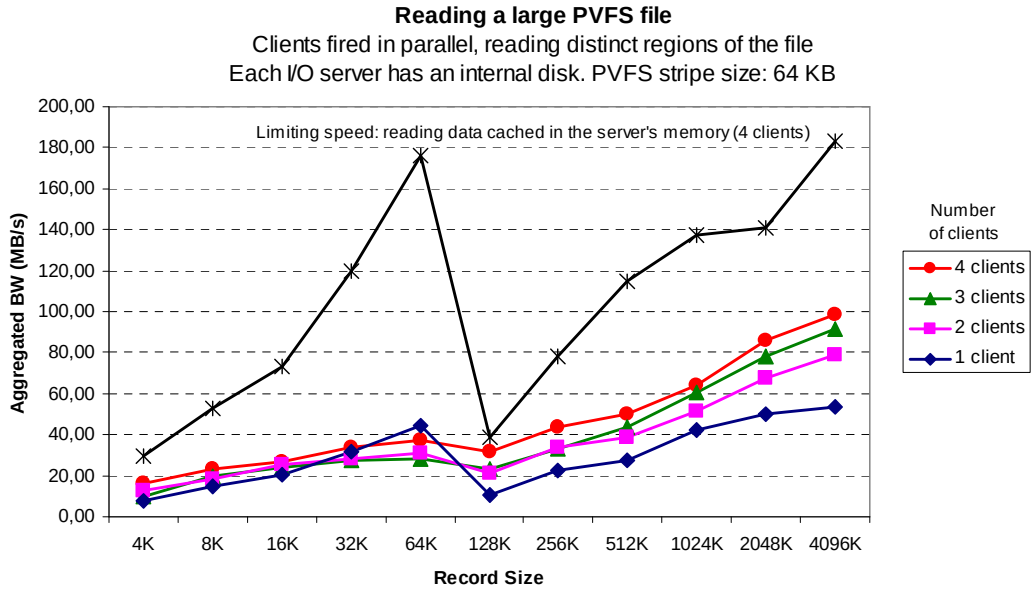


Figure 27.4 Read sharing a large file, segmented access (internal disks)

27.2.1.3 Read tests: conclusion

Our attention is obviously drawn first to the large difference in bandwidth for 64K and 128K reads – all cases exhibit it, independently of the number of clients, and whether access

is sequential (Fig 27.3) or segmented (Fig 27.4) and data is cached or not – effectively creating two distinct ramps where performance steadily rises as the buffer size is increased.

Another interesting result is that bandwidth for cached data access reaches 185 MB/s for sequential access (and very close, in segmented access), which is about 20% less than the maximum reported by `netperf` for GbE, at 116 MB/s per port (with two servers, one could achieve a maximum of 232 MB/s); thus, capabilities of the GbE medium are well utilised.

Finally, segmented access confirms that I/O subsystem performance is fundamental, and that I/O latencies incurred can severely limit what we can achieve, regardless of the peak performance of both subsystems (I/O and LAN); here, even stressed by "quasi-random" seek patterns, internal disks were able to deliver 40 MB/s.

27.2.2 Write tests

27.2.2.1 Segmented file access

For multiple writers over the same file, segmented access is the only test we perform (contended writes over the same region do not make much sense); these tests do not require special "precautions" with regard to coherency, as we've enforced in NFS, because PVFS guarantees coherency in a simple way – clients do not cache data and writes are atomic in respect to each other; PVFS' developers state that, if overlapping accesses are tried, the result is unspecified.

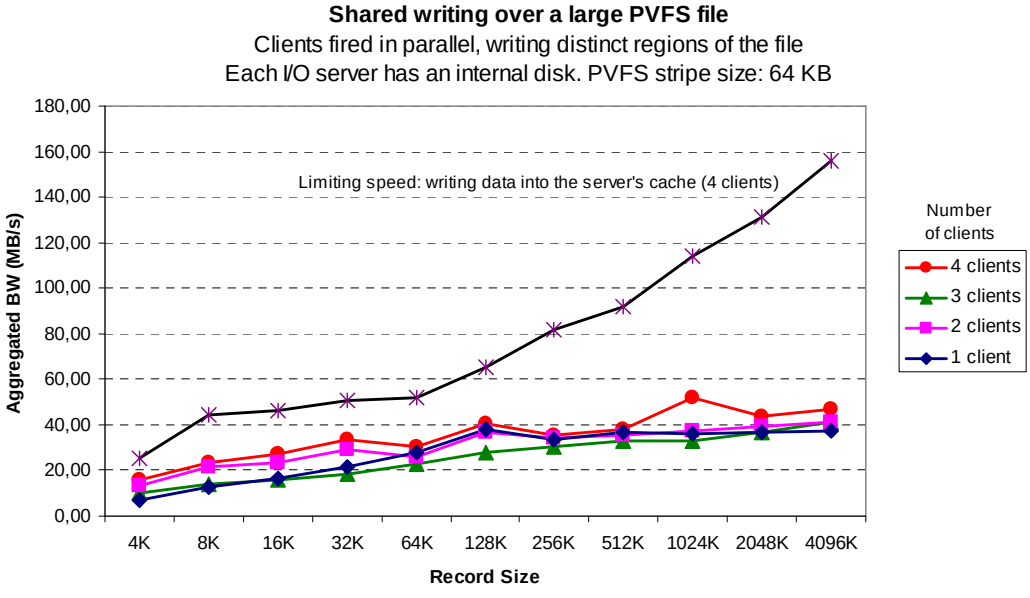


Figure 27.5 Write sharing a large file, segmented access (internal disks)

When compared with reader tests, the above write graph shows a more consistent behaviour across buffer sizes; the only unexpected result is that aggregated bandwidth for

three nodes is worse than for two and four nodes; and, for buffer sizes between 32K and 1024K, it is even slightly worse than single client bandwidth.

Writing data sizes that are small enough to be "contained" within the caches of the PVFS I/O servers results in bandwidth steadily increasing in proportion to the write buffer size; in this experiment we reached a maximum of circa 160 MB/s for cached writes, i.e., about 15% less than the corresponding reading test – but still showing good use of the GbE bandwidth.

27.2.3 Single writer/multiple readers tests

We conclude this set of tests with a single writer/non-overlapping multiple readers test; this test, as the multiple writers test above, can be run with no special precautions other than guaranteeing that either readers do not overlap with the writer or, if they do, the ordering must be enforced by the application because PVFS does not support file locking on its POSIX interface [Chi+07].

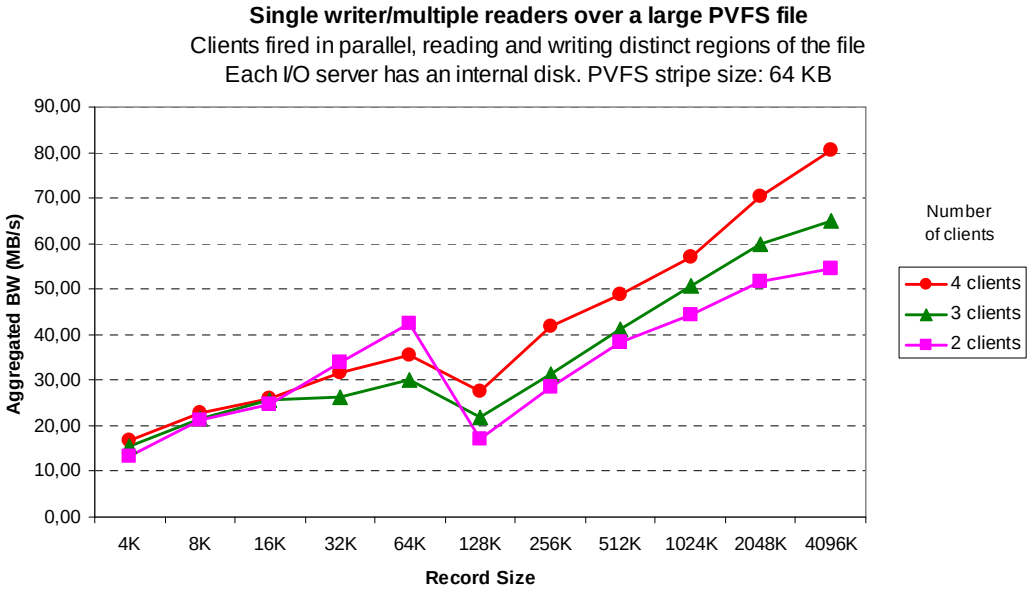


Figure 27.6 Non-overlapping single writer/multiple readers (internal disks)

Aggregated bandwidth for the single writer/multiple readers, as displayed in Fig 27.6, shows that performance increases steadily with buffer size, with the exception of the anomaly in the test with a 128K buffer size, probably a manifestation of the performance drop seen before in the reader tests displayed in Figs. 27.3 and 27.4.

27.2.4 PVFS tests with internal disks: conclusion

Table 27.1 below summarises the test results for the segmented access tests, with CPU usage reflecting worst case (4 writers, un-cached) and including the usage of both metadata and I/O servers.

	Readers				Writers				1 Writer/ N readers			CPU usage	
KB	1	2	3	4	1	2	3	4	1	2	3	Clients	Servers
4	7.9	12.5	9.6	16.1	7.2	13.2	10.4	16.1	13.2	15.6	16.8		
8	14.5	18.0	19.7	23.2	13.0	21.9	14.0	23.2	21.3	21.5	22.9		
16	20.5	25.0	23.9	27.1	16.6	23.2	15.6	27.1	24.7	25.8	26.0		
32	31.8	28.2	27.5	33.7	21.5	29.4	18.7	33.7	34.0	26.3	31.7		
64	44.3	30.7	28.3	37.2	27.7	25.8	23.1	30.3	42.5	29.9	35.6		
128	10.5	21.5	23.0	31.9	37.8	36.6	28.0	40.4	17.0	21.8	27.6	79.6	101.3
256	22.5	33.6	33.4	44.0	33.7	34.0	30.4	35.5	28.4	31.2	41.8		
512	27.8	38.7	43.3	49.7	36.5	35.2	33.0	38.2	38.2	41.2	48.9		
1024	42.3	51.8	60.6	64.4	35.8	37.4	33.1	51.7	44.2	50.8	57.2		
2048	50.2	67.9	77.9	85.7	36.7	39.2	37.0	43.6	51.6	59.9	70.4		
4096	53.4	78.9	91.2	98.8	37.6	41.4	40.9	46.7	54.5	65.1	80.5		

Table 27.1 PVFS results for I/O servers with internal disks, segmented access

27.3 PVFS I/O servers with external disks (HA-PVFS)

27.3.1 Finding the appropriate configuration

We conducted our first external disk tests with in a configuration with a single physical disk per I/O server, one where each server's LUN was owned by a different storage processor in order to provide a contention-free path; the access pattern was segmented, as before, and we performed a single test with four readers; results are recorded in Table 27.2.

	Record Size (KB)										
	4	8	16	32	64	128	256	512	1024	2048	4096
Aggregated BW (MB/s)	4.6	7.7	12.3	18.2	27.3	28.3	35.4	43.1	45.5	48.3	61.6

Table 27.2 Aggregated BW for I/O servers with a single disk per node

A brief look at the test results shows that bandwidth is very low for small sized requests, namely when compared to what we got with internal disks, as reported in Table 27.1: there, for a 4 K record size it was about 3.5 times faster, at 16 MB/s, than here, at 4.6 MB/s; when size is increased, BW also increases but values are always below those previously recorded for the corresponding buffer sizes. We think that this drop in performance is a consequence of the increase in per-request processing latency, as the storage processor's request processing overhead (perhaps in the ms range) gets added up with disk drive latency[1].

The remaining tests were performed with four disks, configured as follows: each node was given a LVM striped volume created from two different disks, each one owned by a different storage processor. The configuration for each volume is thus similar to the one previously used in ext3 and NFS tests, and gives each node access to the maximum available bandwidth, from the node's point of view. However, this configuration raises the possibility of path

[1] Internal disks and array disks, coincidently, are identical in everything but the disk interface (FC for the array vs. Ultra-SCSI 320 for the internal disks)

contention between nodes, as: a) an application request in a node will trigger one request per HBA (to serve the two LVM stripes); b) each PVFS I/O server has two HBAs, and each one will submit one request to each SP; c) therefore, a single application request will drive both storage processors to perform four requests. If two nodes happen to submit their requests "exactly" at the "same time", as PVFS does, there will be two simultaneous requests per SP, data will have to be transferred over the same FC link, and contention occurs; if this as an effect on performance is something we will look at, further down.

27.3.2 Read-only tests

This set of tests was a re-run of the set of reading tests for large files, as performed in 27.2.1, and was carried out to evaluate the contribution of the disk array to PVFS' performance.

27.3.2.1 Full file scanning

Fig. 27.7 reports aggregated bandwidths as the number of clients is increased, each sequentially scanning the whole file; in this graph, we don't plot the BW for cached access, as it's exactly the same as in previous tests.

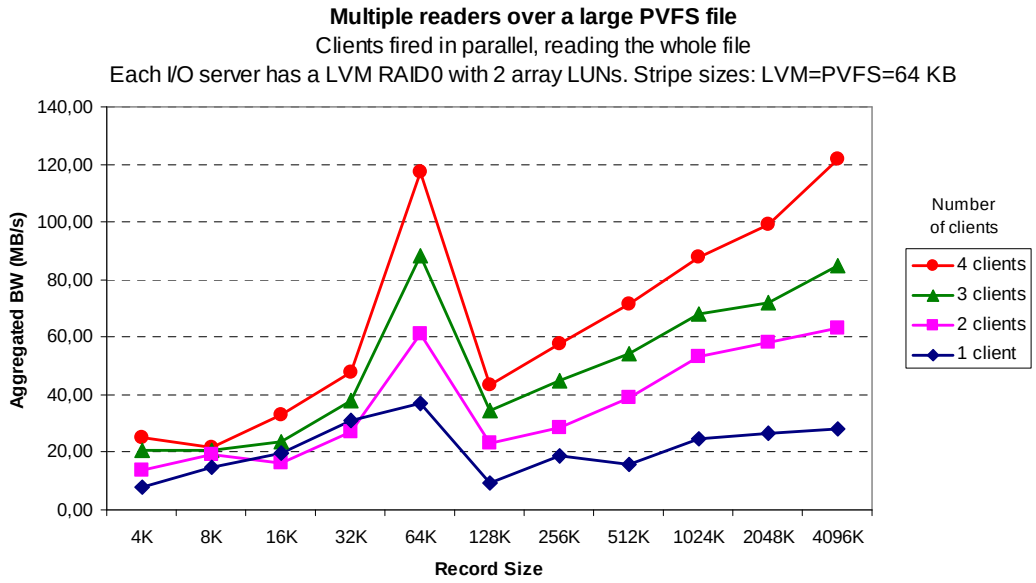


Figure 27.7 Read sharing a large file, sequential access (disk array)

27.3.2.2 Segmented file access

In the segmented reading tests each client reads its own file region, repeating the test with various record sizes, as usual; Fig. 27.8 plots the results, for increasing numbers of readers (once again we do not plot cached BW access).

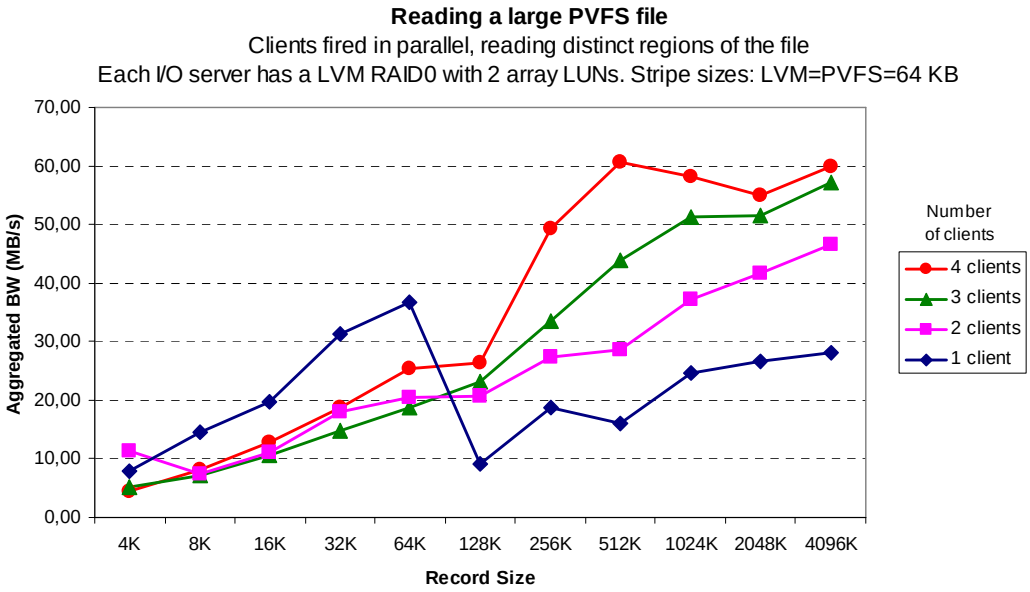


Figure 27.8 Read sharing a large file, segmented access (disk array)

27.3.2.3 Read tests: conclusion

The benefit of having each per node LUN made up from two physical disks can be seen when we compare results for segmented read tests with for four clients, as reported in Fig 27.6 (and summarised in Table 27.3 further down) with those in Table 27.1: they show bandwidth improvements for requests larger than 256K (for smaller ones, it stays essentially the same).

The array seems to reach its maximum at about 60 MB/s, for both 1 and 2-disk LUNs; this, we believe, is not caused by the aforementioned contention at storage processors and/or FC links, as ext3 experiments (see Fig 25.3) had already shown a drop from 45 to 30 MB/s when multiple readers were executed in a single node. Again, we blame latency introduced by the SP as the cause of the performance drop; in the current test, it is clear that the 60 MB/s value can be obtained through the addition of per LUN bandwidth measured under the ext3 multiple readers test pattern which is, precisely, 30 MB/s for a single LUN.

If our assumptions are correct, all results for tests with external LUNs will be worse than those obtained with internal disks; we claim this does not result from resource contention, but from the array itself. So we are currently unable to prove that a PVFS configuration with external array disks will suffer from contention problems (on the FC/array infrastructure) and deliver lower performance than one with internal disks; in order to prove it, we need to get hold of a better disk array, and rerun these tests.

27.3.3 Write tests

In this set of tests we assess the performance of our HA-PVFS configuration both under the segmented file writing test and the single writer/multiple readers test; as usual, we perform these tests with various buffer sizes and an increasing number of clients.

However, we have a new test here: we want to assess if block allocation does hamper performance: each run of the new "block allocation test" starts with an empty file, one that writer processes will "fill" as they proceed; this will trigger both metadata (indirect blocks and bitmaps) and data block allocation on the fly. We have not performed this test before, either in the NFS or in the "PVFS with internal disks" setups because we feel that other results we gathered in those tests were sufficient for our purposes, and HA-PVFS is our most important "HPC filesystem" test.

27.3.3.1 Segmented writing tests, no block allocation

Results gathered in the set of segmented write tests and plotted in Fig. 27.9 below show that our previous assumption – that bandwidths for the external disk configuration would be lower than those for internal disks – still holds; however, differences among segmented writing tests with internal vs. external disks are not so obvious as they were in the readers test: we have now reached 45 MB/s, not far from the 52 MB/s measured in the setup with internal disks.

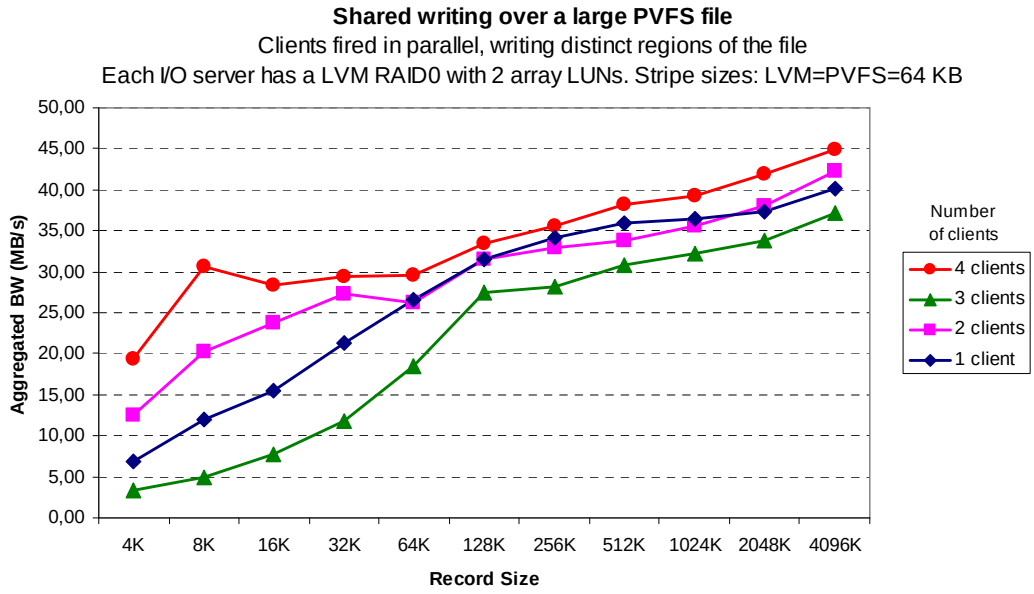


Figure 27.9 Writing a large file, segmented access, no block allocation (disk array)

27.3.3.2 Segmented writing tests, with block allocation

Here, as previously described, the sole file existing in the PVFS file system is truncated before each test; this setup guarantees reproducible test conditions as, with no other applications running, file structures will always be allocated in the same disk "areas".

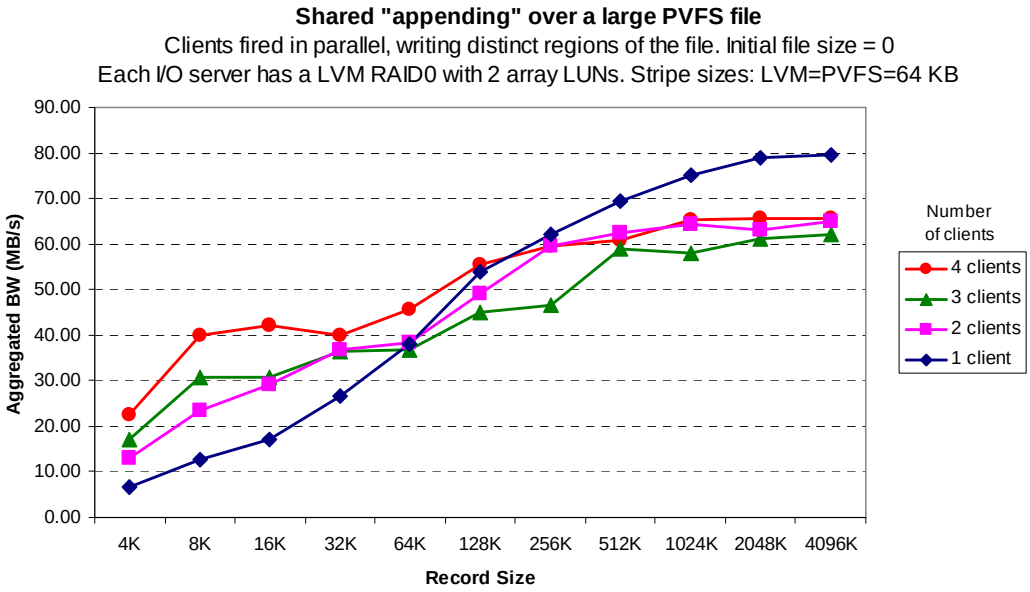


Figure 27.10 Segmented writes over a large, empty file (disk array)

27.3.3.3 Write tests: conclusion

Tests run against an initial empty file, as depicted in Fig. 27.10, intriguingly show better performance than those where there is no newly allocated data and metadata; a similar situation was also reported in [Leb06] for NFS writing against empty files. We have not thoroughly investigated this issue, but we think that lower performance may be a consequence of writes, in the pre-allocated file case, needing some extra work; they require: 1) reading the indirect blocks; 2) reading the data itself[2]; merging data gathered in (2) with new data; and finally, 3) writing the data and metadata. When the file is empty, (1) and (2) do not take place (of course, data management structures, e.g., bit maps, must be consulted and updated in both cases).

27.3.4 Single writer/multiple readers tests

We conclude the set of PVFS experiments with a test on file sharing between a single writer and multiple, non-overlapping, readers.

[2] This may depend on the file system implementation; surely, if record size is less than a filesystem block (or page, if the FS is page-oriented), the block (or page) has to be read in, first.

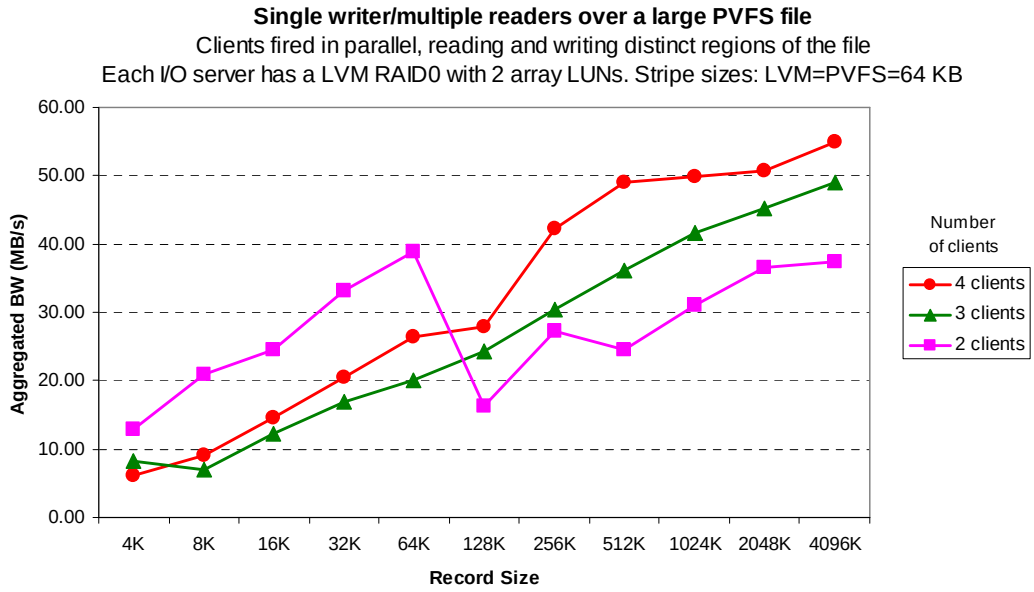


Figure 27.11 Non-overlapping single writer/multiple readers (disk array)

27.3.5 PVFS tests with external disks: conclusion

Finally, we summarise the results tests we performed with our (high availability) HA-PVFS configuration with three servers: one metadata server and two I/O servers. Each I/O server was given access to an LVM-based RAID0 LUN, created on top of two disks in the array; physical disks were assigned to different storage processors in a balanced configuration, to extract the best possible performance. For conservative use of layout space, results were grouped into two tables: Table 27.3 summarises results gathered in tests performed against a pre-allocated, 16 GB fixed-size file, whereas Table 27.4 includes two distinct sets, one (a) for results gathered from the full file scan reader tests (where the whole file was sequentially accessed by all readers), and another, (b) for results obtained from writer tests performed against an empty file "filled" by non-overlapping writers.

	Readers				Writers				1 Writer/N readers			CPU usage	
KB	1	2	3	4	1	2	3	4	1	2	3	Clients	Servers
4	8.0	11.3	5.2	4.5	6.9	12.6	3.3	19.4	12.8	8.2	6.0		
8	14.6	7.4	7.1	8.0	11.9	20.3	4.9	30.6	20.8	7.1	9.2		
16	19.8	11.1	10.6	12.9	15.4	23.7	7.7	28.4	24.5	12.3	14.5		
32	31.2	17.9	14.9	18.6	21.4	27.3	11.9	29.4	33.1	16.8	20.4		
64	36.8	20.4	18.7	25.3	26.6	26.3	18.5	29.6	39.0	20.0	26.5		
128	9.2	20.7	23.3	26.4	31.5	31.5	27.4	33.5	16.2	24.3	27.9	59.2	96.3
256	18.7	27.4	33.4	49.2	34.2	32.9	28.2	35.5	27.2	30.4	42.3		
512	16.0	28.7	43.8	60.6	35.8	33.8	30.7	38.2	24.5	36.1	49.0		
1024	24.6	37.3	51.2	58.1	36.4	35.6	32.3	39.2	31.0	41.6	49.9		
2048	26.6	41.5	51.6	55.0	37.3	38.0	33.7	42.0	36.4	45.2	50.6		
4096	28.2	46.7	57.2	59.8	40.2	42.3	37.2	44.9	37.4	49.0	54.9		

Table 27.3 PVFS results for I/O servers with external disks, part 1

Note: CPU usage reported above is the worst case value, and occurs in the test where four writers access the file with a 4K record size; under the label “servers” we have added consumption for all PVFS servers: the two I/O servers and the metadata server.

	Readers (full file scan)			
KB	1	2	3	4
4	8.0	14.0	20.7	25.1
8	14.6	19.4	20.9	21.8
16	19.8	16.4	23.9	33.0
32	31.2	27.2	38.1	47.7
64	36.8	61.3	88.5	117.3
128	9.2	23.3	34.6	43.6
256	18.7	28.8	44.8	57.6
512	16.0	39.0	54.2	71.5
1024	24.6	53.4	67.8	87.7
2048	26.6	58.0	72.0	98.9
4096	28.2	62.9	84.8	121.7

(a)

	Writers (empty file)			
KB	1	2	3	4
4	6.8	13.1	17.1	22.5
8	12.7	23.5	30.6	39.8
16	17.0	29.3	30.8	42.0
32	26.8	36.7	36.5	40.0
64	38.1	38.5	36.7	45.6
128	53.9	49.0	44.8	55.5
256	62.0	59.5	46.5	59.6
512	69.3	62.5	59.0	60.9
1024	75.0	64.3	58.0	65.3
2048	79.0	63.2	61.1	65.7
4096	79.5	65.0	62.2	65.7

(b)

Table 27.4 PVFS results for I/O servers with external disks, part 2

27.4 PVFS: resource usage

The graphs exhibited in Figs. 27.12 to 27.14 correspond to the test where four writers write over an empty file, reported in Table 27.4 (b) above.

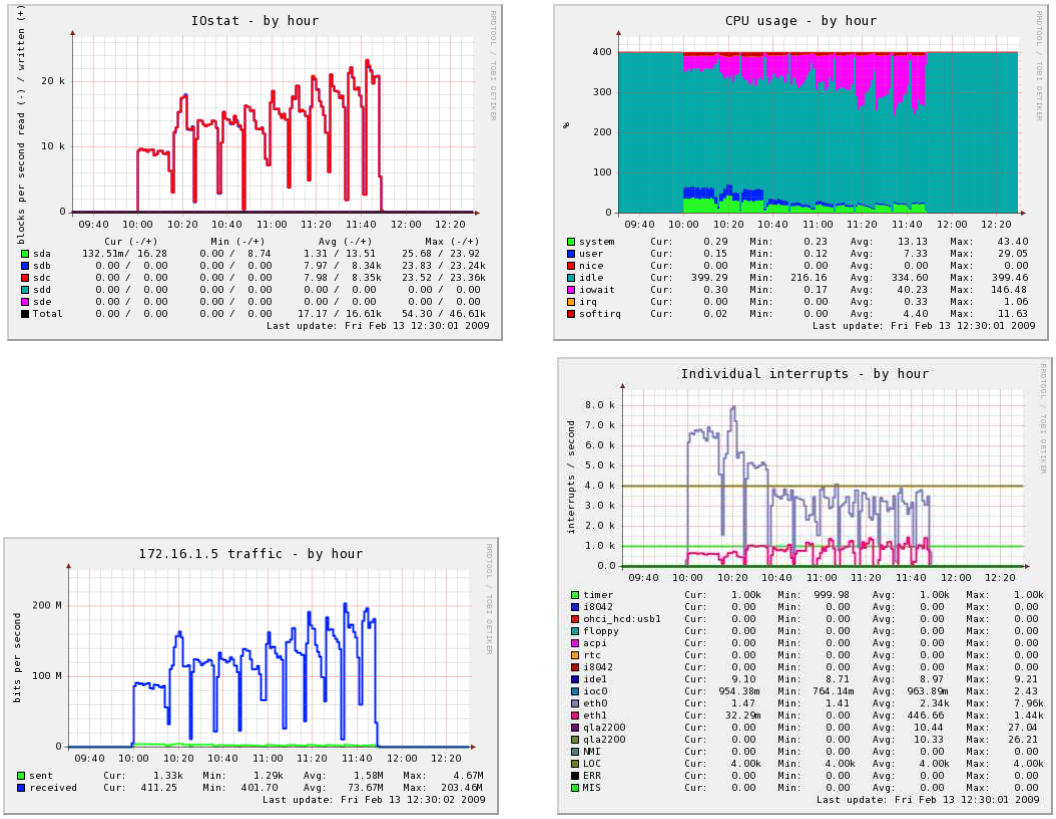


Figure 27.12 Resource usage at the PVFS I/O servers (only one server shown)

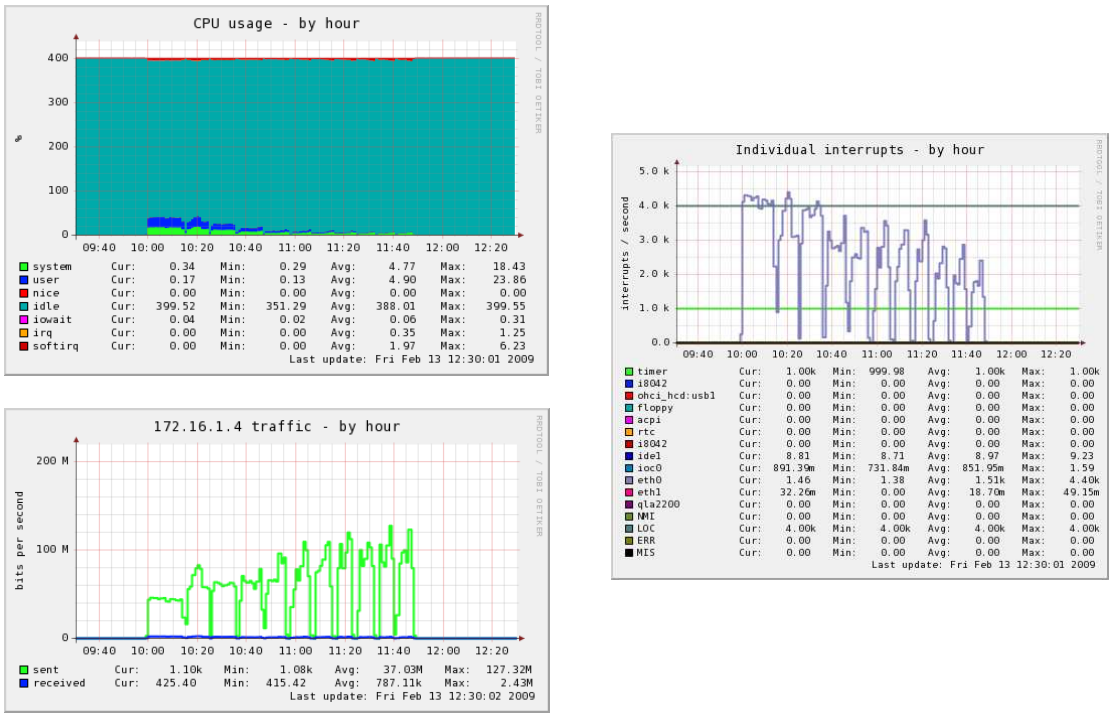


Figure 27.13 Resource usage at the PVFS MD server (see text)

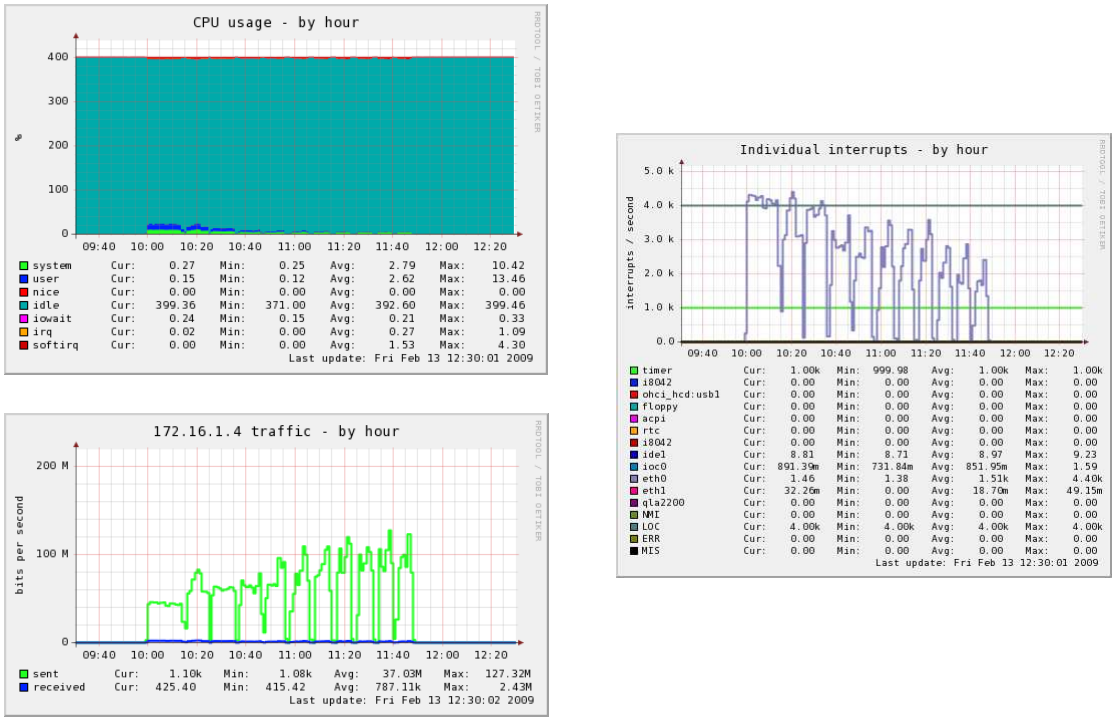


Figure 27.14 Resource usage at the PVFS clients (only a single client shown)

These graphs show that link usage at I/O servers peaks at about 1/5th of the full capacity (20 MB/s), while worst case CPU usage is already at 48% per I/O server. To that must also add the CPU usage at the metadata server, which is about 10% – as we have used the metadata

server to also run a client (because we had a hardware problem in one of the clients), to compute the CPU usage at the MD server we must pick the total from Fig. 27.13 and subtract the client usage taken from Fig. 27.14 (and divide by two, to adjust the reported "hyper-threaded value" to the number of "real" CPUs).

27.5 PVFS: closing remarks

PVFS strengths are well known and widely publicised, both in papers and technical reports; to start, aggregated bandwidth scales well with I/O node addictions and can reach high levels not only in MPI-based applications (in the order of GB/s if we include specialised interconnects such as Infiniband), but also in POSIX ones.

On the other hand, PVFS "weaknesses" other than the effort required to redistribute a PVFS volume across newly added I/O nodes, or those related with server failures (although, as we said before, they can be quite conveniently handled by the HA-PVFS setup) are not so well understood and/or reported, so we have tried to address a few:

- *PVFS is quite sensitive to the stripe size when reading* data, as "two ramp" graphs clearly show.
- *Bandwidth is quite low for small record sizes*. Although the latest PVFS versions allow the user to specify per directory (and even per file) striping sizes, and this is something that may improve BW (thus alleviating the problem above), for small record sizes (below a few KB) bandwidth is still quite low.
- *CPU consumption in I/O servers can be high* (unless more expensive interconnects are used), something that discourages users from using server nodes to run applications.
- The *cost* of having dedicated I/O servers and also external disk arrays completely demolishes the much touted argument (not by the developers!) of PVFS being a low cost solution.

Unfortunately, we cannot show that HA-PVFS configuration using a disk array – which is the *de facto* setup used in production environments – performs sub-optimally when compared to a similar configuration with internal disks, something we were aiming to prove; we believe the entry level disk array used in this tests to be the problem, as it (we think) introduces a per request latency overhead that masks out the effects we intended to show, namely interconnect contention that would arise when a client issues a request against the PVFS servers and the servers dispatch several concurrent (one could almost say "simultaneous", here) requests to the disk array thus (possibly) creating a "contention effect" in the FC paths to the disks.

28 Cluster File System testing: pCFS and GFS

28.1 Test infrastructure

pCFS and GFS tests were carried out in a configuration with five nodes: four FC-connected plus an "independent" node used for the pCFSd user-level daemon, as shown in Fig. 28.1.

Each node had 4 GB of memory and two Xeon processors; nodes 4, 5 and 6 had them running at 3.06 GHz while nodes 2 and 3 had them at 2.6 GHz.

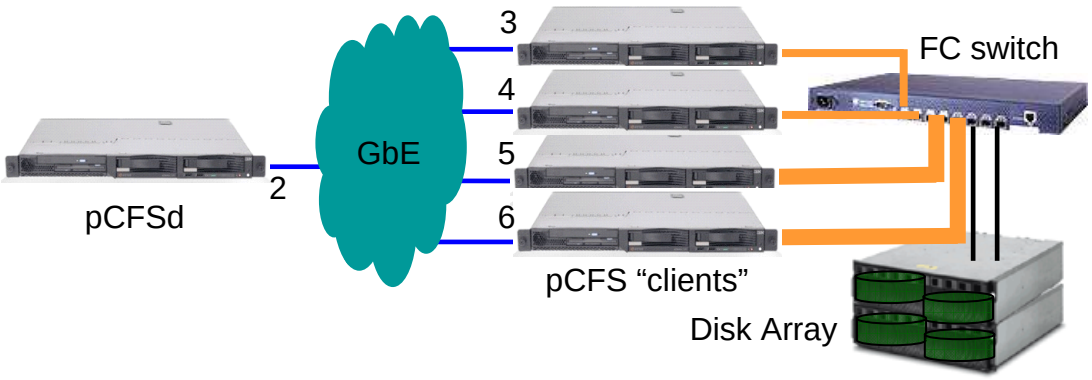


Figure 28.1 pCFS/GFS test infrastructure

All FC links were run at 1 Gbps; nodes 3 and 4 had a single FC HBA (thin links, in the figure), while nodes 5 and 6 had two FC HBAs (fat links). A single host based Clustered LVM volume was carved out from four disks, two per storage processor; the logical volume was defined with a stripe size of 32 KB, thus creating a configuration which was “equivalent” to the one used for PVFS, in terms of the number of physical disks used. The volume was formatted as a GFS filesystem and a single 16 GB file was created; finally, for the Gigabit links, regular frames (MTU 1500) were used.

28.2 pCFS vs. GFS and cached vs. un-cached testing

As we have shown before [Lop+08], performance differences among pCFS and GFS both in single writer and in single or multiple reader tests are so small (less than 1%) that they are obfuscated by variances in the tests themselves; therefore, unless we want to draw the reader’s attention to some specific GFS issue, the majority of the tests reported here were performed against pCFS, i.e., with the `O_CLSTSOPEN` pCFS flag included in the file `open()` call; so, unless marked otherwise, graphs labelled as pCFS are also considered valid GFS graphs.

We did not measure the bandwidth of cached access as, in a similar vein to what happened with local file systems (e.g. ext3), they would only give insights on the VFS cache performance itself, as well as on the overheads of the specific file system (i.e., pCFS) delivering bandwidths ranging from several hundred MB/s up to a few GB/s for a single node; therefore, all our tests access un-cached data. We went to great lengths to assure that, for successive tests, no data stays in the cache: besides using a large file[1], the Linux

[1] Notice that a 16 GB file in segmented access is actually 4GB per node in four node tests, which no longer is the double of the node’s memory.

`/proc/sys/vm/drop_caches` pseudo-file is used to force data in the page cache to be released, and the file system is un-mounted and remounted before a new test is started.

28.3 Read-only tests

28.3.1 Full file scanning

The first test was a sequential full file scan: on each node, a reader process would open the file, start at the beginning and proceeded reading it sequentially to completion.

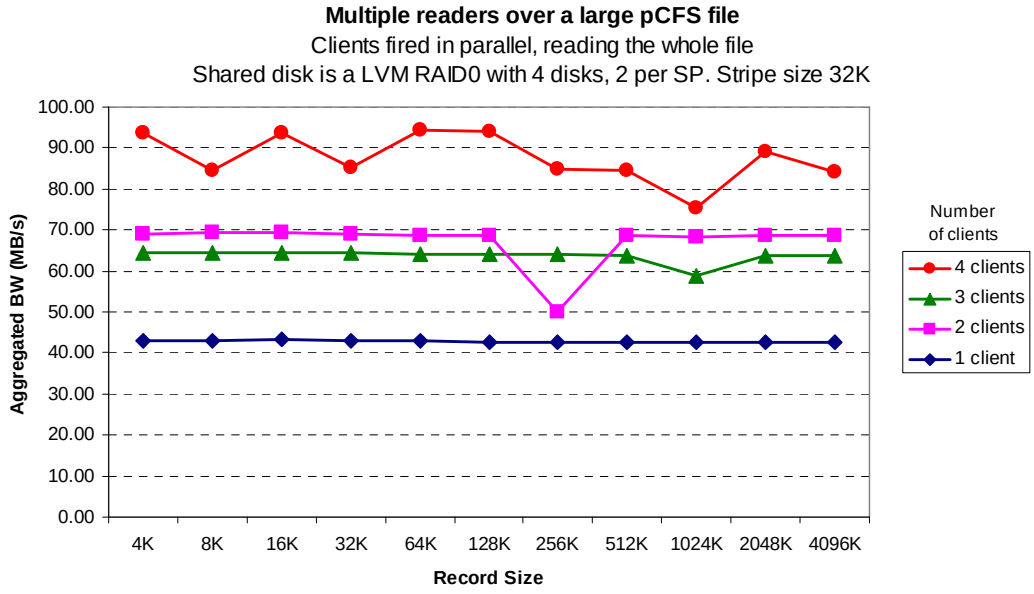


Figure 28.2 Read sharing a large file, sequential access

Test results reported in Fig. 28.2 do not, contrary to what we initially hoped for, unambiguously show the positive influence of the array's cache; maximum bandwidth in this test is 94 MB/s which, although exceeding the advertised sustained rate of the array (at 70 MB/s) is remarkably inferior to 90 MB/s *per* storage processor we got in Fig. 24.6; that would present us with a total of 180 MB/s. Our explanation is that, although processes in reader nodes were fired in parallel, their ability to proceed "in sync" (although somewhat loosely) and benefit from data already in cache is negated by configuration issues such as node heterogeneity (number of HBAs) and the small size (88 MB) of the array's cache.

28.3.2 Segmented file access

Then, a segmented access test was performed over a 16 GB file; results, shown above, demonstrate a pCFS reading behaviour remarkably similar to ext3's or, shall we say, to the behaviour of any typical "VFS integrated" local filesystem: performance is not adversely affected by small record sizes, as the VFS read-ahead mechanism "kicks in", raising it. It also shows that, for our configuration built around a four disks set, 55 MB/s is the maximum bandwidth achievable under situations where a high number of seeks is performed.

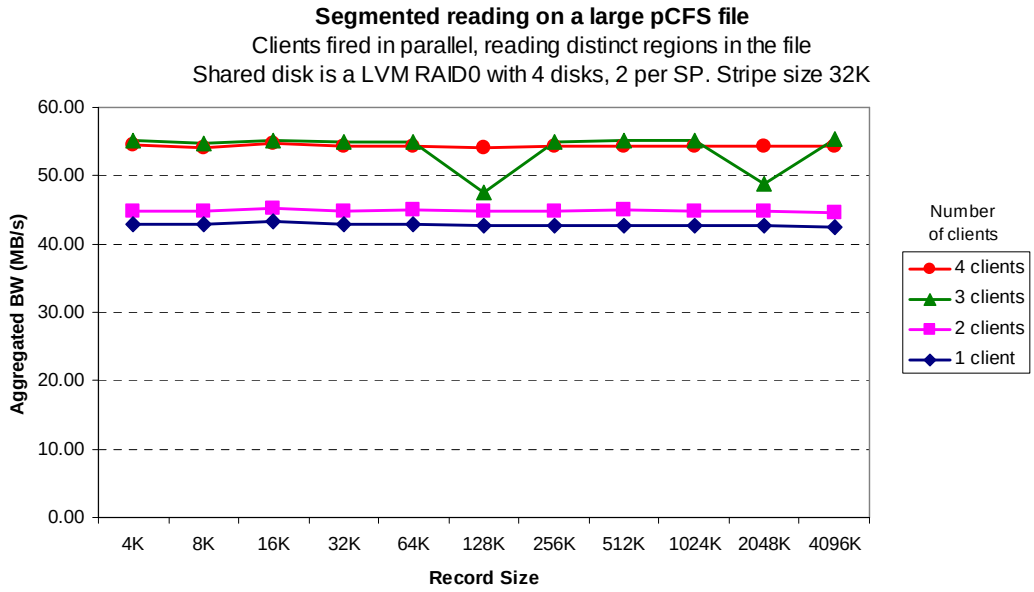


Figure 28.3 Read sharing a large file, segmented access

28.4 Write tests

28.4.1 Segmented writing tests, no block allocation

Write sharing a GFS file is, with regard to coherency, similar to PVFS – it does not require user-level file locking; in fact, as previously noted, GFS implements POSIX single node equivalent semantics and, therefore, even if two processes in distinct nodes concurrently access overlapping file sections, the result is a serialisation of the accesses and a coherent "disk" image. pCFS is different as, for disk-based data movement, it requires POSIX advisory locks to define file regions a process is allowed to access.

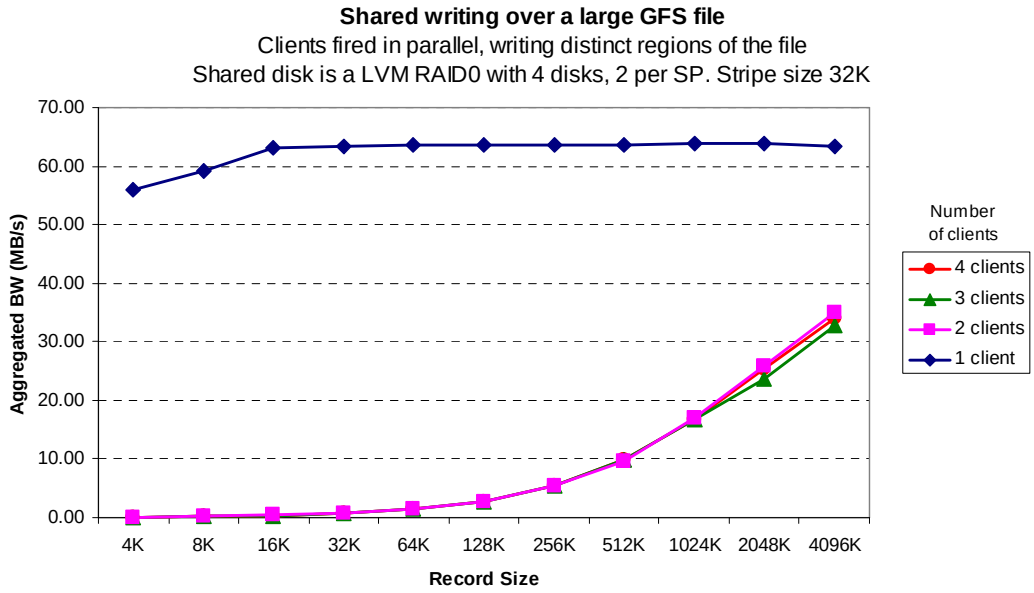


Figure 28.4 GFS: write sharing (full region locks, segmented access pattern)

GFS' use of a cluster-wide ginode lock, one which is locked for the duration of a read or write call, results in very low bandwidths (less than 2 MB/s for record sizes smaller than 128 KB) as a running writer is forced to flush out all data it has accumulated in memory to disk (an operation which takes a few milliseconds) before handing out the lock to another node.

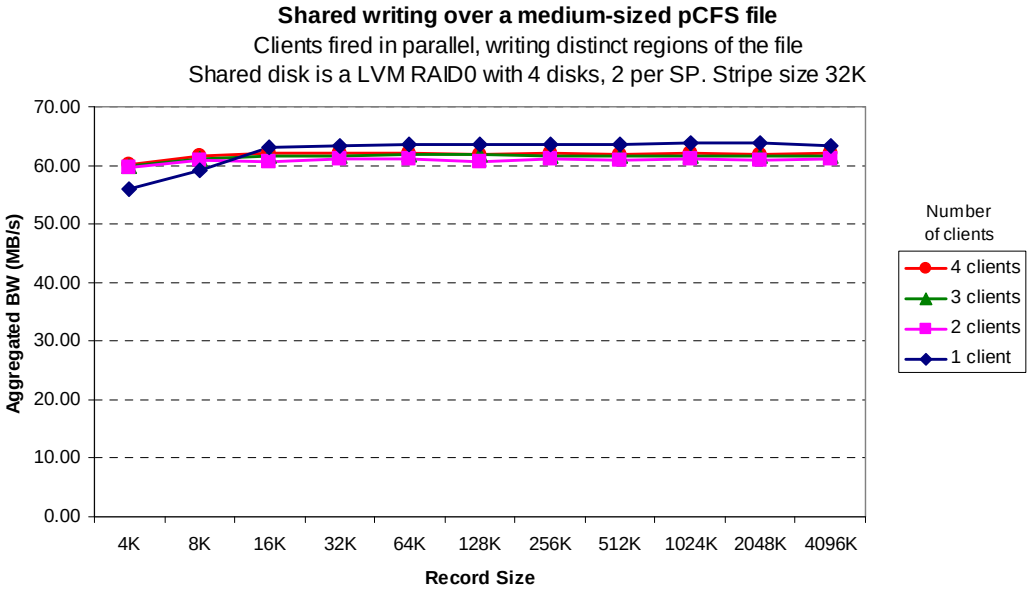


Figure 28.5 pCFS: write sharing (full region locks, segmented access pattern)

pCFS clearly overcomes GFS in the shared writers test: Fig. 28.5 shows the segmented write test[2] with pCFS, where each node starts out by laying out its region and then loops to perform all writing: aggregated bandwidth is now 60 MB/s, twice the value GFS offers on large buffer sizes, and 600 times what it offers on small record sizes.

Finally, as a last test in the string of writer tests we use a per-call lock/unlock, i.e., our exerciser performs a "`fcntl(); write(); fcntl();`" sequence where the first `fcntl` is called with an `F_WRLCK` argument while the last uses an `F_UNLCK` argument. Quite surprisingly, as shown in Fig. 28.6, a single GFS process writing experiences very low performance at small record sizes, mimicking what happens in Fig. 28.4 where processes in different nodes share the same file.

[2] For an explanation on why the size of the file under test was changed from "large" to "medium", see section 29.3.2.2 at Part IX, "Conclusion".

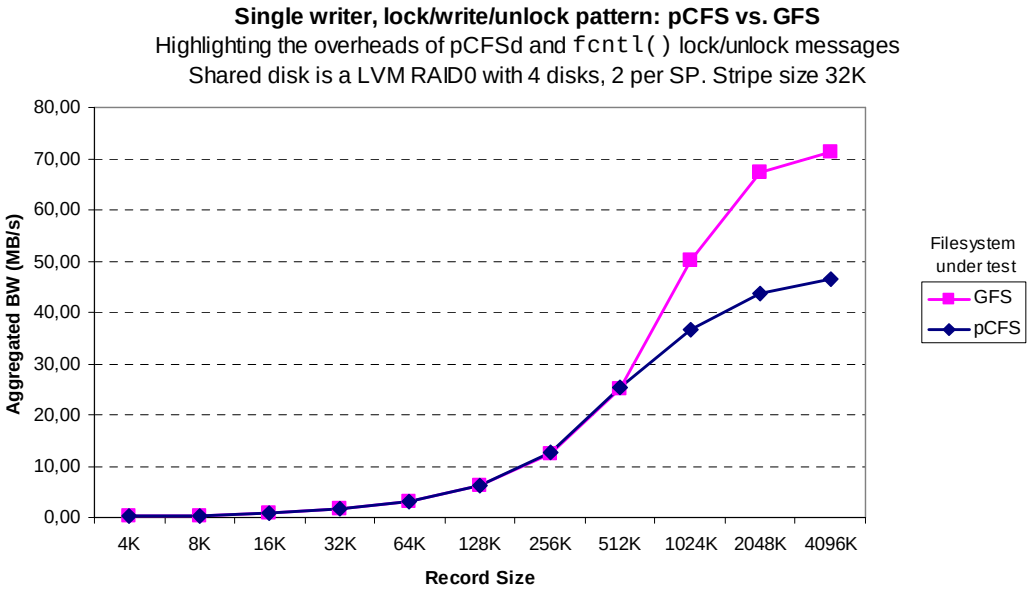


Figure 28.6 GFS and pCFS: writing with per-call locks

To search for the cause for this behaviour, we remounted the GFS filesystem on a single node (172.16.1.6), launched a single writer (starting with a record size of 4 MB and descending to 4 KB), and monitored the LAN traffic. Our conclusion, looking at Fig. 28.7, is that the time it takes for DLM to exchange messages among all nodes to support the `fcntl()` causes a start/stop behaviour that severely limits I/O bandwidth (here we have only included the graphs for node 172.16.1.5, but those for nodes .4 and .3 are identical).

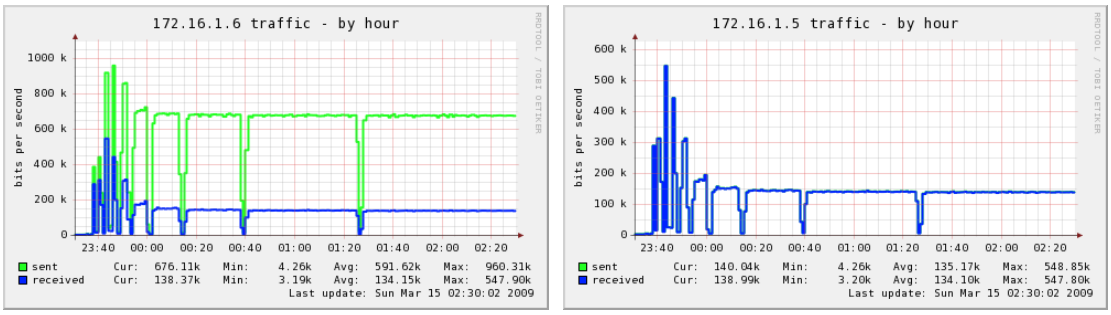
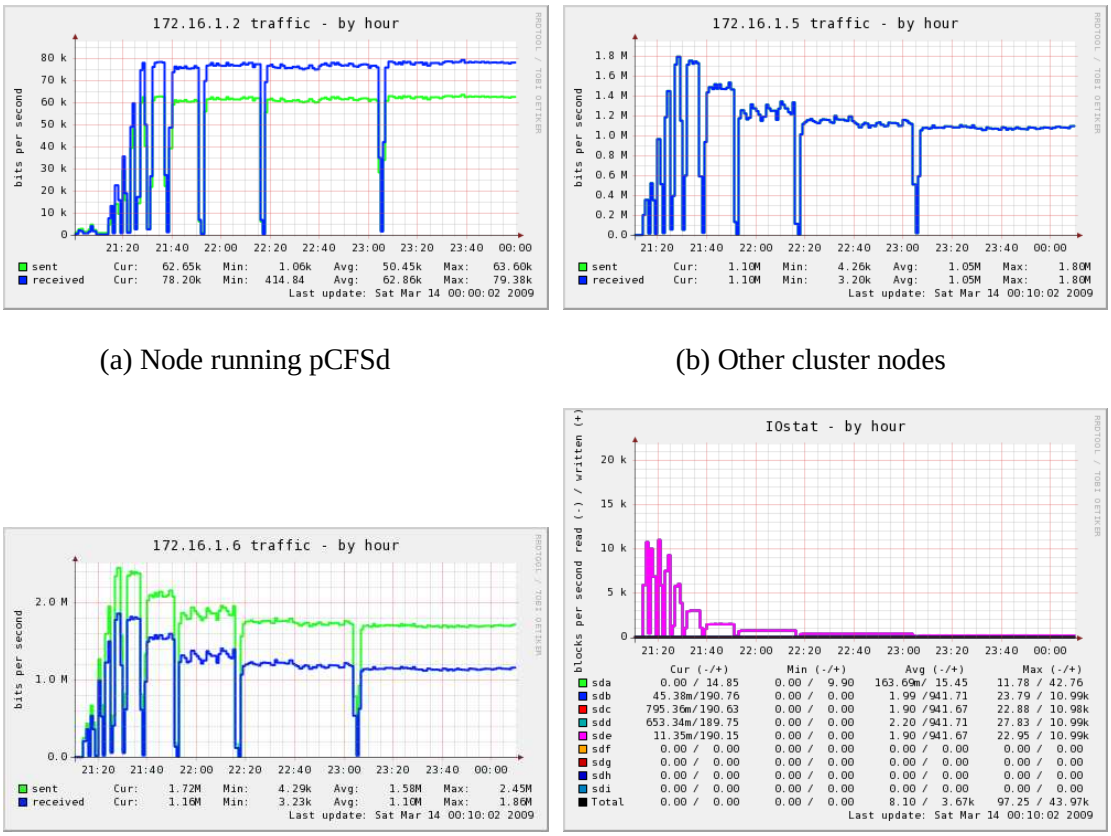


Figure 28.7 GFS: DLM traffic among nodes to support fcntl() calls

The same conditions were reproduced in order to perform a single writer test under pCFS: we started the pCFSd daemon on node 172.16.1.2, mounted the pCFS (GFS) filesystem on node 172.16.1.6, and launched a single writer on that same node; now, looking at Fig. 28.8 below, we can see that pCFS modifications have caused DLM traffic among nodes to increase by an order of magnitude (from 100 kbps to 1 Mbps), while pCFS traffic coming from the writer node (the pCFSk kernel module in .6) to pCFSd (.2) reaches about 80 kbps.



(a) Node running pCFSd

(b) Other cluster nodes

(c) Node where the filesystem is mounted and where the test was run

Figure 28.8 pCFS: pCFSd and DLM traffic to support fcntl() and write()

The tenfold increase in traffic among nodes does not influence the pCFS/GFS performance ratio for record sizes up to 512 KB, as we can see that pCFS follows exactly the same “line” as GFS (Fig. 28.6); however, for larger sizes, pCFS lags behind GFS, its performance getting progressively worse as buffer sizes are increased.

A major reason for pCFS’ performance loss with regard to GFS is the way pCFS (currently) maintains coherency: it forces a flush-to-disk operation each time a region is unlocked – something which, in this test, coincides with every write, so we have a per-write flush. As to what causes the increase in DLM traffic, the root cause is also related with the way coherency is implemented: as we force a “flush-to-disk” we also drop the Glock from the node’s cache, and this triggers more DLM messages across nodes.

28.4.2 Single writer/multiple readers tests

GFS single writer/multiple reader tests do exhibit the same type of behaviour as the segmented writer tests reported in the previous section, as they share the same root cause, the cluster-wide ginode lock; aggregated bandwidth is again quite small for record sizes under 128K, reaching a maximum of 30 MB/s for a record size of 4 MB (Fig. 28.9).

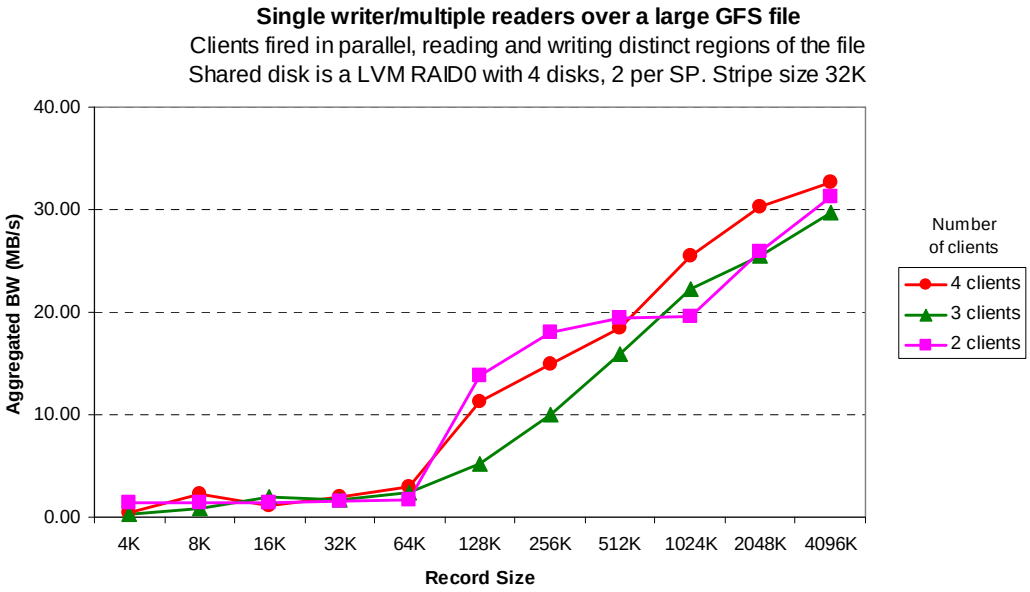


Figure 28.9 GFS: non-overlapping single writer/multiple readers

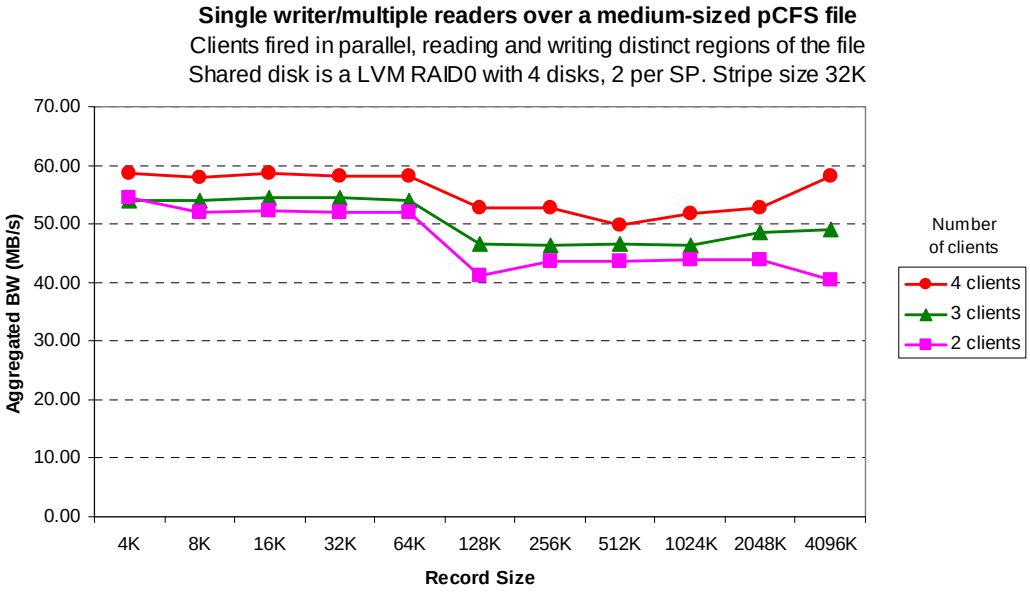


Figure 28.10 pCFS: non-overlapping single writer/multiple readers

Fig. 28.10 shows that, again, pCFS betters GFS by a large margin in this test; the performance increase for small record sizes is not so pronounced as it was for the writer/writer tests (there a 600 time difference between GFS and pCFS and here the difference is about 60 times) while for large record sizes it is, for both cases, twice the GFS bandwidth.

28.4.3 Segmented writing tests with block allocation

To investigate the influence of intra-file metadata operations on the overall performance of segmented writing, tests were run against an initially empty file; results under GFS show that these tests, as those carried out over NFS and PVFS, do offer slightly increased bandwidths with regard to those where writes were over previously allocated data blocks: under GFS without block allocation (Fig. 28.4) we got 35 MB/s for 4 clients when using a 4 MB buffer size, while the new test with block allocation runs at 40 MB/s, as seen in Fig. 28.11 below.

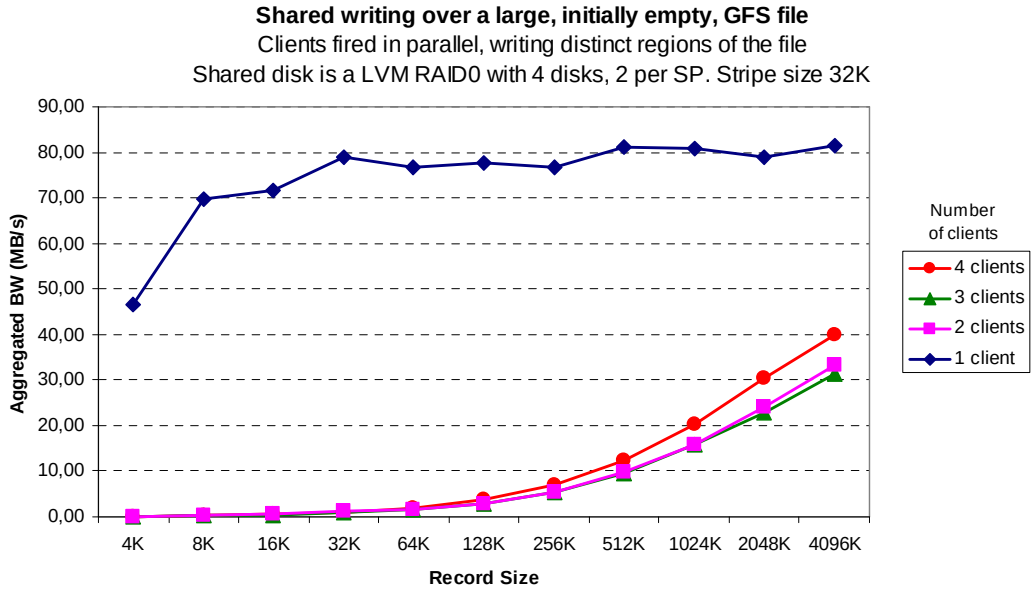


Figure 28.11 GFS: Segmented writing over a large, initially empty file

As for pCFS, our prototype does not yet support write sharing with metadata allocation; however, as previously referred, two different mechanisms can be made available: one which uses glock promotion to the exclusive state, pCFS thus behaving exactly as GFS does; the other which resorts to data shipping over the network. We expect the “glock promotion” path to deliver the same performance as GFS, i.e., its “test chart” will resemble that of Fig 28.11. As for the “data shipping” approach, we think its performance will be similar to NFS’, as displayed in Fig. 26.3.

28.5 Summarising results for pCFS and GFS

Tables 28.1 and 28.2 summarise the results for our GFS “cluster file system setup”; although they refer mainly to GFS, we note that values gathered in reader tests are also applicable to pCFS. Table 28.3 summarises the results for pCFS (although referring primarily to pCFS, the first column is also shared with GFS). As before, CPU usage is the observed worst case value, and occurs in tests where four writers write-share a file accessing it with a 4K record size.

	Readers (same as pCFS)				Writers				1 Writer/N readers			CPU usage
KB	1	2	3	4	1	2	3	4	1	2	3	
4	42.8	44.8	55.2	54.4	55.8	0.1	0.1	0.1	1.5	0.3	0.4	
8	43.0	44.9	54.7	54.2	59.1	0.2	0.2	0.2	1.4	0.9	2.3	
16	43.3	45.2	55.1	54.8	63.1	0.4	0.3	0.2	1.4	2.0	1.1	
32	42.8	44.9	54.9	54.3	63.4	0.8	0.8	0.8	1.5	1.7	2.0	
64	42.9	45.0	54.9	54.3	63.7	1.5	1.5	1.5	1.7	2.4	2.9	
128	42.7	44.7	47.4	54.2	63.6	2.8	2.7	2.7	13.9	5.2	11.3	14.0
256	42.6	44.8	54.9	54.2	63.5	5.4	5.4	5.4	18.0	9.9	14.9	
512	42.6	44.9	55.2	54.3	63.7	9.6	9.8	9.8	19.4	15.9	18.4	
1024	42.6	44.8	55.2	54.3	63.9	17.0	16.7	16.7	19.5	22.3	25.6	
2048	42.6	44.7	48.7	54.3	63.8	26.0	23.7	25.5	25.9	25.5	30.3	
4096	42.5	44.7	55.3	54.2	63.3	34.9	32.7	34.0	31.2	29.7	32.6	

Table 28.1 GFS tests, part 1

	Readers (full file scan)			
KB	1	2	3	4
4	42.8	69.0	64.4	93.5
8	43.0	69.2	64.5	84.6
16	43.3	69.3	64.3	93.6
32	42.8	69.0	64.5	85.1
64	42.9	68.7	64.1	94.2
128	42.7	68.6	64.1	93.9
256	42.6	50.0	64.0	85.0
512	42.6	68.5	63.9	84.4
1024	42.6	68.5	58.7	75.2
2048	42.6	68.5	63.9	89.2
4096	42.5	68.7	63.8	84.2

(a) Note: same as pCFS

	Writers (empty file)			
KB	1	2	3	4
4	46.5	0.1	0.1	0.1
8	69.6	0.2	0.2	0.3
16	71.7	0.5	0.4	0.5
32	79.0	1.1	0.8	1.1
64	76.7	1.7	1.5	2.0
128	77.7	2.9	2.8	3.7
256	76.8	5.5	5.3	6.9
512	81.0	9.8	9.4	12.4
1024	80.8	15.9	15.9	20.3
2048	78.9	24.0	22.8	30.4
4096	81.4	33.3	31.4	39.9

(b)

Table 28.2 GFS tests, part 2

	Readers (same as GFS)				Writers				1 Writer/N readers			CPU usage
KB	1	2	3	4	1	2	3	4	1	2	3	
4	42.8	44.8	55.2	54.4	55.8	59.5	60.0	60.2	54.4	53.9	58.7	
8	43.0	44.9	54.7	54.2	59.1	60.9	61.1	61.7	51.9	54.0	57.9	
16	43.3	45.2	55.1	54.8	63.1	60.7	61.6	62.0	52.2	54.4	58.7	
32	42.8	44.9	54.9	54.3	63.4	61.2	61.7	62.2	52.1	54.6	58.2	
64	42.9	45.0	54.9	54.3	63.7	61.1	61.9	62.1	52.1	54.0	58.2	
128	42.7	44.7	47.4	54.2	63.6	60.7	61.9	61.9	41.2	46.6	52.8	16.9
256	42.6	44.8	54.9	54.2	63.5	61.0	61.7	62.2	43.7	46.4	52.8	
512	42.6	44.9	55.2	54.3	63.7	60.8	61.7	62.0	43.6	46.5	49.7	
1024	42.6	44.8	55.2	54.3	63.9	61.2	61.6	62.0	43.8	46.4	51.7	
2048	42.6	44.7	48.7	54.3	63.8	61.0	61.6	61.8	43.9	48.5	52.8	
4096	42.5	44.7	55.3	54.2	63.3	61.2	61.7	62.2	40.4	49.1	58.3	

Table 28.3 pCFS segmented access tests

As for Table 28.4 below, it reports both the GFS and pCFS results for the single writer using a per-call lock/unlock, previously shown in Fig. 28.6 and which surprised us with its low performance.

Aggregated BW (MB/s)	Record Size (KB)										
	4	8	16	32	64	128	256	512	1024	2048	4096
GFS	0.2	0.4	0.8	1.60	3.10	6.2	12.5	25.2	50.3	67.3	71.4
pCFS	0.2	0.4	0.8	1.60	3.10	6.3	12.6	25.4	36.7	43.8	46.5

Table 28.4 Single writer with a lock/write/unlock pattern

28.6 Resource usage

The last step of this report on pCFS (and GFS) is to present the Munin graphs gathered during the four writer tests. In these graphs, given that the time to run each test (i.e., running all buffer sizes from 4 KB to 4 MB) was smaller due to the shortened size of the file, the three runs that were taken are clearly visible; only a single node is shown, all others being quite similar.

Figure 28.12 pCFS resource usage

28.7 pCFS and GFS: closing remarks

GFS performance may be quite insufficient for applications that require write sharing (i.e., at least one process is a writer) of a file among processes running in several nodes: when record sizes below 128K are used, bandwidth is less than 2 MB/s – something not far from the speed of a diskette! Also, there is no scalability, as adding nodes does not result in any sizeable bandwidth increase. GFS single node write performance, however, is quite good, at 63 MB/s, and we can get about half of it in multi-node write sharing if one uses very large record sizes, e.g., 4 MB. GFS read scalability is also very good but, unfortunately, limitations of the disk array we have used do not unmistakably allow us to show it – although we can get a glimpse, when we look at the full file scan reader tests reported in Table 28.2 (a).

pCFS delivers high performance sharing, bettering GFS by two times on very large record sizes – e.g., 4 MB records – while the results for small record I/O show gains of two orders of magnitude for write/write sharing and one order of magnitude for read/write sharing.

Part IX:

Conclusion

This Part assesses the benefits of pCFS – its use of an integrated approach to data movement, cooperative caching, and low latency cache coherence operations – and how they succeed in overcoming the I/O bottleneck. Finally, it introduces ideas for future work.

29 Conclusion

29.1 Revisiting the I/O bottleneck

It is a well known fact that a successful computing architecture is based on a suitable balance of three subsystems: processor, memory, and I/O (both storage and networking); however, we currently face a situation where performance of these subsystems (at least for off-the-shelf components) is increasing at very disparate rates, with a clear advantage on the processor side, and the storage being the worst performer. This requires system architects to foster new storage solutions, both in hardware and in software; for example, disk arrays have entered the mainstream and can now be found everywhere, from small ones, individually attached to a single host, to large ones, deployed in storage area networks and shared across multiple systems; they have become the basic "building block" solution to two problems: I/O performance and high availability.

But good performance of a single system may come at a very large cost, and cost is something that today is regarded to be of utmost importance; therefore, one continuously looks for better price/performance alternatives, and one of the best calls for the coordinated use of multiple computer systems – i.e., a cluster – as a platform to solve "bigger" problems in a cost efficient way. However, sharing data across multiple computing nodes creates new problems and, therefore, new solutions must be brought in, in the form of "distributed", "clustered", or "parallel" data base and file systems.

File systems for multi-node computer architectures have evolved across two separate tracks, much in the same way to what happened to distributed vs. shared memory: on one side, distributed-disk file systems were developed on the assumption that storage is based on disks which are private to the nodes and that the "global" filesystem vision is implemented by moving data across network interconnects; on the other side, file systems for shared-disk architectures assume that disks are shared across nodes and that the "global" filesystem vision is implemented by writing data to disk in a node and reading it on another.

29.2 Restatement of the objectives

As a preliminary step we have tried to characterise the somewhat fuzzy terminology used when discussing I/O on multi-node architectures, such as "parallel I/O", or "parallel", "cluster" and "distributed" file systems. As work progressed, we found that sometimes a unique concept was being handled as different things when we moved across layers (with boundaries not always clearly defined) while in other cases we found that the opposite was occurring, i.e., two different concepts where being subsumed in a single, not very clear one; therefore we propose a reference model that encompasses all layers from (but excluding) the

application down to the physical disk and, for the most relevant ones, i.e., File System, Object Storage and Storage Access layers, taxonomies are proposed.

Our pCFS proposal combines two previously divorced approaches, those of shared vs. distributed disks: it assumes a shared-disk architecture (where all nodes have shared access to all disk volumes), and implements a coherent global vision across nodes *either* through data movement across network interconnects *or* writing it to disk on a node and reading it from disk on another. We expected such an approach to have good performance while keeping full POSIX compliance, allowing pCFS to be used both for general as well as HPC applications in small to medium-sized clusters, up to, say, a hundred nodes directly attached to a SAN which caters for the cluster's shared storage.

The implementation of pCFS was carried out "on top of" Red Hat's GFS. Using synthetic benchmarks, we tested pCFS against GFS itself, and then against NFS and PVFS – two widely used file systems both in HPC as well as in more "general" file sharing environments (although both have drawbacks when used outside their primary target environments, e.g., NFS may be too slow for HPC use, and PVFS may be unsuited for some "file sharing" applications).

29.3 Assessment of the contributions

29.3.1 Reference model

Development of the "Reference Model for Data Management Architectures" was carried out along Part IV, with section 13 introducing the taxonomy for file system classification, which was used in section 15 to compare among several "classes" of distributed file systems, notably: symmetrical distributed file systems (GPFS and GFS); asymmetrical distributed file systems of the un-partitioned "single-server type" (NFS); and asymmetric partitioned "multiple-server types" (PVFS, AFS, DCE/DFS). Along with the model's proposal, precise definitions have been introduced.

29.3.2 pCFS

29.3.2.1 The proposal

pCFS was introduced in Part V, starting with its "conceptual" architecture, distinguishing features – most notably *cooperative caching* and *fine-grain locking*. Then, we presented the programmer's view of pCFS, a strict POSIX compliant file system where the *only* two things a programmer must do to choose pCFS behaviour is to add a single (new) flag to the file's `open()` call and, if appropriate, use standard POSIX `fcntl()` locks.

29.3.2.2 The implementation

The implementation of pCFS is described in Part VII; it calls for a single, clusterwide user level daemon, pCFSd, and two per-node kernel modules, pCFSk and pCFSc. Current

prototype limitations are a consequence of the decision to keep GFS data structures unmodified; this has (i) introduced more complexities at the code level, and (ii) deterred us from supporting VFS-initiated asynchronous operations, such as those triggered by the kernel VM subsystem to flush out file pages, resulting in failures when writing to very large files as the page cache gets "full" and VM triggers the Linux kernel daemons (e.g., `pdflush`) to flush them out to decrease memory pressure.

We feel confident, however, that we have proved that even the un-implemented features are viable, and that a production-grade version would be a very interesting file system to have for a broad range of applications.

29.3.2.3 The benchmarks

We are quite happy with the benchmark results; they unequivocally show that when sharing a file using "large regions" pCFS, from a point of view of:

1. **Aggregate Bandwidth**
   a. When compared to GFS:
      - Surpasses GFS in all tests involving write sharing of a single file, delivering from **twice** up to a **600 times** increase in BW.
      - Matches GFS in all tests involving only readers.
   b. When compared to NFS:
      - Surpasses NFS in all tests, nearly **doubling** its performance.
   c. When compared to PVFS:
      - Performs better than PVFS for small number of clients (< 4).
      - Betters PVFS for all write-sharing situations (i.e., write/write or read/write)
      - Surpasses or runs close to PVFS in the full file scan readers test
      - Is quite insensitive to changes in the buffer size
2. **Aggregate CPU usage**
   a. When compared to GFS:
      - Uses about the **same** fraction of CPU (pCFS worst-case is 17% while GFS' is 14%).
   b. When compared to NFS:
      - Worst-case NFS uses the **same** CPU but its BW is about 2.3 times smaller.
   c. When compared to PVFS:
      - Worst-case PVFS draws circa **155%** (for a third of the pCFS' BW).

As we have seen, current results are not so shiny for access patterns which require a per--call lock/unlock, as bandwidth is too low for record sizes which are smaller than 128 KB, being in the region of 0.2 MB/s (for 4K records) to 6.3 MB/s (for 128 KB records). The solution for this problem may require i) an application rewrite, or ii) some "hint-based" approach which would be capable of converting the small region pattern into a large region one, or, finally iii) the data shipping approach which, when fully implemented, may result in bandwidths that are closer to those available from NFS, but still far from the pCFS bandwidths for "large region" accesses, at 60 MB/s.

29.4 Future work

Future work on pCFS may progress on two separate tracks: a standalone version for regular Linux kernels and a specialised version for kernel-level DSM (or VSM[1]) Linux kernels.

29.4.1 Standalone version for regular Linux kernels

Continuing the development of the pCFS into a more robust, production level standalone with version is feasible as short term task, (i.e., it is not a research project); it would differ from the current prototype in minor aspects, such as:

- Small changes to GFS data structures
  - For example, the ginode could carry a pCFS flag; this would allow us to test for a "pCFS inode" without accessing the VFS file structure, something that can only be referenced when executing in a user context (and not in, e.g., a daemon context, such as in `pdflush`).
- Use of the TIPC kernel subsystem[2] for all communication tasks
  - This allows much better failure handling; perform recovery; use of broadcast and/or multicast; establishment connections on demand, etc.
  - We may, therefore, dispense with the `pCFSd` forwarding.
- Merge `pCFSk` and `pCFSc` into a single, multithreaded kernel module
  - Using kernel abstractions such as kernel threads and work queues.
  - Increasing the level of concurrence both in intra-node, inter-node, and node-to-daemon (`pCFSd`) operations.

The outcome should be a production-level pCFS version; there are, however, longer term tasks that should also be carried out in the standalone version, such as: (i) providing a fully implemented, clusterwide cooperative cache, one which can be used to provide multiple paths for data transfer (therefore enhancing performance) whereas in the current prototype we just maintain caches coherent across nodes; and, (ii) use it as a way to further enhance the high availability of the file system, via inter-cache replication of modified pages.

29.4.2 pCFS on DSM Linux kernels

A pCFS version supported over a kernel-level DSM/VSM, such as Kerrighed, would be a longer term, research driven, project; some of the answers that such a project must provide are:

- Can the DSM-provided consistency mechanisms be used as the sole basis for clusterwide page coherency? Is the performance acceptable?
- Should the global Page Cache be the sole "user" of the DSM mechanisms, or should these be applied to all file system objects (and caches) such as inodes, dentries, etc., therefore either re-implementing glocks as DSM-based objects?

[1] As previously noted, we use the term DSM for hardware-aided distributed shared memory and VSM for pure software implementations

[2] We thank the Kerrighed/Kerlabs team for introducing us to TIPC in the Kerrighed Summit'08 (some topics on the panel discussions available on www.kerlabs.com/docs/Kerrighed_summit_08/)

- Should these new objects be implemented on a per-filesystem basis, or should one try to apply these concepts to VFS and re-implement it as a clusterwide layer (as proposed in kDDM [Leb+08]) so that any currently available filesystem that plugs into VFS can be made available to all cluster nodes?

29.5 New avenues for pCFS

Longer term research on pCFS will focus on investigating pCFS' adequacy to efficiently support the shared disk / shared file system paradigm over wide area networks, e.g., in cluster federations (with dedicated fibre links).

We believe that the cooperative cache mechanism, which has proved its usefulness in distributed file systems, will be a major driver for pCFS' success on these environments, as it can be the topmost layer that supports three extremely important aspects: caching, replication and fault tolerance.

Acronyms

ADT	Abstract Data Type
ACID	Atomicity, Consistency, Isolation, Durability
AFS	Andrew File System
API	Application Programming Interface
BW	Bandwidth
cc-NUMA	Cache Coherent Non-Uniform Memory Architecture
CAS	Content Addressable Storage
CEFT-PVFS	Cost Effective, Fault Tolerant, Parallel Virtual File System
CFS	Cluster File System (concept)
CI	Cluster Infrastructure
CIFS	Common Internet File System
CM	Cache Manager (pCFS)
COMA	Cache-Only Memory Architecture
COTS	Common Off-The-Shelf
CPU	Central Processing Unit; here used as synonym for processor
CRC	Cyclic Redundancy Check
CVFS	Comprehensive Versioning File System
DAS	Direct Attached Storage
DASD	Direct Access Storage Device
DBL	Database Layer (RM-DMA)
DBMS	Database Management System
DCE	Distributed Computing Environment (from the Open Software Foundation)
DCE/DFS	Distributed File System (integrated with DCE)
DD	Distributed Disks (a.k.a. PD)
dentry	Directory entry (VFS)
DFS	Distributed File System (either a concept or an Intel Paragon FS)
dinode	Disk inode (on-disk image of a GFS inode)
DLM	Distributed Lock Manager
DMA	Direct Memory Access
DMD	Data Management Domain (RM-DMA)
DMEP	Device Memory Export Protocol (GFS)
DML	Data Management Layer (RM-DMA)
DOS	Distributed Operating System
DSM	Distributed Shared Memory (hardware-based)

EVMS	Enterprise Volume Manager System
ext2	Extended File System, version 2 (a.k.a. Second Extended File System)
ext3	Extended File System, version 3
FAT	File Allocation Table
FC	Fibre Channel
FLOPS	Floating-point Operations per Second
FS	File System
FSB	Front Side Bus
FSL	File System Layer (RM-DMA)
GbE	Gigabit Ethernet
G-Lock	Global Lock (GFS)
GDLM	GFS Distributed Lock Manager (GFS)
GFS	Global File System
ginode	GFS inode (in-core image, linked into the VFS vnode)
glock	an abbreviation for G-Lock (see G-Lock)
glops	an abbreviation for a G-Lock vector of operations
GPFS	General Parallel File System
GULM	Grand Unified Lock Manager (GFS)
HA	High Availability
HA-NFS	High Availability Network File System
HA-PVFS	High Availability PVFS
HBA	Host Bus Adapter
HDF	Hierarchical Data Format
HiPPI	High Performance Parallel Interface
HPC	High Performance Computing
IB	Infiniband
INCITS	International Committee for Information Technology Standards
inode	Information node
I/O	Input/Output
IOPS	I/O operations per second
ISAM	Indexed-Sequential Access Method
iSCSI	IP-based SCSI
IT	Information Technology
JBOD	Just a Bunch Of Disks
LAN	Local Area Network
LD	Logical Disk (a.k.a. LV)
LDAP	Lightweight Directory Access Protocol

LM	Lock Manager
lmlock	Abbreviation of Lock-Manager lock (GFS)
LMM	Lock Manager Module (GFS)
LUN	Logical Unit (a.k.a. Logic Unit Number)
LV	Logical Volume (a.k.a. LD)
LVB	Lock Value Block (see GFS)
LVM	Logical Volume Manager (concept)
LVM	Logical Volume Manager (Red Hat LVM)
MM	Memory Management
MPIO	Multi Path I/O
MPP	Massively Parallel Processor
NAS	Network Attached Storage
NASD	Network Attached Secure Disks
NetCDF	Network Common Data Form
NFS	Network File System
NFSP	Netware File Sharing Protocol
NIC	Network Interface Card
NIS	Network Information Service (a.k.a. Yellow Pages)
NLM	Network Lock Manager (NFS)
NOS	Network Operating System
NoW	Network of Workstations
NSD	Network Shared Disks (GPFS)
NSM	Network Status Monitor (NFS)
NTFS	New Technology File System
NUMA	Non-Uniform Memory Architecture
OBSD	Object-Based Storage Device (a.k.a. OSD)
OCFS	Oracle Clustered File System
OLAP	On-Line Analytical Processing
OLTP	On-Line Transaction Processing
OO	Object Oriented
OOM	Out Of Memory
OS	Operating System
OSD	Object Storage Device (a.k.a. OBSD)
OSF	Open Software Foundation
OSF/1	Open Software Foundation’s Operating System/1
OSL	Object Storage Layer (RM-DMA)
PADFS	Partitioned-Asymmetric Distributed File System (RM-DMA)

PC	Personal Computer
PCB	Printed Circuit Board
pCFS	Parallel Cluster File System
PCI	Peripheral Component Interconnect
PD	Partitioned Disks (a.k.a. DD)
PFS	Parallel File System (Intel)
PoP	Pile of PCs
POSIX	Portable Operating System Interface Architecture
PVFS	Parallel Virtual File System
QoS	Quality of Service
RAID	Redundant Array of Inexpensive (a.k.a. independent) Disks
RDBMS	Relational Data Base Management System
RDMA	Remote DMA
RG	Resource Group
RM	Reference Model
RM-DMA	Reference Model for Data Management Architectures
RPC	Remote Procedure Call
SAL	Storage Access Layer (RM-DMA)
SAN	Storage Area Network
SCI	Scalable Coherent Interface
SCSI	Small Computer System Interface
SD	Shared Disk
SDL	Storage Device Layer (RM-DMA)
SDM	Shared Disk Manager (pCFS)
SMD	Storage Management Domain (RM-DMA)
SMP	Shared Memory Multiprocessor (unless otherwise noted)
SNIA	Storage Networks Industry Association
SNL	Storage Network Layer (see RM-DMA)
SP	Storage Processor
SSI	Single System Image
SVL	Storage Virtualisation Layer (RM-DMA)
TIPC	Transparent Inter-Process Communication
TOE	TCP Offload Engine
UFS	UNIX File System
UMA	Uniform Memory Architecture
vnode	Virtual node (VFS)
VFS	Virtual File System (a.k.a. Virtual Filesystem Switch)

VM	Virtual Memory
VSD	Virtual Shared Disk
VSM	Virtual Shared Memory (software-based)
WAN	Wide Area Network
WCC	Weak Cache Consistency
XDR	External Data Representation

References

[Ada+04] Adabala, S. *et al*. Single Sign-On in In-VIGO: Role-Based Access via Delegation Mechanisms Using Short-Lived User Identities. Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04), 26-30 April 2004, Santa Fe, New Mexico, USA.

[Ada+05] Adabala, S. *et al*. From Virtualized Resources to Virtual Computing Grids: The In-VIGO System. Journal of Future Generation Computer Systems, Vol. 21, Issue 6 (June 2005), pp. 896-909.

[Aga95] Agarwalla, R. DFS Token Manager Redesign. Open Software Foundation Request For Comments: 73.0, October 1995.

[Amd67] Amdahl, G. The validity of the single processor approach to achieving large scale computing capabilities. Proceedings of the Spring AFIPS Joint Computing Conference, volume 30, 1967, pp. 483-485 (electronic version by Guihai Chen, with additional notes).

[And+95] Anderson, T. *et al*. The Case for NOW (Networks of Workstations). IEEE Micro, Vol. 15, No. 1, 1995, pp. 54-64.

[And+96] Anderson, T. *et al*. Serverless Network File Systems. ACM Transactions on Computer Systems, Vol. 14, No. 1, Feb 1996.

[And96] Anderson, T. DFS Changes to Support a Scalar 64-bit type. Open Software Foundation Request For Comments: 51.3, August 1996.

[And99] Anderson, D. Object Based Storage Devices Presentation to T10. ANSI/T10 Technical Committee, Document No. 99-340R0, Oct 1999. www.t10.org

[Bar+98] Barak, A. and La'adan O. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. Journal of Future Generation Computer Systems (13) 4-5, March 1998, pp. 361-372.

[Bar+99] Barak, A. and La'adan O. Scalable Cluster Computing with MOSIX for Linux. Proceedings of the Linux Expo '99, Raleigh, N.C., May 1999, pp. 95-100.

[Bar+00] Barry, A. *et al*. SCSI Device Memory Export Protocol, version 0.9.8. Sistina Software Inc., Oct 2000.

[Ben+02] Benoit, G. and Hersch, R. Comparing Multimedia Storage Architectures. In High Performance Mass Storage and Parallel I/O. Jin, H., Cortes, T. and Buyya, R., Editors. IEEE Press, 2002, pp. 548-554.

[Ber+94] Berrendorf, R. *et al*. Intel Paragon XP/S - Architecture, Software Environment, and Performance. Forschungsezentrum Julich GmbH, Zentralinstitut fur Angewandte Mathematik, KFA-ZAM-IB-9409, 1994.

[Bha01] Bhatt, A. Creating a Third Generation I/O Interconnect. Intel® Developer Network. www.intel.com/technology/pciexpress/devnet/comms.htm

[Bha+03] Bhattacharya, S. *et al*. Asynchronous I/O Support in Linux 2.5. Proceedings of the Linux Symposium. July 23th–26th, 2003, Ottawa, Ontario, Canada.

[Bhi+91] Bhide, A. *et al*. A Highly Available Network File Server. Proceedings of the Winter USENIX Conference, 1991, pp 199-206.

[Bli+04] Bligh, M. *et al*. Linux on NUMA Systems. Proceedings of the Linux Symposium. July 21st–24th, 2004, Ottawa, Ontario, Canada.

[Bor+05] Bornträger, C. and Schwidefsky, M. Providing Linux 2.6 support for the zSeries platform. IBM Systems Journal, Vol. 44, Issue 2, Jan. 2005, pp. 331-340.

[Bov+05] Bovet, D. and Cesati, M. Understanding the Linux Kernel (3rd Edition, Nov. 2005). O'Reilly.

[Bra03] Braam, P. The Lustre storage architecture. Technical Report, Cluster File Systems, Inc 2003. Available: www.lustre.org

[Cad+08] Cadavez, T. *et al*. Graphical Tool for the Tomographic Characterization of Microstructural Features on Metal Matrix Composites. To appear in the International Journal of Tomography & Statistics.

[Cal00] Callaghan, B. NFS Illustrated. Addison-Wesley, 2000.

[Cap+03] Capps, D. *et al*. Iozone Filesystem Benchmark. Available, http:// www.iozone.org.

[Car+00] Carns, P *et al*. PVFS: A Parallel File System for Linux Clusters. Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, 2000, pp. 317-327.

[Cha98] Charlesworth, A. Starfire: Extending the SMP Envelope. IEEE Micro, Vol. 18, No. 1, Jan./Feb. 1998, pp. 39-49.

[Chi+06] Chinner, D. and Higdon, J. Exploring High Bandwidth Filesystems on Large Systems. Proceedings of the Linux Symposium, July 19th–22nd, 2006 Ottawa, Ontario, Canada.

[Chi+07] Chin, A. *et al*. Non-contiguous Locking Techniques for Parallel File Systems. Proceedings of the International Conference for High Performance Computing Networking, Storage, and Analysis 2007 (SC 07), November 10-16 2007, Reno, USA.

[Cor+96] Corbet, P. and Feitelson, D. The Vesta Parallel File System. ACM Transactions on Computer Systems (TOCS) Volume 14, Issue 3 (August 1996), pp. 225-264.

[Cor+02] Corbett, P. *et al*. Overview of the MPI-IO Parallel I/O Interface. In High Performance Mass Storage and Parallel I/O. Jin, H., Cortes, T. and Buyya R., editors. John Wiley & Sons, 2002, pp. 477-487.

[Cra+04] Crandal, P. *et al*. I/O Characterization and Analysis. In Scalable Input/Output, Achieving System Balance. Daniel Reed, Editor. The MIT Press, 2004, pp. 1-33.

[Dob03] Dobson, M. *et al*. Linux Support for NUMA Hardware. Proceedings of the Linux Symposium. July 23rd–26th, 2003, Ottawa, Ontario, Canada.

[Don+01] Dongarra, J. *et al*. The LINPACK Benchmark: Past, Present, and Future. Concurrency: Practice and Experience, 15, 2003 pp. 803-820.

[Dre+05] Drepper, U. and Molnar, I. The Native POSIX Thread Library for Linux. RedHat, Inc., Feb 2005. http://people.redhat.com/drepper/nptl-design.pdf

[EMCa06] EMC. Centera Content-Addressed Storage Product Description Guide. EMC Corporation. http://www.emc.com/products/systems/centera/pdf/C938.6_Centera_PDG_ldv.pdf

[EMCb06] EMC. Invista. EMC Corporation, Data Sheet H1437.2, Nov. 2006.

[Fac+05] Factor, M. *et al*. Object Storage: the future building block for storage systems. IEEE International Symposium on Local to Global Data Interoperability - Challenges and Technologies, June 20th–24th, 2005 Sardinia, Italy, pp. 119-123.

[Fas06] Fasheh, M. OCFS2: The Oracle Clustered File System, Version 2. Proceedings of the Linux Symposium, July 19th–22nd, 2006 Ottawa, Ontario, Canada.

[FC-FCP] ANSI. SCSI-3 Fibre Channel Protocol (X3.269:1996).

[Fos+97] Foster, I. and Kesselman, C. Globus: A Metacomputing Infrastructure Toolkit. The International Journal of Supercomputer Applications and High Performance Computing, Vol. 11, No. 2, Summer 1997, pp. 115-128.

[Fos+01] Foster, I. *et al*. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International Journal of High Performance Computing Applications, Vol. 15, No. 3, Fall 2001, pp. 200-222.

[Fos05] Foster, I. Globus Toolkit Version 4: Software for Service-Oriented Systems. Proceedings of the IFIP International Conference on Network and Parallel Computing, Beijing, China, Nov. 30 – Dec. 3, 2005, Springer-Verlag LNCS, Vol. 3779, pp. 2-13.

[Fre+01] Frey, J. *et al*. Condor-G: A Computation Management Agent for Multi-Institutional Grids. Proceedings of the 10th IEEE Symposium on High Performance Distributed Computing (HPDC10) San Francisco, California, August 7-9, 2001.

[Ganglia] Ganglia. ganglia.sourceforge.net

[Gan+03] Ganek, A. and Corbi, T. The dawning of the autonomic computing era. IBM Systems Journal, Vol. 42, Issue 1, Jan. 2003, pp. 5-18.

[Gar+05] Gara, A. *et al*. Overview of the Blue Gene/L system architecture. IBM Journal of Research and Development, Vol. 49, No. 2/3, Mar/May 2005, pp. 195-212.

[Gib+98] Gibson, G. *et al*. A Cost-Effective, High-Bandwidth Storage Architecture. Proceedings of the 8th International Conference on Architectural support for programming languages and operating systems (ASPLOS'98), 1998, San Jose, California, pp. 92-103.

[Gif+91] Gifford, D. *et al*. Semantic File Systems. Proceedings of the 13th ACM Symposium on Operating Systems Principles, Asilomar, Pacific Grove, CA, Oct. 1991, pp. 16-25.

[Gor04] Gorman, M. Understanding the Linux Virtual Memory Manager. Bruce Perens' Open source series, Pearson Education, Inc. (publishing as Prentice Hall Professional Technical Reference), 2004.

[Grid5000] Grid'5000. www.grid5000.fr

[Gus88] Gustavson, J. Reevaluating Amdahl's law. Communications of the ACM, Vol. 31, No. 5, May 1988, pp. 532-533.

[Gus92] Gustavson, D. The Scalable Coherent Interface and related standards projects. IEEE Micro, Vol. 12, No. 1, Feb 1992, pp. 10-22.

[Hae+83] Haerder, T. and Reuter, A. Principles of Transaction-Oriented Database Recovery. ACM Computing Surveys, Vol. 15, No 4, December 1983.

[Her04] Hertel, C. Implementing CIFS: The Common Internet File System. Bruce Perens' Open source series, Pearson Education, Inc. (publishing as Prentice Hall Professional Technical Reference), 2004.

[Hil+06] Hildebrand, D. *et al*. pNFS and Linux: Working Towards a Heterogeneous Future. University of Michigan CITI Technical Report 06-06, June, 2006.

[Hog+02] Hogan, E. *et al*. Examining Semantics In Multi-Protocol Network File Systems. Carnegie Mellon University Technical Report CMU-CS-02-103, January, 2002.

[How+88] Howard, J. *et al*. Scale and Performance in a distributed file system. ACM Transactions on Computer Systems (TOCS) Volume 6, Issue 1 (February 1988) pp. 51-81.

[How95] Howes, T. The Lightweight Directory Access Protocol: X500 Lite. Center for Information Technology Integration (CITI), University of Michigan, 1995.

[Hug+05] Hugh-Jones, R. *et al*. Performance of 1 and 10 Gigabit Ethernet Cards with Server Quality Motherboards. Journal of Future Generation Computer Systems (FGCS), Vol. 21 Nº 4, 2005, pp. 469-488.

[IBM-07] IBM Corporation. Tuning IBM System x Servers for Performance. Fifth Edition, February 2007, Order No.: SG24-5287-04.

[IBMa06] IBM Corporation. General Parallel File System: Concepts, Planning, and Installation Guide. First Edition, April 2006, Order No.: GA76-0413-00.

[IBMafs] IBM Corporation. AFS documentation. www.ibm.com/software/stormgmt/afs/library/

[IBMb06] IBM Corporation. General Parallel File System: Administration and Programming Reference. First Edition, April 2006, Order No.: SA23-2221-00.

[IBTA01] Infiniband Trade Association. InfiniBand™ Architecture Specification, Vol. 1. Infiniband Trade Association, 2001. www.infinibandta.org

[IEEE92] IEEE Std 1596-1992. IEEE standard for scalable coherent interface (SCI). Aug. 1993.

[IEEE04] The IEEE and The Open Group. "IEEE Std 1003.1, 2004 Edition" or "The Open Group Base Specifications Issue 6", available online at www.opengroup.org

[Jes05] Jessel, D. AMD Opteron™ Processors: A Better High-End Embedded Solution (*whitepaper*). AMD Boston Design Center, 2005. www.hypertransport.org

[Kan+01] Kannan, S. *et al*. Workload Management with LoadLeveler. IBM RedBook SG24-6038, 2001.

[Kaz+88] Kazar, M. *et al*. Synchronization and Caching Issues in the Andrew File System. ACM Transactions on Computer Systems, Vol. 6, 1988, pp. 51-81.

[Kle86] Kleiman, S. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. Proceedings of the Summer USENIX Conference, Atlanta, 1986, pp 238-247.

[Kon+05] Kongetira, P. *et al*. Niagara: A 32-way multithreaded Sparc processor. IEEE Micro, Vol. 25, Issue 2, 2005, pp. 21-29.

[Kru+02] Krueger, M. Small Computer System Interface protocol over the Internet (iSCSI) Requirements and Design Considerations (RFC 3347). July 2002.

[Leb06] Lèbre, A. aIOLi: Contrôle, Ordonnancement et Régulation des Accès aux Données Persistantes dans les Environnements Multi-applicatifs Haute Performance. Thèse Doctorale, Institut National Polytechnique de Grenoble, Septembre 2006.

[Leb+08] Lèbre, A. *et al*. Reducing kernel Development Complexity in Distributed Environments. Proceedings of the 14th Euro-Par Conference, Las Palmas, Spain, August 2008.

[Lee+96] Lee, E. *et al*. Petal: distributed virtual disks. Proceedings of the 7th International conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, Massachusetts, United States, 1996, pp. 84-92.

[Len+95] Lenoski, D. and Weber, W. Scalable Shared-Memory Multiprocessing. Morgan Kaufmann, 1995.

[Lev+90] Levy, E. and Silberchatz, A. Distributed File Systems: Concepts and Examples. ACM Computing Surveys, Volume 22, Issue 4 (December 1990) pp. 321-374.

[Lew05] Lewis, A. LVM How To. Red Hat, Inc., 2005. www.tldp.org/HOWTO/LVM-HOWTO

[Li+86] Li, K. and Ross, R. Parallel I/O and the Parallel Virtual File System. In "Beowulf Cluster Computing with Linux, 2nd Edition", Gropp, W. and Lusk, E., editors. The MIT Press, 2003, pp. 493-534.

[Lig+03] Ligon, W. and Hudak, P. Memory coherence in shared virtual memory systems. Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing, Calgary, Alberta, Canada, 1986, pp. 229-239.

[Lop+05] Lopes, P. and Medeiros, P. pCFS: A Parallel Cluster File System. Proceedings of the ParCO 2005 Conference, Málaga, Spain, Sep 13-16 2005.

[Lop+06] Lopes, P. and Medeiros, P. Cooperative Caching in the pCFS parallel Cluster File System. Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC), Paris, France, Jun. 19-23 2006.

[Lop+08] Lopes, P. and Medeiros, P. Enhancing write performance of a shared-disk cluster filesystem through a fine-grained locking strategy. Proceedings of the Second International Workshop on High Performance I/O Systems and Data Intensive Computing (HiperIO'08), held within IEEE International Conference on Cluster Computing 2008, Sep. 29th Oct 1st 2008.

[Lor+05] Lorentz, C. *et al*. EVMS User Guide, 2005. http://evms.sourceforge.net/user_guide/

[Lot01] Lottiaux, R. Gestion globale de la mémoire physique d'une grappe pour un système à image unique: mise en œuvre dans le système GOBELINS. Thèse Doctorale, Université de Rennes 1, Décembre 2001.

[Lov03] Love, R. Interactive Kernel Performance. Proceedings of the Linux Symposium. Jul. 23rd-26th, 2003, Ottawa, Ontario, Canada.

[Mad+04] Madhyastha, T. *et al*. Informed Prefetching of Collective Input/Output Requests. In Scalable Input/Output, Achieving System Balance. Daniel Reed, Editor. The MIT Press, 2004, pp. 135-160.

[McC95] McCalpin, J. Memory bandwidth and Machine Balance in Current High Performance Computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, Dec. 1995, pp. 19-25.

[McK+84] McKusick, M. *et al*. A Fast File System for UNIX. ACM Transactions on Computer Systems, Vol. 2, 1984.

[Mes+03] Mesnier, M. *et al*. Object-based storage. IEEE Communications Magazine, Vol. 41, Issue 8 (August 2003), pp. 84-90.

[Mor+04] Morin, C. *et al*. Towards an efficient single system image cluster operating system. Journal of Future Generation Computer Systems, Vol. 20, Issue 4, 2004, pp. 505-521.

[Munin] Munin. http://munin.projects.linpro.no

[Myr00] Myricomm. S. Myrinet 2000 Serial Link. www.myri.com/open-specs/

[Nai04] Naik, D. Inside Windows Storage: Server Storage Technologies for Windows 2000, Windows Server 2003, and Beyond. Addison-Wesley, 2004.

[Nan+95] Nanette, B. *et al*. Myrinet: A Gigabit-per-Second Local Area Network, IEEE Micro, Vol. 15, No. 1, 1995, pp. 29-36.

[NCSA99] NCSA HDF Development Group. HDF User's Guide. www.hdfgroup.org

[Ndi+04] Ndiaye, B. *et al*. A Quantitative Comparison between Raw Devices and File Systems for implementing Oracle Databases, Oracle/HP whitepaper, April 2004.

[Nie+96] Nieuwejaar, N. and Kotz, D. The Galley Parallel File System. Proceedings of the 10th International Conference on Supercomputing, Philadelphia, Pennsylvania, United States, 1996, pp. 374-381.

[Nit+95] Nitzberg, B. and Fineberg, S. Parallel I/O on Highly Parallel Systems, SC'95 Tutorial M6, December 4, 1995.

[Nor+96] Norman, M. *et al*. Much Ado About Shared-Nothing, ACM SIGMOD Record, Vol. 25, Issue 3 (September 1996), pp. 16-21.

[OCFS] Oracle Cluster File System (OCFS) Projects. http://oss.oracle.com/projects/ocfs/ and also http://oss.oracle.com/projects/ocfs2/

[OG-DCE] Open Group. DCE Bookstore. www.opengroup.org/bookstore/catalog/dz.htm

[Ols+99] Olson, M. *et al*. Berkeley DB. Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, Monterey, California, USA, June 6–11, 1999.

[openAFS] openAFS. openAFS documentation. www.openafs.org/doc/index.htm

[openGFS] OpenGFS Project. http://opengfs.sourceforge.net

[openPBS] openPBS Project. www.openpbs.org

[Pai+99] Pai, V. *et al*. IO−Lite: A Unified I/O Buffering and Caching System. Proceedings of the 3rd Symposium on Operating Systems Design and Implementation New Orleans, Louisiana, February, 1999.

[PARIS] INRIA. PARIS: Programming distributed parallel systems for large scale numerical simulation. www.inria.fr/recherche/equipes/paris.en.html

[Pat+89] Patterson, D. *et al*. A Case for Redundant Arrays of Inexpensive Disks (RAID). Proceedings of the 1989 ACM-SIGMOD International Conference on the Management of Data, ACM, 1989, pp. 109-116.

[Paw+94] Pawlowski, B. *et al*. NFS version 3 design and implementation. Proceedings of the Summer USENIX Conference, June 1994, pp 137-152.

[PCI-X] PCI-SIG. PCI-X 2.0 Overview. Peripheral Component Interconnect Special Interest Group (PCI-SIG), www.pcisig.com.

[Pfi01] Pfister, G. Aspects of the InfiniBand™ Architecture. Proceedings of the 2001 IEEE International Conference on Cluster Computing (CLUSTER'01), Newport Beach, CA, USA, Oct. 8-11, 2001.

[Pra02] Pratt, S. EVMS: A Common Framework for Volume Management. Proceedings of the Ottawa Linux Symposium, June 26-29, 2002, Ottawa Canada, pp 451-458.

[Pre+99] Preslan, K. *et al*. Device locks: Mutual exclusion for storage area networks. Proceedings of the Seventh NASA Goddard Conference on Mass Storage Systems and Technologies in cooperation with the Sixteenth IEEE Symposium on Mass Storage Systems, San Diego, CA, March 1999.

[Raj+07] Rajagopalan, M. *et al*. Thread scheduling for multi-core platforms. Proceedings of the 11th USENIX workshop on Hot Topics in Operating Systems, 2007, San Diego, CA.

[RFC1813] RFC 1813. NFS version 3 Protocol Specification.

[Rid+97] Ridge, D. *et al*. Beowulf: Harnessing the power of parallelism in a pile-of-pcs. Proceedings, IEEE Aerospace, Vol. 2, February 1997, pp. 79-91.

[Rie+01] Riedel, E. *et al*. Active Disks for Large-Scale Data Processing, IEEE Computer, Vol. 34, Issue 6, 2001, pp. 68-74.

[Ril06] Rilling, L. Vigne: Towards a Self-healing Grid Operating System. Proceedings of the 12th International Euro-Par Conference, Dresden, Germany, 2006, pp. 437-447.

[Rod+05] Rodriguez, C. *et al*. The Linux Kernel Primer: a Top-Down Approach for x86 and PowerPC Architectures. Prentice-Hall PTR, 2005.

[Sal96] Salz, R. DCE 1.2 Contents Overview. Open Software Foundation Request For Comments: 63.2, May 1996.

[San+85] Sandberg, R. *et al*. Design and Implementation of the Sun Network Filesystem. Proceedings of the Summer USENIX Conference, Portland, 1985, pp 119-130.

[SAS-1] ANSI/INCITS. Serial Attached SCSI (SAS) (ANSI/INCITS 376-2003).

[Sch+99] Schikuta, E. and Stockinger, H. Parallel I/O for Clusters: Methodologies and Systems. In High Performance Cluster Computing, Volume 1. Rajkumar Buyya, Editor. Prentice Hall PTR, 1999, pp. 439-462.

[Sch+02] Schmuk, F. and Haskin, R. GPFS: A Shared-Disk File System for Large Computing Clusters. Proceedings of the Conference on File and Storage Technologies (FAST'02), 28–30 January 2002, Monterey, CA, pp. 231–244.

[SCSI-1] ANSI. Small Computer System Interface (X3.131-1986).

[SCSI-2] ANSI. Small Computer System Interface-2 (X3.131-1994).

[SNIA03] Storage Networks Industry Association. The SNIA Shared Storage Model (2003). www.snia.org/tech_activities/shared_storage_model/

[Sol97] Soltis, S. The Design and Implementation of a Distributed File System based on Shared Network Storage. PhD thesis, University of Minnesota Graduate School. Minneapolis, MN, August 1997.

[Sou+03] Soules, C. *et al*. Metadata Efficiency in Versioning File Systems. Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST), San Francisco, CA, 2003, pp. 43-58.

[Sto98] Stockinger, H. Dictionary on Parallel Input/Output. Master's Thesis, Institute for Applied Computer Science and Information Systems, Department of Data Engineering, University of Vienna, Austria, Feb. 1998.

[Sun02] Sun Microsystems, Inc. ONC+ Developer's Guide. Part No: 816-1435-10, 2002.

[T10-04] INCITS T10. SCSI Object-Based Storage Device Commands (OSD). T10/1355-D10, International Committee for Information Technology Standards (INCITS), July 2004.

[Tan92] Tanenbaum, A. Modern Operating Systems. Prentice-Hall International Editions, 1992.

[Tat+06] Tate, J. *et al*. IBM System Storage SAN Volume Controller. IBM RedBook SG24-6423, Sep. 2006.

[Tha+04] Thakur, R. *et al*. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Memorandum No. 234, Mathematics and Computer Science Division, Argonne National Laboratory, May 2004.

[TIPC] The Transparent Inter-Process Communication (TIPC). tipc.sourceforge.net/index.html

[Tol+03] Tolia, N. *et al*. Opportunistic Use of Content Addressable Storage for Distributed File Systems. Proceedings of the 2003 USENIX Conference, June 2003, pp 127-140.

[Uni06] Unidata. The NetCDF User's Guide. www.unidata.ucar.edu

[Van+00] Vanel, L. *et al*. AIX Logical Volume Manager, from A to Z: Introduction and Concepts. IBM RedBook SG24-5432-00, 2000.

[Van+07] Vangal, S. *et al*. An 80-Tile 1.28 TFLOPS Network-on-Chip in 65 nm CMOS. IEEE Solid-State Circuits Conference, 11-15 Feb. 2007, San Francisco, CA. Digest of Technical Papers, pp. 98-99,589.

[Vil+04] Vilayannur, M. *et al*. On the performance of the POSIX I/O interface to PVFS. Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'04), 11-13 Feb. 2004, pp. 332-339.

[VITA98] VITA Standards Organization. Myrinet-on-VME Protocol Specification Draft Standard, VITA 26-199x Draft 1.1 31 August 1998. www.myri.com/open-specs/

[Whi07] Whitehouse, S. The GFS2 Filesystem. Proceedings of the Linux Symposium, June 27th-30th 2007, Ottawa, Canada.

[Won+02] Wong, T. and Wilkes, J. My cache or yours? Making storage more exclusive. Proceedings of the 2002 USENIX Conference, 10-15 June 2002, Monterey, CA., USA, pp 161-175.

[Wu+04] Wu, J. *et al*. Unifier: Unifying Cache Management and Communication Buffer Management for PVFS over InfiniBand. Proceedings of IEEE/ACM International

Symposium on Cluster Computing and the Grid (CCGrid 04), Apr. 19-22, 2004, Chicago, Illinois, USA.

[Xtreem] XtreemOS. www.xtreemos.org

[Zhu02] Zhu, Y. Design, Implementation and Performance Evaluation of a Cost-Effective, Fault-Tolerant Parallel Virtual File System. MSc Thesis, The Graduate College at the University of Nebraska, December 2002.