Inaugural-Dissertation

zur
Erlangung der Doktorwürde
der
Naturwissenschaftlich-Mathematischen Gesamtfakultät
der
Ruprecht-Karls-Universität Heidelberg

vorgelegt von
Diplom-Informatiker Christoph Biardzki
aus Thorn

Tag der mündlichen Prüfung: 19.1.2009

Analyzing Metadata Performance in Distributed File Systems

Gutachter: Prof. Dr. Thomas Ludwig

Abstract

Distributed file systems are important building blocks in modern computing environments. The challenge of increasing I/O bandwidth to files has been largely resolved by the use of parallel file systems and sufficient hardware. However, determining the best means by which to manage large amounts of metadata, which contains information about files and directories stored in a distributed file system, has proved a more difficult challenge.

The objective of this thesis is to analyze the role of metadata and present past and current implementations and access semantics. Understanding the development of the current file system interfaces and functionality is a key to understanding their performance limitations. Based on this analysis, a distributed metadata benchmark termed DMetabench is presented.

DMetabench significantly improves on existing benchmarks and allows stress on metadata operations in a distributed file system in a parallelized manner. Both intra-node and inter-node parallelity, current trends in computer architecture, can be explicitly tested with DMetabench. This is due to the fact that a distributed file system can have different semantics inside a client node rather than semantics between multiple nodes.

As measurements in larger distributed environments may exhibit performance artifacts difficult to explain by reference to average numbers, DMetabench uses a time-logging technique to record time-related changes in the performance of metadata operations and also protocols additional details of the runtime environment for post-benchmark analysis.

Using the large production file systems at the Leibniz Supercomputing Center (LRZ) in Munich, the functionality of DMetabench is evaluated by means of measurements on different distributed file systems. The results not only demonstrate the effectiveness of the methods proposed but also provide unique insight into the current state of metadata performance in modern file systems.

The thesis concludes with a discussion of the results and the identification of areas for future research.

Zusammenfassung

Verteilte Dateisysteme sind wichtige Bausteine moderner IT-Umgebungen. Während parallele Dateisysteme und verbesserte Hardware die Geschwindigkeit des Datenzugriffs bereits erfolgreich verbessert haben, bleibt eine optimale Verwaltung von Metadaten, also den Informationen über Dateien, Verzeichnisse und deren Organisation, eine schwierige Herausforderung.

Das Ziel dieser Dissertation ist es zunächst, die Rolle der Metadaten in Dateisystemen zu erörtern und sowohl historische als auch aktuelle Implementierungen sowie Zugriffssemantiken darzustellen. Ein genaues Verständnis der Entwicklung von Dateisystemen und ihren Schnittstellen ist notwendig, um ihre Leistungsbeschränkungen im Bereich der Metadaten zu ergründen. Auf Basis dieser Untersuchungen wird der verteilte Metadaten-Benchmark DMetabench präsentiert.

DMetabench verbessert bereits existierende Benchmarks und ermöglicht eine gezielte, parallele Erzeugung von Metadaten-Operationen in verteilten Dateisystemen. Sowohl die Parallelität innerhalb, auch auch zwischen Rechenknoten – beides aktuelle Trends bei Rechner- und Systemarchitekturen – können gezielt getestet werden. Dies ist insofern wichtig, als dass verteilte Dateisysteme zwischen unterschiedlichen Knoten oft eine andere Semantik anbieten, als innerhalb einer Betriebssysteminstanz.

Da Messungen in verteilten Umgebungen Performanceartefakte verursachen können, die mit Durchschnittswerten schwer zu erklären sind, bietet DMetabench eine Zeitintervallbasierte Protokollfunktion, die zeitabhängige Änderungen der Geschwindigkeit von Metadaten-Operationen aufzeichnet. Weiterhin werden auch Details der Systemkonfiguration automatisch protokolliert.

Die Funktionalität von DMetabench wurde mit Hilfe von Messungen an großen, verteilten Produktions-Dateisystemen am Leibniz-Rechenzentrum (LRZ) in München überprüft. Die Ergebnisse zeigen nicht nur die Wirksamkeit der vorgestellten Techniken, sondern geben auch einzigartige Einblicke in den Stand der Technik bei der Metadatenleistung von verteilten Dateisystemen.

Abschliessend werden die Ergebnisse der Arbeit diskutiert und interessante Aufgaben sowie Probleme für zukünftige Untersuchungen vorgestellt.

Acknowledgements

This work would not have been possible without the continuous support of Prof. Thomas Ludwig, who supervised my research and whose expert advice not only saved me from making many beginner's mistakes but also greatly improved quality. He and his support team at the University of Heidelberg have created a comfortable, efficient and very friendly environment for my research activities.

I would also like to thank all my colleagues at the Leibniz Supercomputing Center in Munich for their strong support during the course of my research. I would like to thank Reiner Strunz for his invaluable insights into AFS. The entire Linux team, particularly Mike Becher, Bernhard Aichinger, Peter Simon and Dr. Reinhold Bader, not only provided constant advice and assistance but also bravely endured and resolved the countless interference of my research with the production systems. Here I would also like to thank Dr. Herbert Huber and Dr. Horst-Dieter Steinhöfer, who gave permission to use LRZ resources and supported me in every imaginable way. Many other co-workers at the LRZ also contributed to this thesis, whether they know it or not. A big thank you to all of you – without your help, I would never have finished my thesis.

Table of Contents

Chapter 1

Motivation

A paradigm shift to massively distributed computing environments has sparked new interest in scalable storage solutions. Although distributed file systems have been used for many years, they were only scaled for specific applications, such as file serving in desktop environments, for much of their existence. Only recently did parallel file systems enable the creation of large file systems with 100s of Terabytes. However, determining how to cope with millions or billions of files in an efficient manner remains a challenge.

1.1 A very short history of file systems

From their introduction, the first truly universal computers, such as the IBM System/360, offered different kinds of storage devices, such as tape drives and magnetic drums. However, because they had no file systems every programmer was responsible for placing his or her data on the storage devices. As Amdahl et al. explained in "Architecture of the IBM System/360" [ABFPB64]:

> It was decided to accept the engineering, architectural, and programming disciplines required for storage-hierarchy use. The engineer must accommodate in one system several storage technologies, with separate speeds, circuits, power requirements, busing needs, etc. [...] The system programmer must contend with awkward boundaries within total storage capacity and must allocate usage. He must devise addressing for very large capacities, block transfers, and means of handling, indexing across and providing protection across gaps in the adressing sequence.

As every programmer had to develop his or her own own data storage formats, the exchange of data between different applications was awkward at best [Cla66]. The appearance of the first true file system can be traced back to the introduction of the MULTICS operating system, as described by Daley [DN65]:

> This formulation provides the user with a simple means of addressing an essentially infinite amount of secondary storage in a machine-independent and device independent fashion. The basic structure of the file system is independent of machine considerations. Within a hierarchy of files, the user is aware only of symbolic addresses. All physical addressing of a multilevel complex of secondary storage devices is done by the file system, and is not seen by the user.

An important break through for file systems was the development of the first version of the UNIX operating system in the 1970s. UNIX utilized ideas from MULTICS and the developers Ritchie and Thompson were well aware of the role of file systems [RT78]:

> The most important job of UNIX is to provide a file system.

1.2 File system concepts

The file system implementation in UNIX became a model for many future file systems. The basic unit of storage is a *file*, defined as an ordered sequence of data bytes. Files can be addressed by the means of a symbolic *file name*. *Directories* act as containers for files and other directories and can be used to build a hierarchy (see Fig. 1.1).

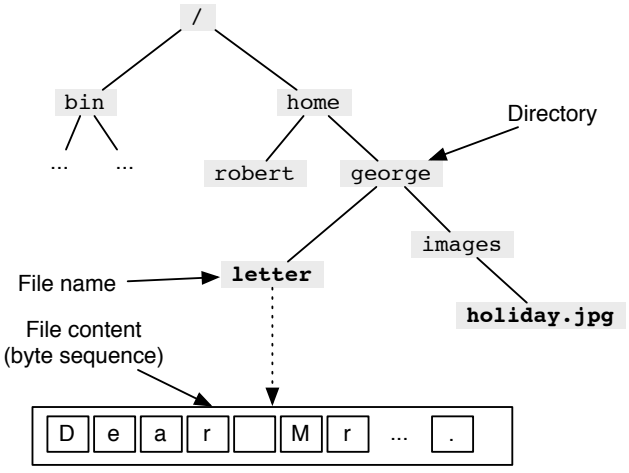

Figure 1.1: A hierarchical file system

Any data inside a file is opaque to the file system in that it does not interpret the information therein. There is no concept of fine-granular structures inside a file: this task depends upon applications. Storing and manipulating files and directories constitute the basic functions of a file system. These operations are part of the operating system and made available to the application programmer by standard library functions.

Traditional file systems* map the abstraction of directories, files and file contents to physical storage devices such as hard disks, which consist of equally sized disk blocks†. To maintain the relationship between the file content and its corresponding physical storage as well as the relationship between the file and directory hierarchy, a file system needs to keep track of additional information called the *metadata*. Managing metadata is the true mission of a file system.

1.3 Distributed file systems and metadata

The replacement of large, monolithic computer systems by smaller, networked PCs and workstations in the mid-1980s, together with a paradigm shift to client-server computing, led to the development of *distributed file systems*. Early distributed file systems such as Novell's "Netware" and Sun's "NFS" enabled access to a local file system of dedicated servers from a multitude of client computers. The main driving force behind this development was the enabling of simple data exchange between separate computers and different programs and users.

As environments grew larger, distributed file systems such as AFS were designed to use multiple servers [HKM+88]. Client computers were required to identify the right server for a particular file and the file system was required to provide additional information about this relationship. This was also a special kind of file system metadata.

A decade later, parallel file systems revolutionized the world of high-performance computing. Whereas each file had previously been stored on a single server, a parallel file system could distribute a file to a multitude of servers. This system leveled the load on the servers and enabled both larger files and faster, distributed access. This process also required that the file system stores additional metadata to be able to locate the server responsible for a part of a file.

The crucial consideration in these developments is that the application program interface to the file system *has not evolved* since the 1970s: the same function calls once responsible for accessing file system data on a 5 MB hard disk are still being used for distributed file systems capable of multi-GB/s throughput on Terabyte-sized files. Of course, transparency is a desirable property for distributed systems, but while considerable efforts have been made to make file systems larger and improve throughput, the true goals – increasing the speed and parallelizing metadata operations – are still being pursued.

*Often called *local file systems.*

†A disk block, also called a*sector*, usually has 512 bytes, which will probably increase to 4,096 byte sectors in the future[Gro06].

1.4 Objectives

A main objective of this thesis is to provide a structured overview of data and metadata operations and semantics in distributed file systems, with special consideration of why metadata performance is especially important in distributed file systems and why improvements in this area are hard to achieve.

A distributed metadata benchmark (DMetabench) is presented that can stress and measure the performance of chosen metadata operations at the file system level. This benchmark framework is then used to analyze different implementations in common distributed file systems by means of measuring metadata performance. The use of a variety of different metadata performance and scalability measurements on large production filesystems is then discussed.

1.5 Structure of the thesis

The thesis is structured in the following manner: Chapter 2 summarizes basic information regarding file system concepts and terms and presents standardized data and metadata operations and their semantics in the context of distributed file systems. It then discusses the motivation behind different operational semantics with regard to targeted distributed environments. It concludes by classifying metadata operations by location dependence and the amount of metadata involved.

Chapter 3 begins with a discussion of previous and related work in the area of metadata benchmarking. Based on the analysis presented in Chapter 2, the framework for a distributed metadata benchmark is designed. A special consideration is large-scale parallelism with a comparison of measurements on computer clusters to SMP-machines and the ability to retrospectively trace the preconditions of benchmark execution. The structure of the framework, its extendibility and the analysis of results is then presented.

Chapter 4 presents different measurements for production file systems, demonstrates the features of DMetabench and then discusses and compares the performance properties of different filesystems.

Chapter 5 concludes the thesis by providing a summary and an outlook on related research opportunities.

Chapter 2

Metadata in Distributed File Systems

An overview of filesystem terms and concepts is presented in the following chapter. Different concepts for distributed file systems are presented and basic models for access sematics and data durability are discussed. Finally, current trends and challenges for file system metadata are presented.

2.1 File system basics

2.1.1 Definitions

Many terms in the field of file systems have, unfortunately, been overloaded with several context-dependent meanings. Frequently the generic concept of a hierarchical file system conflicts with the details of a particular but widespread implementation – the UNIX file system – as described in a common standard IEEE 1003.1 of the American Engineers Association IEEE and the Open Group. This standard, which describes a common operating system environment, is commonly referred to as POSIX. As POSIX defines the behavior of file systems in many UNIX-like enviroments such as AIX, Solaris and Linux it will be subsequently be referenced when UNIX implementation is considered.

File systems

A *file system* is a method of storing and managing information in a computer system using files and directories. *Local file systems* utilize block-based storage devices attached to the computer system. *Virtual file systems* present information from other data sources by means of a file system structure. A particular kind of virtual file system is a *distributed file system*, where the information can be stored on a non-local networked computer.

Depending up on the context in which the term *filesystem* is used, the concept may refer to a particular technical implementation or a specific instance of a particular implementation.

Data and Metadata

In a file system *data* is stored inside of files. The set of all management information necessary for the filesystem to operate is the *metadata*. Management information internal to the filesystem is the *internal metadata* while *external metadata* is made available to higher layers by means of an interface (eg. attributes or directory structures).

Files

A *file* is a linear sequence of data bytes identified by a symbolic address. Files are usually referenced by *file names*, but many file systems contain an additional indirection between the file name and the symbolic address of the file. POSIX, for example, uses a system-wide unique file number termed an *inode number*. Every file exhibits a set of attributes that are either maintained by the file system (e.g., the file size) or defined explicitly (e.g., access rights). The inner structure of file contents, if any, is not defined; the interpretation is the sole task of the application programs, as the file system only understands operations on sequences of bytes.

Directories

Directories consist of a set of directory entries which associate symbolic names with references to files and other directories. The names of directory entries in a particular directory are unique. Referencing other direcories (*subdirectories*) creates a tree-like structure with directories as nodes and files or empty directories as leaves. Every filesystem has a special root directory that makes up the root of the tree. In POSIX directories, entries consist of pairs of *filenames* and *inode numbers*. Every directory has two additional entries with special names: `dot` (.) points to the direcory itself and `dot-dot` (..) to the parent directory. The root directory is the only directory where *dot* and *dot-dot* both point to the directory itself.

Path names

A *path name* is the result of concatenating the names of all directories on the path from the root directory to a file. The components of the path name are separated by a *path separator*.* The relation between path names and the respective files and directories is surjective, as several paths can point to the same file.

Links

Links are special directory entries that enable multiple references to files and directories, and thus the use of multiple file and path names. Because of links, the file system tree

*POSIX uses a forward slash (`/`) as in `/usr/local/bin`.

becomes a directed graph that is not necessarily acyclic. Not every file system supports link functionality.

Two link types supported by POSIX are *hardlinks* and *symbolic links (symlinks)*. Hardlinks, which are all directory entries for a file over and above the first one, directly reference the inode number. Technically, *dot* and *dot-dot* are also hardlinks. Other hardlinks to directories are usually not permitted because of the danger of creating cyclic references. Hardlinks are parts of internal metadata and cannot be used to reference files outside of a single file system.

Symlinks can point to both files and directories by means of their path names and span multiple file systems. Applications must handle softlinks carefully because cyclic references and non-existient link targets are possible to create.

Attributes

Files, directories and softlinks[†] exhibit *attributes*. All file systems define a set of standard attributes. Some implementations also allow for additional attributes using a key-value pattern (*extended attributes*). Extended attributes store arbitrary data (the *value* of the attribute) attached to a symbolic *key*. The standard attributes defined by POSIX are shown in Table 2.1.

Table 2.1: Standard POSIX file attributes.

Attribute	Description
`st_dev`	Devine number of the device containing the file
`st_ino`	Inode number
`st_mode`	File mode
`st_nlink`	Number of hardlinks
`st_uid`	User ID
`st_gid`	Group ID
`st_rdev`	Device ID
`st_size`	File size
`st_atime`	Last access timestamp
`st_mtime`	Last modification timestamp
`st_ctime`	Last status change timestamp
`st_blksize`	Block size
`st_blocks`	Number of allocated blocks

Mount points

The concept of a hierarchical structure can be used to merge several file systems into one virtual file system. Starting with a *root file system*, the root directories of additional file systems are *mounted* on arbitrarily chosen directories of the root file system (*mount points*). The

[†] All hardlinks point to the same inode number and thus the same file with identical attributes.

process of mounting makes the mounted file system appear a subdirectory of the parent file system.

2.1.2 Integration with operating systems

File system implementations are usually tightly integrated into the operating system. Applications access the file system exclusively by means of a well-defined set of system calls that are then either executed inside the system kernel or, as is the case with some distributed file systems, forwarded to a user space file system daemon.

Virtual File System

Operating systems usually support several different local and distributed file system implementations. To separate the common interface from the implementation a virtual file system switch (*VFS*) is used. This concept was first introduced and implemented in Sun OS in 1986 [Kle86]. The VFS allows different file system implementations to co-exist by forwarding the file system-dependent parts of system call to the appropriate file system, depending upon the type of file system. In addition, the VFS can manage physical I/O so that the implementation of a local file system is reduced to providing the mapping from file to physical blocks, as it does in Linux.

Caching infrastructure

Caching data and metadata is very important in reducing the impact of disk and, in the case of distributed file systems, network latencies imposed by file system operations. Although the caching infrastructure is highly dependent upon the particular operating system, in most cases the general idea is to buffer recently accessed data in memory that is not being used by applications.

In Linux, for example (Fig. 2.1), every file system manages its own cache inside a common cache infrastructure called the buffer cache. In local file systems the buffer cache is also responsible for performing I/O-operations into storage devices. Distributed file systems can also use the buffer cache but their I/O-operations are partly or entirely performed via the network protocol stack.

To improve the performance of directory operations the VFS can also maintain a cache of mappings from file names to their corresponding file system-internal structures. In Linux, this task is performed by the directory entry (*dentry*) cache. Although there is a default implementation of the dentry cache specific filesystems can overload the defaults. An internal `d_revalidate()` callback function is provided and called every time the dentry is used in order to determine whether it is still valid. Because the file system can change without the

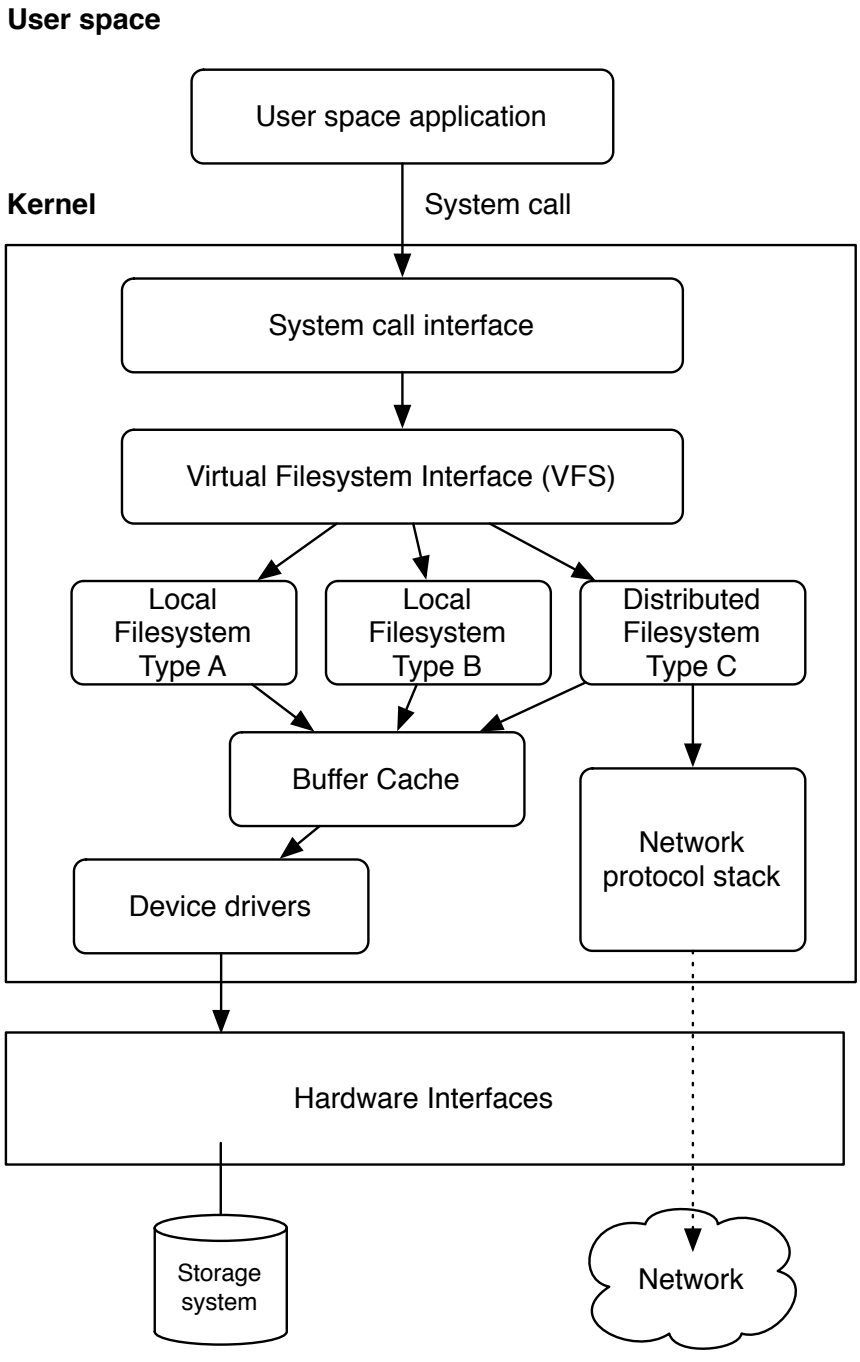

Figure 2.1: VFS in the operating system (Linux [CTT04])

awareness of the local VFS in distributed file systems, the local implementation can provide a means of ensuring fresh data.

2.2 Data operations

Data and metadata stored in a file system can be accessed using a basic set of system calls provided by the operating system kernel. Table 2.2 presents a summary of important data operations.

2.2.1 System calls

An `open()` system call, which takes the path name and associates the calling process with the corresponding file, is required to be able to work with data in a file. For this purpose, the operating system maintains a table of opened files for every process. Files can be opened using different access modes, such as reading and/or writing. A file can also be created if it did not previously exist. As a result of `open()` a *file handle* that contains a reference information necessary for all subsequent operations is returned.

The *file pointer* indicates the position within the byte sequence of the file where `read()` and `write()` operations will take place. To change the current position the `lseek()` operation is available, and to change the length of the file `ftruncate()` can be used.

Data operations work in the following manner. With a freshly created file the file pointer shows position 0. A `write()` operation writes data beginning at this position and, if necessary, extends the length of the file. After every write access the file pointer moves by the number of bytes written. To read data the file pointer is set using `lseek()`, and `read()` fetches the designated number of bytes and moves the file pointer. Both `write()` and `read()` return the number of bytes processed, which can differ from the intended number of bytes because of error conditions (e.g., being at the end of a file when reading or having a full file system when writing).

A file pointer can be set beyond the current length of the file. After a subsequent write access, the space between the previous end of the file and the current position is filled with zero bytes. Some file systems are able to save such files using holes in a space-efficient manner (*sparse files*). The semantics of sparse files are such that when a sparse section is read, a zero-filled result set is returned.

The counterpart to `open()` is `close()`, which removes the association between the process and the files and invalidates the file handle. If a process terminates, the operating system closes all open files automatically.

Table 2.2: File data operations

Signature	Description
open(path, access mode) → filehandle	Open file
close(filehandle)	Close file
write(filehandle, data) → count	Write data
read(filehandle, data) → count	Read data
seek(filehandle, offset, type) → new position	Change file position
ftruncate(filehandle, offset)	Change file length

2.2.2 Data operations and cache control

In some cases applications need to bypass the file system cache or flush it explicitly. The `fsync()` call returns only after all data and metadata connected with the given file descriptor are written to disk. Alternately the `O_SYNC` access mode, which guarantees that all `write()` operations work synchronously, can be used.

`O_DIRECT` is a similar access mode that bypasses the file system cache for writes and reads and is used by applications performing their own caching (e.g., databases). Disabling the cache provided by the operating system avoids some overhead caused by copying data to the cache and also provides synchronous write semantics. Unlike other access modes `O_DIRECT` can require the access size to be a multiple of a file system block or disk sector size.

2.3 Metadata operations

Metadata operations are responsible for changes in and queries on files, directories, and their attributes. Table 2.3 summarizes the basic metadata operations.

2.3.1 System calls

Directories can be created using the system call `mkdir()`. `Unlink()` can be used to remove a file, for directories `rmdir()` is available. Following the corresponding C-Library function a `remove()` call is available that combines both. POSIX specifies that an `unlink()` call on an open file removes the directory entry immediately; however the file itself is deleted only if the entry was the last directory entry for this file and the last process had closed the file.[‡] The function `rename()` combines `link()` and `unlink()` in a way that enables moving a directory entry between different directories, even in systems that do not allow `link()`/`unlink()` on directories. Analogous to files, directories can be read to deter-

[‡]This property is often used for creating temporary files in UNIX.

mine their contents. For this purpose, `opendir()` and `readdir()` are available. By using `opendir()`, a directory can be opened for reading, and `readdir()` can iteratively return directory entries that consist of the entry type and a name.

Table 2.3: Operations on directories and links

Signature	Description
mkdir(path, permission mode)	Create directory
rmdir(path)	Delete directory entry for a directory
unlink(path)	Delete directory entry for a file
remove(path)	Delete directory entry
link(path1, path2)	Create new directory entry (hardlink)
rename(path1, path2)	Rename/Move file or directory
symlink(path1, path2)	Create a symbolic link

File attributes and permission control

POSIX defines several standard file attributes (see Table 2.1) that can be determined using `stat()` and modified with `chmod()`, `chown()`, and `utimes()`. Some file systems also support *extended attributes* based on a key-value pattern.

Checking file permissions operates in the following manner. Every file and directory exhibits 9 status bits that determine the permissions `r` (read), `w` (write), and `x` (execute) for the three classes of *user*, *group*, and *others*. Because these three classes are disjoint an accessing process is assigned to exactly one of them, depending upon its own user ID. Access permissions are checked inside particular system calls, such as `open()`. Details of the procedure and its extension using access control lists (ACLs) are described in detail in [Grü03].

From the perspective of metadata operations, a specific requirement of POSIX is particularly important: To be able to open a file in a directory, a process needs the `x`-permission on all directories that comprise the file path. This means that the entire path must be checked every time a file is opened.

The three timestamps for last status change, last modification, and last access to a file are special attributes automatically maintained by the operating system when correspondent system calls such as `write()` are issued. Numerous programs, particularly backup software, rely upon a correct modification timestamp.

2.3.2 Locks

The purpose of a lock is to guarantee exclusive access to file data and, by doing so, protect data consistency. A lock can be granted for an entire file or a byte range inside of the file. POSIX defines two functions for manipulating locks: `lockf` and the control function

Table 2.4: Operations on file attributes

Signature	Description
stat(path) → attribute	Read attributes
chmod(path, permission mode)	Set permission bits
chown(path, owner, group)	Set owner and group
utimes(path, timestamp)	Set timestamps
listxattr(path) → list of attributes	Read extended attributes
getxattr(path, attr. key) → attribute value	Read extended attribute
removexattr(path, attr. key	Remove extended attribute
setxattr(pfad, attr. key, attr. value)	Set extended attribute

`fcntl`.[§] The interaction between the two functions is not defined and as both offer only *advisory locks*, only one should be used at a time. Processes that use neither are able to freely access the locked file. Some operating systems, such as Linux, also offer *mandatory locks* that control all processes.[¶] However, mandatory locks are problematical in UNIX, as even the superuser is unable to override them, providing opportunities for a denial of service. Unlike UNIX, in the Microsoft Windows family of operating systems, which orginally started as single-user systems, mandatory locks are used extensively. Thus, the sharing of data between Windows and UNIX-based systems can lead to unexpected outcomes, as neither UNIX applications nor their users expect to encounter mandatory file locks.

Both `lockf` and `fcntl` offer checking and setting locks and both at once (*test-and-set*). The function `fcntl` has two kinds of locks: the *read-lock*, which bars other processes from obtaining a *write lock*, which in turn allows exclusive access for reading and writing. Several processes can hold read locks but only one can obtain a write lock. When a process terminates, all its locks are released.

2.4 Basic architecture of local disk filesystems

A closer look at the architecture of the original Unix file system (UFS) is helpful in gaining understanding of the file system interface and the compatibility design decisions which that influence the development of distributed file systems.

2.4.1 UFS: The UNIX file system

Disk file systems provide a mapping of files, directories, and their attributes to fixed-size disk sectors. Three basic internal metadata structures are the superblock, inodes, and direc-

[§] BSD-UNIX introduced a third function called `flock` that is comparable to an `fcntl`-lock.
[¶] Mandatory locks must be explicitly enabled when mounting the file system.

tories. In UFS (see Fig. 2.2), a given block device is partitioned into separate fixed areas for the superblock, inodes, and data blocks.

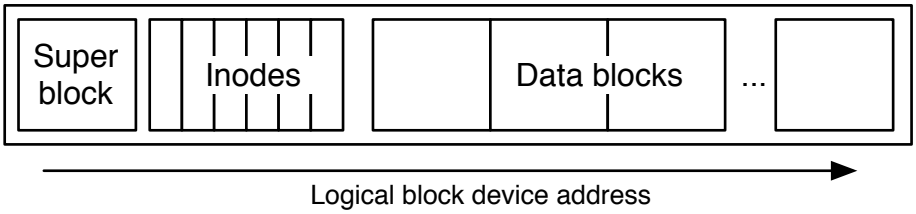

Figure 2.2: Block device partitioning with UFS

The superblock contains summary information about the file system, including the size of the pointers to the inode table. By using the superblock, the existence of a file system can be verified and the other structures can be located.

Inodes (see Fig. 2.3) are on-disk structures holding file attributes and pointers to data blocks for a file. Depending upon the size of the file, either direct or indirect pointers are used. Direct pointers contain the block device address of a data block while indirect pointers point to data blocks that contain additional pointers. The objective is to reduce the amount of metadata for smaller files. As inodes are created when the file system is formatted their number, and thus the number of files in the filesystem, is fixed.

Directories are stored as normal files and marked with a directory attribute. They consist of a list of directory entries that relate a file name to the corresponding inode number that can be used to locate an inode and the file (or other directory).

UFS stores all unused blocks in a file system inside a special hidden file. Data blocks are removed when other files increase in size and returned when files are deleted or truncated.

2.4.2 Metadata improvements in local filesystems

The original UNIX file system was clean and simple in design but quite inefficient because physical head movements had not been considered during its design and the fixed size of data structures imposed limitations. Additionally, the UFS data structures were difficult to repair after an unclean shutdown or crash.

Block allocation structures

An important goal in block allocation is to find an adequate amount of adjunct disk blocks to reduce head seek times and thus avoid file fragmentation. Improved versions of UFS, such as the Fast File System (FFS), introduced two bitmap lists for free and allocated blocks [MJLF84]. To find adjunct blocks, the file system had to locate adjunct groups of bits in the free block bitmap.

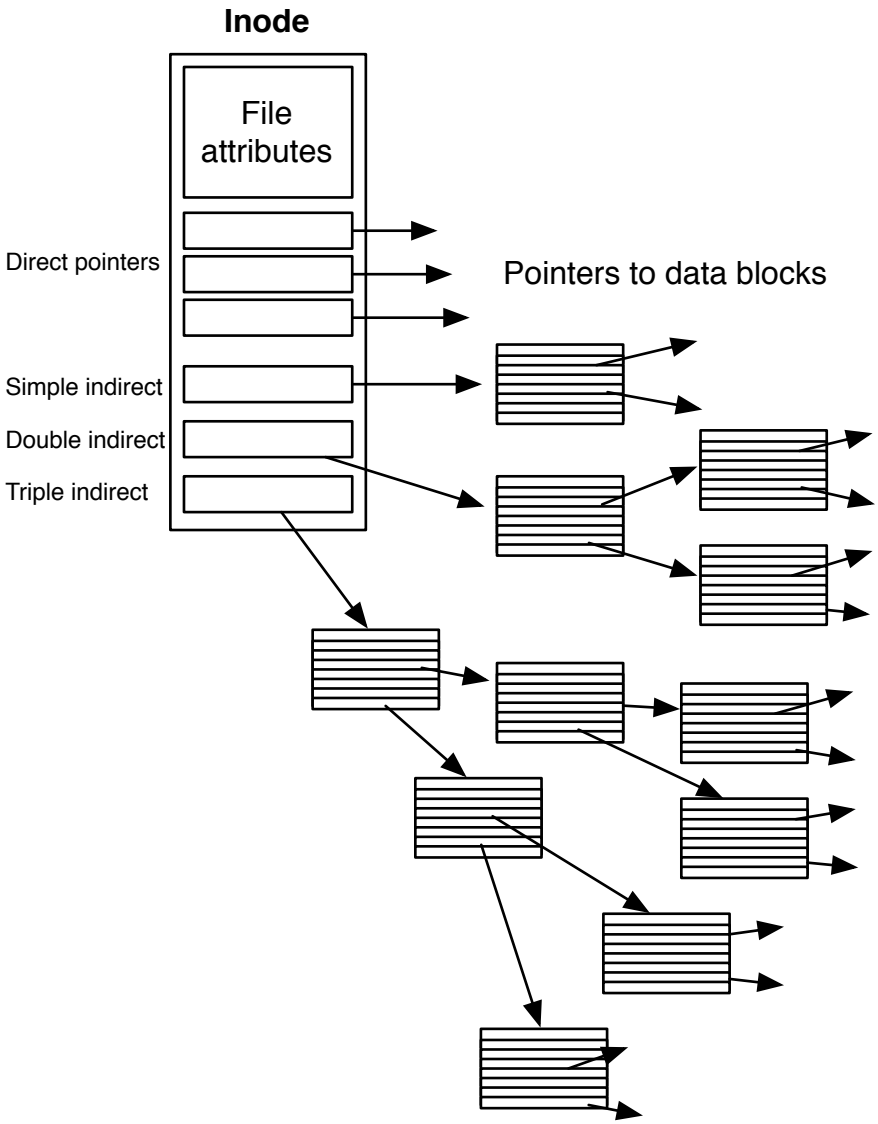

Figure 2.3: UFS inode structure

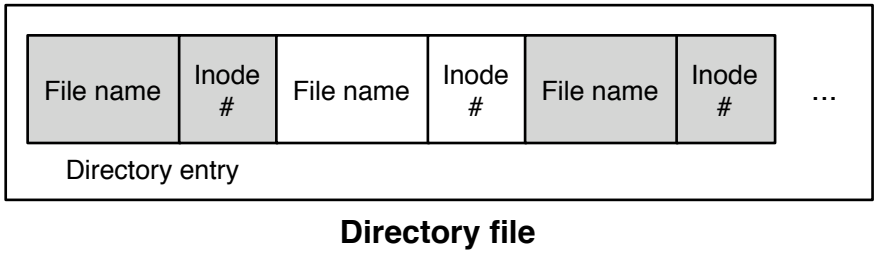

Figure 2.4: UFS directories are simple files.

When using bitmap lists, the allocation of large amounts of disk space required linear time. Modern file systems avoid managing space in small quantities. Using *extents*, which are large, contiguous runs of blocks identified by their length and the starting block, the number of allocation units can be significantly reduced [SDH+96]. Every file then consists of one or more extents and all extents in the file system can be managed using tree structures. The opposite approach is to allow flexible file system block sizes (e.g., ZFS [Sun05]) so that the size of disk blocks can be adjusted depending upon the data.

Delayed allocation

When writing to a file, UFS had to allocate disk blocks every time a write request was processed. Because of its many parallel operations, this process increased file fragmentation. Delaying allocation until it becomes necessary to flush the buffer cache has two advantages. First, adjunct blocks or larger extents can be allocated because delaying allocations provides an opportunity to merge multiple writes. Second, many temporary files can be deleted before they are flushed to disk. Delayed allocation has been implemented in XFS and WAFL, among other systems.

Data structure scaling

Maintaining a single set of internal metadata structures (e.g., block allocation maps) for a file system may result in contention when it is exposed to high parallelism. File systems such as XFS subdivide the block device into multiple separate allocation domains (called *allocation groups* in XFS) where allocation and deallocation can be performed independently and in parallel [SDH+96]. This can also aid in reducing head movements during allocation.

Flexible inode counts

The number of inodes in UFS, which are created while formatting the filesystem, is fixed. Existing inodes use capacity that could be used for files; therefore, no new files can be created if all existing inodes are used. In modern file systems, the number of existing inodes can be increased (e.g., in WAFL) or inodes are created on demand (XFS or ReiserFS).

Directory search

The traditional linear-list implementaton of directories leads to undue O(n) lookup operation complexity. A simple improvement is to store a hash of the name to confine find operations to a smaller data set (e.g., in WAFL [DMJB98]). Other techniques include using tree hashes (e.g., Ext3 [CTP+05]) in addition to the list representation or a complete implementation of directories using B-trees (XFS [SDH+96]).

Data and metadata placement

Early file system researchers noticed that placing directory and file information closely together can improve performance by avoiding the penalty of unnecessary disk seeks [GK97, DM02]. When Ganger et al. merged the inode structure directly into directories, they observed an up to 300% performance improvement in real-life applications. Many filesystems consider this fact and allocate space accordingly (e.g., XFS tries to place files in the same allocation group as the directory while FreeBSD FFS places directories next to their parents [SGI07b, DM02]).

2.5 Basic architectures of distributed file systems

The following section presents the basic building blocks commonly used in distributed file systems (see Table 2.5) to aid understanding of the implications of different solutions on performance and reliability.

2.5.1 Client-fileserver paradigm

The foundation of early distributed systems was a client-server paradigm according to which a server offers access to its local file systems by means of remote procedure calls (RPCs) that resemble the local file system API. The details of the internal metadata management and the persistent storage layer itself are hidden from the client. The RPC layer is responsible for accessing and modifying both data and metadata. This model is also described as *Network Attached Storage (NAS)*.

Widely used file system protocols such as NFS and CIFS use this model [CPS95, SNI02]. All modern operating systems have integrated clients for NFS and/or CIFS, and therefore usually hight reliability and stability on the client side. Because neither protocol requires that clients depend upon each other, both can be easily used in heterogenous, distributed environments.

Namespace aggregation

A single server places capacity and performance bounds upon the entire distributed file system. As a natural extension, the mountpoint concept known from UNIX led to the development of namespace aggregation. While it is simple to mount multiple file servers on a single client into a single namespace, maintaining a common view for multiple clients manually is cumbersome. File systems with a unified namespace provide a single, common and consistent namespace to all clients subdivided into volumes that behave like directories from a client perspective. Different volumes can reside on different servers.

Table 2.5: Overview of basic distributed file system paradigms

	Client-server file system	**SAN file system**	**Parallel file system**
Communication	Data network	Dedicated Storage Area Network (SAN, eg. FibreChannel) or block storage protocol (iSCSI) and data network	Data network
Client software layer	Standard protocol (e.g., NFS, CIFS) or specific file system client (e.g., AFS)	Specific file system client	Specific file system client
Required software/protocol version consistence	Low	High	Medium
Data storage	Local file system on server	Shared block storage device	Local file system on object storage server
Data operations	Delegated to server	Processed by client	Delegated to object storage server(s)
Metadata storage	Local file system on server	Shared block storage device	Local file system or database on metadata server
Metadata operations	Delegated to server	Delegated to metadata server *or* processed directly by client after acquiring locks	Delegated to metadata server
Basic failure mode (client)	No impact on server or other clients	Fencing, replay of transaction logs	No impact on servers or other clients
Basic failure mode (server)	File system inaccessible (part of namespace when namespace-aggregated)	Metadata server (if applicable): file system inaccessible	Object Storage Server: data partly inaccessible, metadata server: file system inaccessible
Example systems	NFS, AFS	CXFS, GPFS	Lustre, PVFS

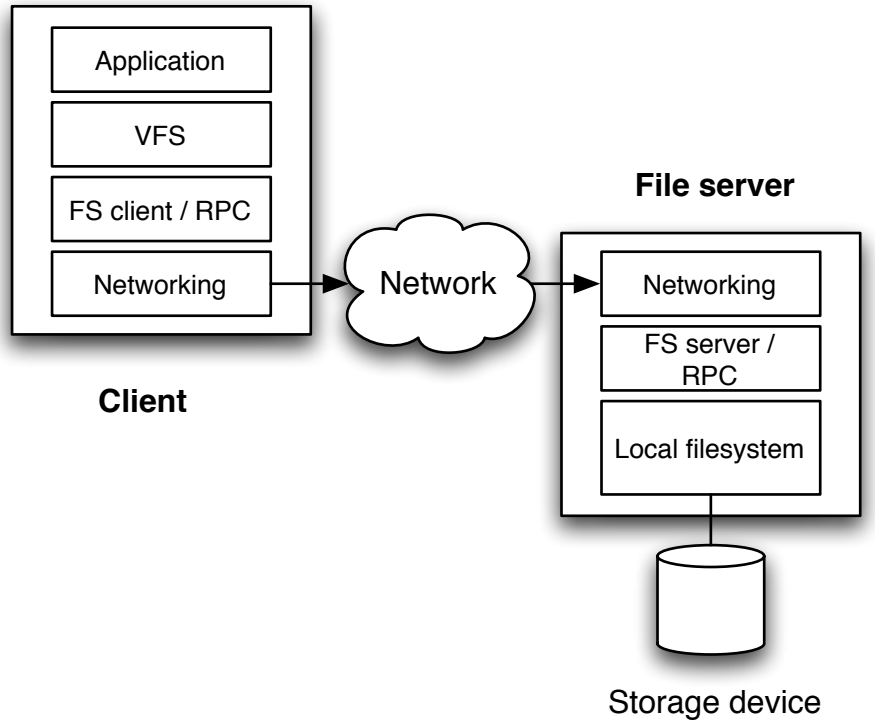

Figure 2.5: Basic client-server distributed file system

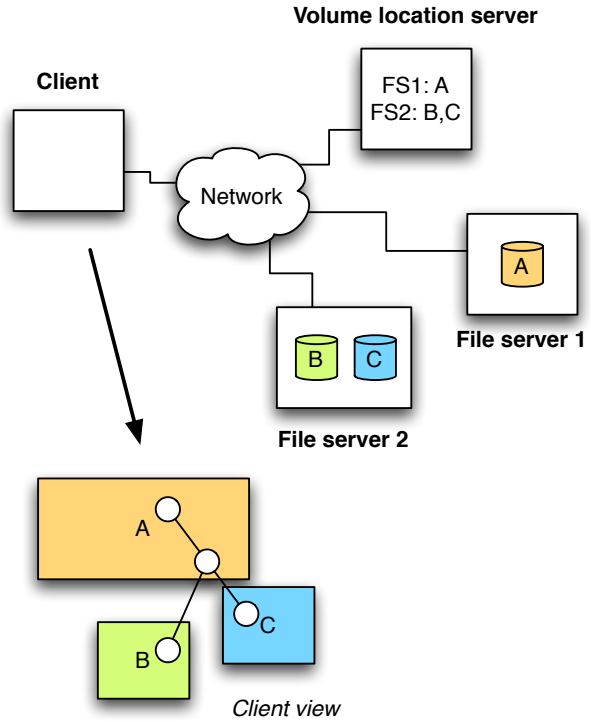

Figure 2.6: External namespace aggregation in a distributed file system

Information about the current[‖] location of a volume is stored in a common repository called the *volume location database (VLDB)*.[**] If the client contacts the volume location database and builds the namespace, then the namespace aggregation is *external* (see Fig. 2.6). This model is used in AFS, DCE/DFS and Microsoft DFS/CIFS [Kaz88, Cam98, Mic05]. External aggregation allows a client to contact multiple servers that may not be able to communicate or trust each other which is particularly important in WAN environments. However, the additional software that manages the lookup and connection process must be installed on every client.

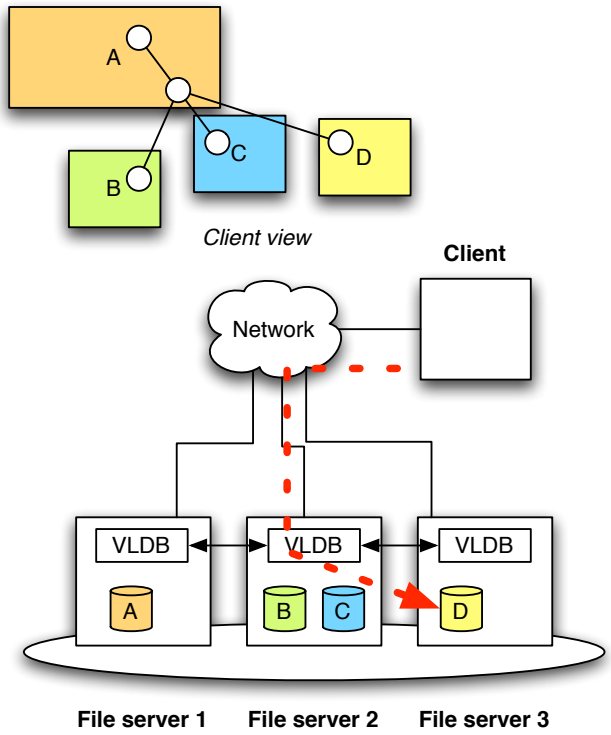

Figure 2.7: Internal namespace aggregation in a distributed file system

An alternative to external aggregation is *internal namespace aggregation* (see Fig. 2.7) wherein the VLDB is integrated into a tightly coupled cluster of file servers. Clients can use standard protocols such as NFS and CIFS and connect to *any* file server. The aggregation process is handled internally: the server receiving a client request looks up the responsible file server, forwards the request, and then delivers the response back to the client. Although forwarding possibly introduces additional overhead and may require an interconnect between all fileservers the process is completely transparent to the client and requires neither protocol changes orn non-standard software. In essence internal aggregation is a simple

[‖] Volumes can be moved transparently between servers
[**] The VLDB is often replicated to multiple volume location servers for increased reliability

upgrade path from a single file server setup. This method is used in Netapps Ontap GX [ECK+07].

2.5.2 Storage Area Network (SAN) file systems

Storage Area Networks (SANs) are networks dedicated to attaching block storage devices to a computer. By using SANs, multiple computers can share and directly access storage devices on a block level and thus, in theory, avoid the overhead required to move data using RPCs. SANs are implemented on either physical, dedicated inteconnects (e.g., FibreChannel [Cla03]), using one of many block-protocol to IP mappings (e.g., iSCSI [Huf02]) or with proprietary protocols (e.g., virtual shared disks [SH02]).

Every computer participating in a SAN file system uses a software stack that maps the abstraction of files to blocks on the storage device, similar to the functionality of local file systems. The challenge is to ensure that simultaneous access and modifications are performed in a manner that guarantees data integrity.

Central metadata management

One solution is the use of dedicated metadata servers that globally coordinate the file system. Clients must delegate all internal and external metadata operations to the metadata server and obtain locks before reading or writing data. The metadata server maintains directory information, allocates blocks and issues locks that ensure buffer cache coherence between clients. An example of a SAN file system with a centralized metadata server is CXFS [SE06].

Distributed metadata management

Other SAN file systems, such as like GPFS, distribute metadata management among clients using a token based mechanism [SH02]. Every computer participating in the file system can become a metadata manager for a file or directory by obtaining the corresponding token. After the token has been secured the computer is solely responsible for all metadata operations regarding the particular file or directory in a manner similar to the central metadata server described above.

Data throughput in SAN file systems can be very similar to the performance of the underlying storage hardware and storage network interconnect. Depending upon the type of SAN interconnect (e.g., FibreChannel), a SAN file system can be prohibitively expensive for large compute clusters. In recent designs high costs have been mitigated by technologies such as virtual disks and TCP/IP based SANs (e.g., iSCSI).

Because all file system clients have block-level access to data, they must be trusted to respect the access rules presented by the metadata server, which may lead to problems with access control. Additionally, from a practical point of view, the level of failure isolation

is lower because a single defective or malicious client can damage the entire file system. Because all computers must maintain the same software level, SAN file systems are typically only used in controlled and trusted environments.

2.5.3 Parallel file systems

Parallel file systems extend the client-server file system paradigm by distributing files to several file servers that can work in parallel on distinct parts of a single file. Clients can thus achieve a higher throughput and larger file sizes than a single server can provide. Data servers use locally attached storage and local file systems to store information. The term of an *object* is often used to describe a part of a file on a data server. Objects not only make disk space allocation on the data server opaque to clients but can also hold additional attributes. A mapping between files and objects is usually maintained by a central metadata server and replaces traditional allocation maps used in local file systems. Thus parallel file systems are also sometimes called object file systems. Examples of parallel file systems include Lustre [CFS07], Panasas [Pan07] and PVFS/PVFS2 [CIRT00, KL07].

2.5.4 Hybrid concepts

The stability and availability of the file system are major factors in the success of a production file system. When a standard protocol is used (e.g., NFS) the coupling between clients and the file system infrastructure is loose and heterogenity does not impose a problem. However, because the abilities of standard protocols are usually the lowest common denominator more advanced features, such as striping in parallel file systems, require additional client software that must be maintained. During the 1990s, several filesystems, particularly AFS and DCE/DFS required such software. While this approach enabled better integration of the distributed file system, it soon became clear that it is very diffcult to keep apace of the development of multiple operating systems, especially with a sensitive component that is tightly coupled to the kernel, such as a file system. With SAN file systems, the software on participating computers must be carefully checked for compatibility, which practically limits their use to controlled environments such as data centers.

Modern storage environments therefore often combine the concepts discussed above. For example, SAN filesystems are directly used by trusted and controlled clients and exported to other systems by means of NFS. This *re-export* model is very popular because it presents a clean, well-specified interface to the file system and enables vendors to use techniques from parallel or SAN file systems behind the scenes without the large-scale disadvantages of proprietary client software.

Another finding is that application-specific storage systems can substitute parts of a well-estabilished file system or protocol to improve particular aspects, typically data transfer performance. Examples of this process include multi-resident AFS (MR-AFS [Reu07]), which

uses AFS for metadata operations but allows a high-performance data transfer over a SAN path and dCache [Fuh07], a replication-based storage system used in high-energy physics grid applications. In dCache, a NFS-based layer is used for metadata operations but other transport protocols, such as HTTP oder gridFTP, must be used for data transfers. There are already efforts underway to standardize this kind of hybrid solution: NFS in Version 4.1 will allow multiple alternatives to data transfer via RPC such as object protocols comparable to parallel file systems or direct block device access like in SAN file systems.

The fact of existence of hybrid storage systems is important: at least for some it may be possible to separate the *access protocol* (eg. NFS) from the *storage infrastructure*. That means that scalability, performance and semantical properties of the protocol and the protocol implementation can also be considered separately.

2.6 Semantics of data and metadata access

To correctly compare different file and storage systems, one must keep in mind the behaviour guaranteed by the file systems in question, termed the *data access semantics*. Two important considerations are the manner by which concurrent and possibly conflicting modifications are handled and the manner by which data persistence can be guaranteed. Both considerations heavily influence the performance of distributed file systems.

2.6.1 Concurrent access to file data

To reduce the amount of synchronisation and locking, distributed file systems often relax consistency guarantees as compared to local file systems, which usually adhere to rules known as 'POSIX-semantics'.

POSIX semantics

POSIX specifications require that immediately after a `write` call issued by a process has returned all subsequent `read`-operations from the same or other processes pertaining to the same file region will return the data previously written [Gro04]:

> After a `write()` to a regular file has successfully returned:
>
> - Any successful `read()` from each byte position in the file that was modified by that write shall return the data specified by the `write()` for that position until such byte positions are again modified.
> - Any subsequent successful `write()` to the same byte position in the file shall overwrite that file data.

A special case are writes to files opened with the `O_APPEND`-flag. In this case each `write()` call is guaranteed to set the file pointer to the end of file before it is executed.

Efficiently implementing this behaviour in a distributed file system can be difficult because operations must be synchronised and/or (in the case of `O_APPEND`-writes) serialized. Therefore alternative, less strict semantics have been proposed, some of which require the execution of explicit steps to refresh any cached data.

Nonconflicting write semantics

The term *nonconflicting write* was introduced by Rob Ross in the specification for PVFS2 [LRT05]. While write operations to disjunct file regions behave similarly to POSIX, the results of simultaneous writes to overlapping file regions are undefined. Nonconflicting write semantics can be implemented by caching data only on servers and disabling client caching.

Close-to-open semantics

Close-to-open semantics guarantees that when one process closes an open file, another process that subsequently opens the file will be able to read the changes. A particularly important consideration here is that opening the file is a necessary precondition for this guarantee. The most important file system using close-to-open semantics is NFS [CPS95].

Open-to-close semantics

Open-to-close semantics, also known as *transaction semantics* guarantees that after a `close` or `fsync`, all other processes can read the changes. Re-opening the file is not necessary as is the case with close-to-open semantics described above. Open-to-close semantics is implemented in AFS.

Immutable semantics

Immutable (read-only) semantics completely forbids write access. While this is not useful for active file systems, most real and virtual copies of active file systems (e.g., snapshots or replicas) exhibit immutable semantics [Cam98, HLM02]. A desireable property of immutability is that it is trivial to reason about freshness of cached or replicated data, because data never changes once written.

Version semantics

File systems using version semantics can maintain several versions of a file[SFHV99, SGSG03]. Changes are applied using transaction semantics (e.g., between `open()` and `close()`). During a transaction every process works on its own version of the file. If changes are made,

new versions are created. Version semantics differs from immutable semantics in its ability to limit the number of versions by merging changes and resolving conflicts [RHR+94]. This process, unfortunately, requires knowledge about the internal structure of file data and thus cannot be implemented by a file system alone but also needs application support, and perhaps even user interaction. Still, version semantics is used in file systems for mobile applications (eg. disconnected mode in CIFS).

2.6.2 Internal metadata semantics

User-level applications do not interact with internal metadata directly. Because allocation of storage is conducted together with file data modifications, the rules for data operations apply.

2.6.3 External metadata semantics

Metadata semantics defines how the file system behaves and what it guarantees during conflicting and concurrent operations on external metadata.

Uniqueness of file names

File systems guarantee the uniqueness of file names in a directory. New directory entries emerge by opening a file with the `O_CREATE` flag, generating a link or symlink, renaming a file, or creating a new directory. In all cases, an error status code will be returned if an entry with the same name already exists in the directory. This uniqueness requirement effectively determines the performance of directory operations.

Atomic rename

POSIX states that the `rename()` function that changes the file path must be atomic. This property is often used by applications to make high-level operations atomic by creating and writing files using a temporary name or location and then moving them into the final path [Ber00]. Atomicity is only guaranteed when the source and destination are inside the same local file system: otherwise, an `EXDEV` error status code is returned because the action would involve file data movement.

In the case of distributed file systems, the behavior of `rename()` is similar to the local case: when the namespace is assembled from separately managed subvolumes it is not possible to atomically move a file between two separate sub-namespaces. For example NFS implements an `NFS3ERR_XDEV` status [CPS95]. In spite of a single mountpoint on the client, a server will respond with an error to requests that try to move a file between separate server file systems.

Race conditions on directory listings

From an application perspective, reading the contents of a directory is an iterative operation based upon calling `opendir()` and subsequent `readdir()` until all entries are fetched. The process is not atomic which means that if the content of the directory changes while an application iterates through the contents, the results will be undefined. This is true both for local and distributed file systems.

Visibility of changes

A side effect of the race condition is that the visibility of changes to metadata, such as in the creation of new files, can be delayed. Some protocols (e.g., NFS) allow time-based caching of directory entries. Because the changes might not be visible until the cache expires, the only way to check the existence of a file is to try to `open()` it; simply listing and iterating over directory contents is not sufficient.

2.6.4 Persistence semantics

Concurrent access semantics defines what state processes can see when they interact with a file system. This does not necessarily correspond to the state of the file system in stable storage. If an interruption (crash) occurs, data in volatile memory caches could be lost, and therefore applications need a way to ensure their modifications achieve persistent storage. This process can have a great influence on data and metadata performance. Depending upon the storage subsystem, persistence may mean not only writing to disk but also to non-volatile memory (NVRAM).

Normal operations

During normal operations, both data and metadata are stored in a cache in RAM and flushed to disk when it either becomes full or memory pressure is created because memory is needed for other applications. Additionally, cache modifications are written to disk on regular intervals. However, just waiting for a certain period does not guarantee a successful modification, as there is still a possibility of I/O-errors while flushing the cache. The operating system reports such errors to applications as a result to `fsync()` or `close()` calls.

File data persistence

To ensure that data is stored persistently, both implicit and explicit techniques can be applied at a single or entire file system level and, in some systems, also on all files in a particular directory.

At the file system level, a mount-time option can make all operations synchronous so that they return only after all modifications have been successfully committed to disk or an I/O-error has occurred. Alternatively, on some systems a direct-I/O mount can be used, which completely bypasses caches for file data access. Depending upon the file system, synchronous access can also be set for single files or complete directories.[††] Processes do not have to be aware of these options.

Other measures include opening a file with an `O_SYNC` flag to enable synchronous operations or issuing an `fsync()` system call on an open file to flush modifications to stable storage. These explicit methods are preferred by applications that want control over the process, such as databases implementing ACID semantics.

In both cases, internal metadata is made persistant.

Metadata persistence

Flushing metadata modifications can be a difficult process because POSIX specifications lack any standardized interface. Synchronous mounting of the entire file system or using default sync-flags on a directory will make directory modifications synchronous. Sometimes it is also possible to open a directory as a file and use `fsync()` or set a special `DSYNC` attribute that only uses synchronous operations for metadata operations. The exact procedure is important for applications that implement a transactional interface on top of a file system using files, such as email servers. To deliver an email, it is not sufficient to create and `fsync()` a file: after a crash the file will be in the file system but might be inaccessible because the referencing directory information is missing.

In distributed file systems, behavior can vary – NFS specifies synchronous behavior for all metadata operations while Lustre keeps a copy of all operations in the client cache until the server has commited everything to disk [CPS95].

2.7 Metadata consistency

Maintaining the consistency of metadata is a sine qua non for file systems, even after system crashes. As metadata is comprised of mutually interdependent data structures, an interruption of execution can lead to inconsistencies and data corruption. Thus a file system must be able to detect and correct any inconsistencies at mount time. Different procedures that can influence the performance of metadata manipulations have been developed to fulfill this task.

[††] For example using `chattr +S /directory` on the Ext3 file system.

2.7.1 Local filesystems

File system check programs

Traditional UNIX filesystems, such as FFS, introduced a consistency flag that marked the file system as consistent and was removed upon mounting [MJLF84]. In case of a crash or unclean unmount, the missing flag was detected on restart, and a check program such as fsck was started to compare all inodes, directories, and allocation maps and to detect and correct problems [Kow78]. Because this process was very time consuming and scaled with the size of the file system, several alternatives have been proposed to bypass fsck and improve startup performance after a crash. However file system checkers are still very useful to repair corruption caused by hardware problems.

Metadata logging

Metadata logging, which is analogous to the transaction techniques used in databases [Gra81], involves keeping a write-ahead change log for metadata. If instant persistence is required, the log must be commited sychronously with the metadata operations: otherwise an asynchronous logging is sufficient as long as the log is written to disk before the actual modifications. After a crash, the log can be compared to the actual state of metadata on disk to redo incomplete operations. If logging is asynchronous, some metadata operations might be lost, but the file system can still be made consistent in a time period, which is bounded by the time needed to reprocess the log. Metadata logging is a standard technique used in popular filesystems such as XFS and Ext3 [SDH+96, TT02, CTP+05].

Soft updates

Another technique used in BSD FFS called soft updates carefully orders metadata operations in a manner that guarantees the presence of no uninitialized data structures [MG99, SGM+00]. For example, an inode is always initialized before it is added to a directory. After a crash the only possible inconsistency are unreferenced data structures, which can be cleaned by a file system checker while the file system is online.

Crash counts

Patocka described an interesting method called a *crash count* whereby all internal metadata structures written to disk are marked with the pair (*crash count*, *transaction count*) [Pat06]. A table on disk stores the current transaction value for every crash count value. When the file system is mounted, this table is loaded into memory and the current crash count is incremented by one in memory. A successful write consists of writing metadata to disk, incrementing the transaction value by one, and then atomically writing the new relation between

the *crash count* and *transaction count* to disk. If the process is interrupted, after a reboot the crash count value on disk will not match the current value and the piece of metadata will be ignored by the file system.

Log-based file systems

Log-based file systems extended the idea of logging both data and metadata and by making the log a part of the file system [RO92]. An example of such a system is the WAFL file system, where all metadata is also kept in a file [HLM02]. As per definition no existing blocks are ever overwritten, changes always result in writes to empty disk locations. Modifications are stored in a new metadata file, and a root metadata file pointer is atomically changed from an old to a new pointer. Thus, per definition the on-disk state of the file system is consistent at any time. Although non-volatile RAM can be used to replay operations that were in progress after a crash, doing so is not *necessary* for consistency. A very similar technique is also used in Suns ZFS [Sun05].

2.7.2 Distributed file systems

Client-server file systems rely on the server to store data persistently. From a client perspective, an operation is stable when the server confirms it. For example the NFSv2 protocol requires every successful write operation to be persistent while NFSv3 presents a choice between this implicit commit and an asychronous mode with an explicit *commit* operation. If a server crashes before the *commit*, the client it expected to later resend the data. The local file system provides persistence on the server.

In SAN file systems, the central metadata server can maintain a metadata log because it is exclusively responsible for all operations (e.g., CXFS [SE06]) and therefore recovery is similar to that of local file systems. When metadata control is distributed, as in GPFS, every node has it own transaction log that can be replayed by other nodes in case of a failure. In SAN file systems a reliable detection of failed nodes is necessary (e.g., by timing out access tokens) to prevent an irregular access to the shared device. When a failure is detected the failed node must be hindered from modifying the file system by *fencing* it from the rest of the system; that is by physically switching it off or blocking its access to the SAN. These techniques are quite similar to those used in high-availability clustering [MB04].

Parallel file systems first use the local server file systems to recover a data or metadata server after a crash. An additional layer of logging can then provide transaction replay by using buffered data in the client's cache that has not yet been committed to the server (e.g., Lustre [CFS07]). The process is conceptually similar to a transaction replay in a local file system, with the difference being that clients must contact a server after the crash to resend their uncommited data. Other parallel file systems, such as PVFS/PVFS2, rely on

fully synchronous operations with no client-side caching; for them a simple recovery of the server is sufficient because there is no cached state on the client.

2.8 Trends in file system metadata management

Two very basic considerations are how usage patterns of filesystems influence the amount of metadata stored and whether the efficiency of external metadata processing truly matters in very large file systems.

2.8.1 Techniques for data management and their impact on metadata

Hierarchical Storage Management systems

As previously discussed the fraction of *hot data* frequently accessed in a file system is often quite small. *Hierachical storage management (HSM)* systems[‡‡] make use of this fact to allow the trading of some access latency for storage capacity by migrating file data from its original location to other storage tiers. Depending upon the system, additional tiers use cheaper disk and/or tape storage. After a successful migration, file data is deleted and a *stub* is left in its place that consists of metadata and sometimes a small part of the file. [§§] When a migrated file is accessed, the HSM system transparently loads the file back into its original location.

HSM helps reduce the amount of data on a primary file system by moving rarely used data to cheaper storage. The amount of metadata does not change: in fact, it even increases as the information needed to restage file data (i.e. its new location) must also be recorded. Some systems place this information in the stub as data or metadata whereas others use an external data store such as a database. In both cases, the fraction of metadata in an HSM file system will be higher than that in a corresponding normal file system.

Compression

Transparent compression of file data is available in several local file systems (e.g., NTFS or ZFS [Sun05]) and can be added to others using in-band file compression at the protocol level [SA05] through the use of compression appliances. Current implementations of file system level compression operate only on single files that retain all of their metadata. In contrast, archiving utilities such as TAR, JAR and ZIP merge files and their metadata into a single and possibly compressed file that also contains the directory structure. Unfortunately, even read-only access to archive files has only been implemented as a protocol handler for higher-level languages, and is not available at the file system level. Merging multiple files into one could lead to a significant reduction of metadata of seldom used and/or archived files.

[‡‡] Currently also known as *Information Lifecycle Management*-systems.
[§§] This enables easy identification of the file type without requiring restaging of the file.

Data de-duplication

An current approach used to reduce storage requirements is data de-duplication. Due to common workflow patterns, such as several users saving the same email attachment data, a file with identical data can be stored in different locations in a file system. The purpose of de-duplication is to find and merge these files by linking all occurences to a single, common data location. Identical data is recognized by first comparing the hash values of the data in question and then, if a match is found, comparing the data itself. Two basic types of de-duplication in filesystems are block-based [KPCE06] and file-based de-duplication [Mic06]. In the former, merging occurs at the file system block level so that identical blocks in different files can be identified, but only within a single file system. File-based de-duplication requires that all of the data in the files is identical.[¶¶] If data is written to a merged file, the affected data part of it is split up again. Depending on the type of data, at LRZ savings of 30% on standard home directories and 75% on VMware virtual machine disks were observed.

The impact of de-duplication on the amount of metadata is similar to that of HSM: metadata for merged files remains in place and is augmented by additional internal metadata needed to manage de-duplication. It can therefore be assumed that de-duplication also does not reduce the amount of metadata.

Snapshots

A *snapshot*, which provides a point-in-time view of previous states of file system data and metadata, is a space-efficient copy of a file system created by keeping old versions of modified blocks and all metadata. At the API level, snapshots can be accessed in exactly the same manner as the live filesystem by browsing through a file system tree. Depending upon the snapshot technology, the snapshot data does or does not belong to the original file system.

An important difference between a snapshot and a full copy of a file system is that a snapshot physically shares unchanged data blocks with the active file system. Snapshots can be created using several different techniques.

With a *copy on write*-approach (COW), the file system is not aware of the snapshot because the snapshot process takes place at the block device layer. This layer intercepts writes to a block device with an active snapshot, copies the original content of a changed block to a separate storage device and then writes the new data block. From the file system perspective there are no changes, but there is significant overhead from the read-write-write-process. If the snapshot must be deleted, the content of the separate device can be purged. To access the snapshot, the block device offers the 'snapshot' view as a virtual device that can be mounted as it has been for the original filesystem.

[¶¶]While quite similar to hardlinks, the de-duplication process is implicit and does not require application support.

The second technique can only be used with log-based file systems such as WAFL [HLM02] or ZFS. In log-based filesystems, a default behavior is not to overwrite existing data. Therefore implementation of snapshots is quite intuitive: at the timepoint of a snapshot, the root pointer to internal metadata, and thus the current block allocation data is preserved until the snapshot is deleted. Although the write process does not change in any manner, the file system must determine whether a block is still a part of a snapshot when releasing blocks. In this configuration, the existence of snapshots does not influence performance. One option to present snapshots to the API layer is as a special directory (`.snapshot` in WAFL).

The basic application of snapshots is the rapid creation of fully consistent views of file system for backup or replication purposes. An example is the hot backup of a running database that is told to write to log files during the creation of a snapshot. The database file itself, which is captured by the snapshot, is then consistent from the perspective of the database. In contrary, normal 'copy'-based backups may have problems with a race condition between ongoing changes to the file system and the backup process. Data management operations such as copying, searching and replicating data on snapshots can then rely on read-only semantics and do not have to track any changes in the file system.

2.8.2 File number and size distribution in filesystems

In their paper Agrawal et al. presented the results of a five-year study of file system metadata from several thousand local filesystems at Microsoft from 2000 to 2004 [ABDL07]. They found that mean file size had increased from 108 kB to 189 kB while the average number of files per file system grew from 30k to 90k files (see Fig. 2.8). The authors concluded that *file system designers should ensure their metadata tables scale to large file counts and anticipate that file system scans that examine data proportional to the number of files and/or directories will take progressively longer*. The environment examined in this study consisted of only software developers' workstations, and network file servers were not examined.

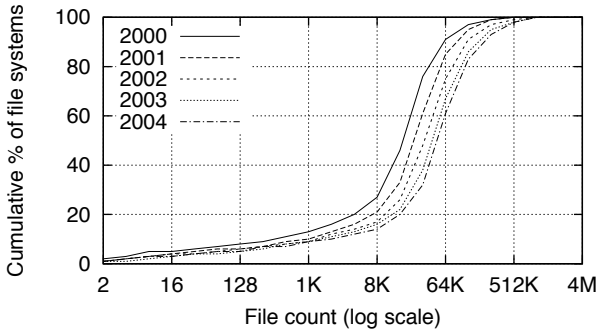

Figure 2.8: Distribution of total file count in filesystems examined by Agrawal et al., Source: [ABDL07]

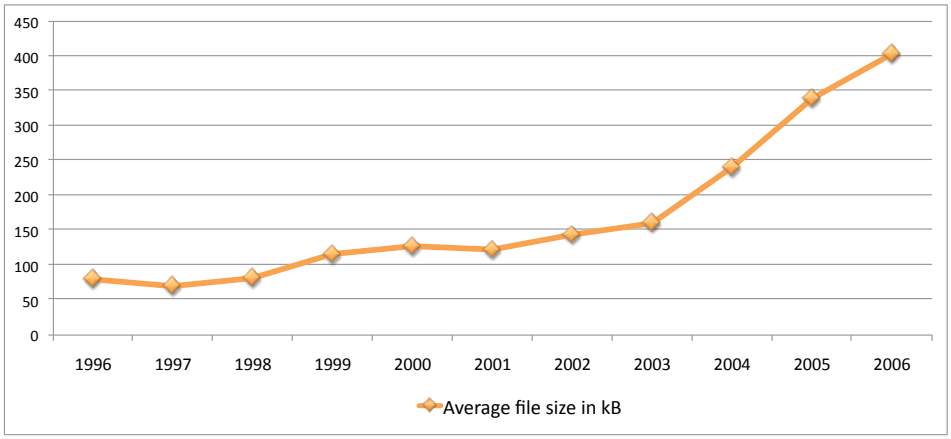

Figure 2.9: Average file size growth in the LRZ central backup system from 1996 to 2006

Publicly available statitistical data on the number and volume of files from LRZ, which has provided backup and archive services to all of Munich universities since 1999, can be used for comparison purposes [LR07b]. File systems that were backed up belong to a variety of different operating systems that are mostly file servers. Workstations are usually not backed up at LRZ.

The average file sizes in the backup data sets increased in accordance with the sizes reported in the Microsoft study until 2003. From 2004 to the present onwards an almost linear growth in average file size has been observed, which could be explained by the increasing availability of digital sensor equipment (e.g., cameras) being used for scientific purposes.

Unfortunately the exact number of computer systems that performed only backup operations is not known, but the total number of systems was 65 in 1999, in contrast to 3,912 in 2006. This corresponds to a maximum 60-fold increase in the number of systems, a 250-fold increase in number of files (from 6.9 million to 1.72 billion), and a 1250-fold increase in the amount of data (from 0.54 TB to 680 TB).

We can reason that the increasing capacity of filesystems allows users to keep older, smaller files when they migrate to larger disks and storage systems. In this manner, the total number of files in a file system continues to increase even if users regularly access only a smaller subset of files.

2.8.3 Metadata access and change notifications

Applications and data-management software access the file system in different manners: while applications operate on a smaller subset of existing files, data-management applications must often verify specific properties of all files in a file system.

Examples of data-management applications include virus scanners, file-based backups and archiving systems. An archiving system might be responsible for archiving all files

older than a month. As there is no standard file system API that allows queries for files with a specific property, such as in a database system, the entire filesystem must be searched for matching files. Other examples include incremental backup applications that identify all files that have changed since the last backup, replication systems that create a replica of modified files and search engines that index all new files.

Listing a majority of files and metadata in a file system is a slow process because placement optimizations often assume a locality of reference. Event-based mechanisms were introduced into local file systems to avoid the necessity of scanning the filesystem. Examples of these mechanisms include the *File alteration monitor FAM* for Linux that provides a notification API [SGI07a]. Applications can register and be notified when a particular directory or file changes, but it is impossible to be notified of *all* changes in the file system. The *Spotlight* search engine in the Mac OS X operating system utilizes a kernel hook that notifies an indexing module about any changes to files in *local* file systems.

A similar notification-based technique is used in NetApp file servers. On these servers operations such as creating, opening, renaming and deleting a file trigger notifications to an external file-policy server that can perform addition actions, such as searching the file or archiving it, before deciding whether to allow or to deny the operation.

2.8.4 File system vs. database storage

Many modern applications use databases rather than simple files to store their data. Transaction semantic with ACID properties offers better data consistency guarantees than do file systems. Additionally, complex queries in a relational database can be processed much faster and more effectively than in a filesystem, which resembles the old hierarchical database model with only one dimension. Thus, databases are a natural choice for applications using structured data.

On the other hand, database support for large binary objects (*BLOBs*) and the throughput when accessing large binary content never truly matched the file system layer. The reason for this is simple: Databases traditionally stored their data inside of a file system that presented a natural upper performance limit. A popular solution is to store a file path in a database and leave the file in a file system. Although Sears et al. discussed and measured some aspects of storing BLOBs in databases in comparison to file systems in [SvIG06], they focused on fragmentation issues and transfer performance.

As the amount of less structured information, mainly digital sensor data such as images, video and surveillance data rapidly increases, this situation is starting to change. Some database systems already implement their own SAN-based clustered filesystems (e.g., Oracle OCFS [Ora06]) and improved storage capabilities for binary objects (e.g., Oracle SecureFiles [Ora07b]) which include encryption, compression and de-duplication. Another interesting approach is using the NFS protocol as a standard interface for a storage system by

implementing this protocol directly in the application software instead of using the operating system (Oracle Direct NFS [Ora07a]).

Traditional relational databases manage table information using rows where all attributes of a single row are stored together. A more recent trend is the use of *vertical store* databases, where data is stored by column and not by row [SMA+07, CDG+06]. This approach is very beneficial when storing unstructured and binary data in a database because it easily maps to a backing store based upon a file system.

2.8.5 Solid-state storage devices

The design of file systems has been heavily influenced by the physical limitations of hard disks, including their head movements and platter rotation. Because todays storage systems use the same mechanical architecture as did the first hard disks developed in the 1970s, they have the same limitations.

A current disruptive trend is the introduction of solid-state disks (SSDs), which are based on silicon chips and with no moving parts, to commercially available storage systems. SSDs have been popularized through consumer electronics such as music players or digital camera equipment. In contrast those of hard disks, the access times of SSDs do not depend upon physical movements, so true random access performance is significantly better. Another advantage of SSDs is their lower power consumption and their high storage densities, which are currently increasing according to Moore's law.

NAND flash memory cells, which are commonly used, allow only a limited number of write operations, variying from 10,000 to 100,000, during their lifetime. To avoid damaging the SSD with frequent operations on the same cell, wear-leveling algorithms, which change the location of logical data blocks in a manner similar to that of a log-based file system and distribute write operations to all cells, are used. Additionally spare cells are available to replace failed ones.

SSD-based storage devices will dramatically change the design of local file systems because their improved random-access capability allows a much faster search in existing metadata structures. Traditional hard disks discouraged additional index structures (e.g., an index of all files by change time stamp) because of the additional seek operations required. Random-access penalties are reduced with SSDs, so additional metadata structures for indexing become possible.

Distributed files systems will indirectly profit from the introduction of SSD storage: while latencies from physical storage will decrease, network and processing latencies will remain unchanged. Concerning the efficiency of metadata processing in distributed file systems, this shifts the focus from implementation issues regarding the underlying local file system to the properties of the distributed file system protocol.

2.9 Summary

The purpose of a file system is to provide a common abstraction for data storage using files and directories. Filesystem data comprises the information stored in files while metadata describes the additional information needed to describe and organize the data. Operating systems provide a common set of data and metadata operations that can be used by applications and typically do not depend on the exact type of the file system used.

Local file systems store data on a block-based storage device such as a disk. Distributed file systems allow access to data on a remote system using the same file system abstraction. Basic distributed file system models include client-server, SAN, and parallel file systems, which many production filesystems combine into hybrids.

Because of the higher likelihood of failure in a distributed system, the semantics for data and metadata access and persistence can vary considerably between different (and even differently configured) file systems. This fact must be considered when comparing performance.

The role of the file system in data storage is constantly changing. Although the amount of unstructured (e.g., sensor) data is increasing, increases in disk capacities and hierarchical storage management techniques permit the indefinite retention of data. Thus, the amount of metadata has been constantly increasing while the file system APIs have remained basically unchanged for years.

With the upcoming introduction of solid-state storage devices to file server systems, local file systems will become faster and additional opportunities for acceleration will become available. At the same time, the efficiency of metadata operations in distributed file systems will depend less greatly upon the storage device and more greatly upon the semantics and protocol.

Chapter 3

Distributed metadata benchmarking

This chapter discusses previous work in the field of metadata benchmarking and the capabilities of existing benchmarks. Based upon this discussion, it identifies additional requirements that influenced the architecture of the DMetabench framework and presents analogies to application benchmarking in HPC. It concludes by presenting the design and implementation of DMetabench.

3.1 Previous and related work

The field of file system benchmarking has been thoroughly explored in previous publications, many of which described the different benchmarks proposed to obtain performance numbers for metadata operations. Benchmarks can be categorized into *micro-benchmarks*, which selectively stress particular operations or aspects of file systems (e.g., sequential throughput) and *macro-benchmarks*, which simulate application behaviour or even complete multi-application workloads.

3.1.1 Andrew Benchmark

The importance of metadata performance for scalability in distributed file systems was recognized quite early when Howard et al. used a benchmarking tool in their work about AFS [HKM+88]. They described their benchmark as a script that simulates different phases of a program compilation by creating a directory, populating it with files, and then reading and compiling these files. The load on the file system generated by a single instance of the script is called a *Load Unit*. Howard et al. then examined the performance of AFS and NFS servers based up a given number of load units. The benchmark script later named the *Andrew benchmark* is an example of an application-level benchmark.

3.1.2 Trace-based benchmarking tools

A *trace* is a recording of the sequence of requests sent and received in a distributed system. In the context of distributed file systems, traces can be used to simulate the load imposed on the file system during the original recording. Trace-based tools are typically used for testing NFS servers. Because it is mostly free of state, the NFS protocol can be easily recorded and processed.

LADDIS and SPEC SFS

LADDIS is a distributed benchmark tailored to measuring NFS server performance [WK93]. An updated version is part of the SPEC SFS 97 benchmark suite from the Standard Performance Evaluation Council (SPEC). LADDIS uses multiple processes on several nodes called *load generators* to submit NFS RPC requests to a given NFS server. As the NFS client and file system layer is bypassed, greater control can be achieved over timing and latency values. A benchmark run generates an increasing load on the NFS server until it is completely saturated. Saturation is detected when the RPC response time reaches a threshold. The load itself consists of a pre-defined mix of NFS operations. The original LADDIS paper described the load as *half file name and attribute operations (LOOKUP and GETATTR), roughly one-third I/O-operations (READ and WRITE) and the remaining one-sixth spread among six other operations*. The number of files and their size scales proportionally with 5 MB of data and 40 files for every operation per second of load applied. The final result is a table and a graph showing the RPC request latency, depending upon the number of NFS operations per second (see Fig. 3.1).

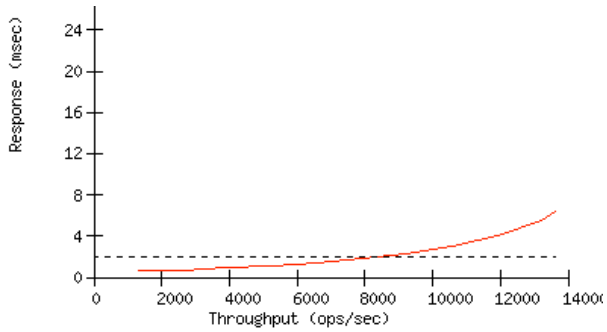

Figure 3.1: Example results of a SPEC SFS measurement of an NFS server

SPEC SFS 97 is a well-accepted parallel NFS benchmark that maintains a large share of metadata operations in the workload. In 2008 SPEC SFS was updated to SPEC SFS 2008, which modernized the workloads used (i.e. increased file sizes) and added a similar but separate benchmark for the CIFS distributed file system [SPE08]. The benchmark suite is still by design tied to the NFS/CIFS protocol and thus not useful for testing other distributed

file systems. The main focus is on the performance of the server and it does not (again, by design) take into account the behaviour of the client software. Therefore, SPEC SFS is useful and accepted among storage vendors for comparing NFS and CIFS server performance.

TBBT

In [ZCCE03] Zhu et al. discussed the use of traces to benchmark NFS servers and then proposed their own toolkit they called *Trace Based filesystem benchmarking tool (TBBT)*. The paper contains an interesting discussion about how to scale up and down a given trace to generate a higher load: a *spatial scale-up* generates the load on disjoint directories and enlarges the working set while a *temporal scale-up* speeds up the playback of a trace and maintains the amount of data. TBBT is, however, unable to test any filesystems other than NFS.

3.1.3 Lmbench-suite and Lat.fs

McVoy et. al introduced the microbenchmark suite *lmbench*, which contains different components for measuring memory bandwidth, memory latencies, system calls, signal processing times, and other system parameters [MS96]. A part of lmbench is *lat.fs*, a file system benchmark that measures the "file system latency" - the time necessary to create or delete an empty file. In his paper, McVoy compared the file system latencies between the different local file and operating systems of the 1990ies. Interestingly, the paper briefly described why the persistence semantics of the file system is an influential factor in performance.

3.1.4 Postmark

Postmark, another macro-benchmark developed by J. Katcher, an employee of Network Appliance, simulates the work of a simple mail server that receives messages, puts them into the file system, and then reads and deletes them [Kat97]. A postmark run consists of three phases. In the creation phase, a number of subdirectories and files are created. The transaction phase simulates the workload of the imaginary mailserver by creating, appending, reading, and deleting files. The third phase removes the files and subdirectories.

The benchmark itself is written in C and is strictly single-threaded. Therefore, it also mainly measures file system latency and is not able to show the maximum scalability of a file system for multi-threaded applications.*

3.1.5 FileBench

A newer benchmark, FileBench from Sun Microsystems, is a framework for the simulation of application loads in file systems [MM06]. FileBench uses a definition language for operations

*As Network Appliance sells NFS server appliances using a non-volatile memory cache that reduces latency for NFS writes, the benchmark was definitely a positive development.

and several pre-defined models that simulate the workload of common applications, such as mailservers or database servers. The chosen workload model can be simulated using several threads or processes in parallel. Unfortunately, as of 2008, it is not possible to distribute the simulation among several computers.

3.1.6 IOzone

IOzone is a data throughput benchmark that can also perform intra-node and multi-node parallel measurements [NC06]. Multi-node measurements are coordinated using an RSH/SSH based login. A recent addition is the `fileops` program which offers metadata performance measurements for all basic metadata operations in a single-process scenario but does not allow parallel measurements.

3.1.7 Fstress

Fstress is an older distributed NFS benchmark quite similar to SPEC SFS 97 but purely synthetic [And02]. It includes several example workloads (e.g., a "web server"). The output of Fstress also shows the per-operation latency, depending upon the number of operations per second.

3.1.8 Clusterpunch

Clusterpunch is a software framework that can be used to execute small, customizable benchmarks on UNIX-based compute clusters [Krz03]. Default benchmarks available in clusterpunch stress the CPU, memory and file I/O. Each benchmark run is very short (a *punch*) so that the framework can be used to determine the current load and to classify cluster nodes by their benchmark throughput. Possible applications include monitoring and improving load balancing by quickly estimating the load of nodes in a cluster.

3.1.9 Parallel I/O benchmarks

Several benchmarks have been designed for measuring the performance of parallel file I/O, including b_eff_io and the NASA NAS parallel benchmarks[RKPH01, WdW03]. These benchmarks are able to utilize a large parallel environment using the MPI programming interface. Parallel file I/O is, however, the exact opposite of metadata-intensive operations, as it results in simultaneous access by many processes to a single file.

3.1.10 Benchmarks created at the LRZ

The Leibniz Supercomputing Center (LRZ) in Munich has been researching metadata benchmarking for strictly practical purposes: the first German national supercomputer HLRB [†] – a Hitachi SR 8000 – ran 1,344 processors in 168 nodes using a single instance of Hitachi's UNIX operating system. Unfortunately, metadata operations in its parallel file system were implemented inefficiently and data structure locks were needed across all the processors. The machine was capable of less then 10 metadata operations per second, which, because clearly not sufficient for a national supercomputer, required the use of external machines for compiling software and preprocessing data. Since 2004, the LRZ has been carefully testing metadata performance on each new supercomputer procurement using its own benchmarks.

Metabench

Metabench is the result of a student's work during the research for this thesis [Str04]. It is a proof-of-concept work that enhances Postmark with a simple, MPI-parallelized runtime system. While all participating computers still work independently cumulative numbers computed using MPI barriers are given after each phase. As a functional prototype, Metabench has shown the usefulness of automated and graphical result analysis. The student paper additionally presents the results of a 50-node Linux cluster using AFS, NFS and PVFS.

Acceptance benchmarks for the HLRB2

For the procurement of the new national supercomputer HLRB II [LR07a], the LRZ used its own distributed metadata benchmark, which is quite similar to Metabench but only computes a total throughput in metadata transactions per second (tps) for a given number of processes. In the procurement itself, different performance levels were required for the home file systems (5,000 tps) and the parallel file system (800 tps). These values were obtained experimentally as a minimum acceptable performance level for interactive and batch usage.

3.2 Discussion of objectives for a new benchmark framework

Existing file system benchmarks focus on measuring quantitative aspects of file system performance, in particular the throughput of data access. The few available metadata-specific benchmarks, such as fileops, do not permit measurements in distributed environments with multiple nodes.

The idea of and motivation behind this thesis is to provide a benchmark framework for metadata operations that does not depend upon a particular file system (i.e. is *platform and file system independent*), is usable both on small systems and highly parallel supercomputers

[†] *Hochleistungsrechner in Bayern (HLRB).*

(i.e. is*scalable*) and works in a manner which makes it useful for comparing different implementation concepts, finding bottlenecks, and tuning particular file system environments.

The following section will discuss these aims in detail and identify the necessary requirements.

3.2.1 Portability and file system independence

As described in Chapter 2, although there are several standardized distributed filesystems, such as CIFS or NFS, many modern developments with advanced concepts, such as Lustre, are still available only for selected operating systems.

Protocol-specific benchmarks, such as the SPEC SFS benchmark, allow a quantitatively precise comparison of different file system servers for a given workload. Unfortunately, specialization on a single protocol (NFS in the case of the SPEC SFS 97 benchmark) is, at the same time, a limitation. It allows identification of a high performing server but not comparison to other file systems that might exhibit other and possibly more suitable behaviour and semantics. Thus, if interested in analyzing filesystems at not only the implementation but also the conceptional level, a more generic approach that utilizes the standard file system API is preferable.

3.2.2 Benchmarking distributed systems

The scalability of distributed filesystems has dramatically changed over the last two decades: while the 1988 Andrew benchmark paper [HKM+88] cited a *goal of supporting 50 clients per server*, current distributed file systems can easily support thousands of clients per server. For example, in 2008 a single standard production file server at the LRZ simultaneously supported 450 workstation clients using CIFS and 100 servers using NFS on a dataset of 16 TB.

Changes in distributed systems architectures

In addition to larger distributed environments in terms of node counts, the architecture of a single node is also changing. Current problems in increasing the CPU clock rates have dictated a shift to multi-core CPUs and larger SMP nodes. It is now common to find 16, 32, or more CPU cores inside a single computer node, and the current state of the art is 1,024 cores in single operating system instance (e.g., on the SGI Altix architecture [Sil06]).

Load placement awareness

How have these changes impacted benchmarking file systems? A modern benchmark must be able to run and stress the file system in accordance with one of two types of parallelism: parallelism *inside a single computer node* and parallelism *among multiple nodes*.

Intra-node parallelism describes standard operating system techniques, such as processes or threads, which enable a parallel execution of operations, in this case on the file system. As file system access inherently includes waiting for the execution of I/O operations, multiple processes can increase improvements even for single CPU systems. Intra-node parallelism can be used to test the interaction inside a single operating system instance for both local and disributed file sytems.

Inter-node parallelism describes simultaneous access to a file system by multiple computer nodes with independent operating system instances. Inter-node paralelity requires a distributed file system. The crucial difference between inter-node parallelism and intra-node parallelism is that their synchronization and cache coherence techniques differ significantly. The main reasons for this difference include a possible heterogenity of the nodes (e.g., different operating systems), lower synchronization efficiency due to communication latencies and the presence of failures. As has been previously discussed, file system semantics can also differ for simultaneous operations inside a single node and operations on multiple nodes.

It can be concluded that a file system benchmark needs to be aware of these models and be able to selectively stress both kinds of parallel operations.

Data placement awareness

parallelism is not an exclusive feature of the application side. Modern storage systems distribute data and processing to improve capacity, reliability, and performance. Correctly utilizing these distribution properties is a key to high performance and scalability.

As discussed in Chapter 2, depending upon the file system architecture, data and metadata can be stored together (e.g., in a simple client-server file system) or separately (e.g., in a parallel file system). An important technique for distributing file system metadata is the aggregation of multiple namespaces. A benchmark program should be able to differentiate namespaces and selectively operate on any given subset. In combination with load placement, this technique can be useful in revealing the interaction between client access parallelism and processing parallelism on the file system side from a black box perspective.

3.2.3 Metadata operations and scalability

From an user API perspective, the duration of a single metadata operation, such as the creation of a file, can be very brief when compared to that of the entire I/O process as specified by metadata semantics. Chapter 2 discussed the importance of semantics in chapter 2 and how file systems and operating systems can differ significantly with regard to guarantees. For example the first metadata operation could fit into a cache or be carried on top of an RPC call, but the second one might have to wait until the first has finished before it can begin. In such a case measurement results for the operation will differ greatly, depending upon the

current state of the entire system. Additional factors such as time-based events (e.g., flushing writeback caches at regular intervals) further complicates the matter because operating systems do not provide information about internal tasks to userspace programs.

A common solution used in micro-level benchmarking is to execute an operation n times to prolongate the measurement. Is this solution applicable to metadata operations? Performing long series of identical operations, for example reading the attributes of all files in a directory, are not atypical for metadata-intensive workloads. In particular, automatic data management applications, such as those discussed in Chapter 2, are characterized by highly repetitive processes. This behavior differs significantly from other fields in benchmarking, such as CPU performance benchmarks, where hundreds of identical micro-operations (e.g., performing *add* or *multiply*) are rarely encountered stand-alone.

Choosing the problem size

To measure the performance of a particular metadata operation (e.g., creating a file), the correct *problem size*, defined as the number of operations performed during a measurement, must be selected. The problem size should be large enough to guarantee a runtime during which a stable level of performance is reached and start-up effects are bypassed. On the other hand, if the problem size becomes too large, side effects (e.g., decreased search performance in large directories) might influence the measurement, leading to the total benchmark run time to become too long.

Parallelism, scaling and runtime

In a distributed benchmark, the relationships between the problem size and the multiple processes performing operations must be considered. Application benchmarking in HPC environments uses two categories of scaling. If the problem size increases linearly with the number of processes, *weak* or *isogranular scaling* is tested and if a fixed problem size is processed by an increasing number of processes, *strong scaling* is tested (see Table 3.1).

The reason for differenciating between both types is that side effects, such as caches, can have a large influence on performance. A smaller problem size might fit into the CPU cache, while a bigger one needs memory access and an even bigger one NUMA or MPI communication across multiple CPUs or nodes. In capability-type computing, strong scaling determines how rapidly a particular problem can be solved, while weak scaling identifies how many instances of a given problem can be solved in parallel using a given system. Can these concepts be applied to metadata and how does multi-process scaling interact with the problem size?

With reference to the file creation example in Table 3.1, isogranular scaling means the operation of creating a file is repeated 6,000 times in every process. If the file system under

Table 3.1: Examples of weak/isogranular and strong scaling with an initial problem size of n = 6000

Number of processes	Isogranular scaling		Strong scaling	
	Total	Per-process	Total	Per process
1	6000	6000	6000	6000
2	12000	6000	6000	3000
3	18000	6000	6000	2000
4	24000	6000	6000	1500
5	30000	6000	6000	1200
10	60000	6000	6000	600
100	600000	6000	6000	60
1000	6000000	6000	6000	6

observation does not scale perfectly – and most do not – the wallclock runtime of the measurement will differ very significantly between the lower and upper number of processes. With many processes, the total number of objects created might lead to unacceptable and unpredictable runtimes that slow down the measurement process without delivering additional, relevant data. To compensate for this problem, the initial problem size could be reduced, but then the runtime for a low number of processes might become very brief.

Strong scaling reduces the per-process problem size with an increasing number of processes. With stron scaling, the runtime for a large number of processes might become very brief and the issues are exactly the inverse of those that arise with weak scaling.

In a file system context, strong scaling is useful in determining the degree of parallelism that is best for a given problem size. Weak scaling can then describe how the entire file system will perform under an incrementally increasing load.

Ideally a benchmark tool should be able to provide information about both types of scaling, as does the interval-sampling technique described later.

3.2.4 Extendability

When the performance of HPC applications is estimated with benchmarks, *application kernels*, small pieces of code that implement the core functionality of an application, are often used. Metadata workloads can also be modeled with multiple operations, such as `open()` and `close()` or `readdir()` and multiple `stat()` calls, that resemble "metadata kernels".

There are two basic processes that can specify the operation of a benchmark program. The first one parametrizes built-in kernels using the options given at runtime. An example of this type of benchmark is IOzone, which has several built-in operations such as sequential writes, sequential reads and random writes and more than 20 different parameters that specify details for every operation. An advantage of using a built-in list of operations and

parameters is that the results can be compared easily. On the other hand, it is more difficult to add new operations because, for example, the change of units between a random I/O benchmark (ops/s) and a sequential benchmark (MB/s) requires custom measurement infrastructure or every operation.

An alternative process creates custom definitions of an operation using user-supplied program code loaded and executed in a benchmark framework. This framework provides a common runtime environment and measurement infrastructure while small benchmark plugins implement the desired operations. An *operation per second* is used as the common measurement unit and what an 'operation' exactly does is defined by the particular benchmark plugin: it could create a single file or perform a single `stat()` call.

3.2.5 Interpreting measurements

By using a plugin infrastructure, the benchmarking process is largely abstracted from the file system operation itself. As performance is usually given as work completed per time unit, one can run a given number of operations, measure the time needed, and then claim that, on average, one measured 3,000 operations per second of operation A or 2,000 operations per second of operation B. But is this enough data to draw conclusions about the performance of the file system? For complex, distributed measurements with multiple dimensions of parameters it is not enough, because it leads to the loss of much useful information.

Result compression

For **application-level benchmarks**, an effective approach to describing the performance of a file system for a particular workload is to summarize the results, sometimes even into a single number. Postmark, for example, runs a suite of different metadata-intensive operations and returns a number indicating the *transactions per second* [Kat97]. It is simple to compare different filesystems when considering the same workload, but because the relationship between this single result and the operations performed is opaque[‡] this method is less suitable as a tool to systematically identify issues and improve a file system. Too much information is averaged and/or lost. In contrast **microbenchmarks**, such as fileops, break down measurements into detailed performance numbers directly connected to specific operations, but again, information about time-related effects and their influence on performance is lost.

Neglect of the time parameter

One parameter that has been neglected in metadata benchmarking is time: more precisely the runtime of the benchmark. Why is the runtime important for a file system benchmark?

[‡]The exact combination of operations in Postmark is random.

As previously discussed, a benchmark for distributed file systems must utilize multiple processes working in parallel, and different processes perform their tasks independently. In a sequential benchmark, there is one process and one point in time when it completes. With multiple processes that may be running on different nodes, it is quite common that different processes do not work at the same speed. This may be due to systematic reasons (e.g., unfair processing of requests by the file system), differences in hardware or software configuration, or mistakes in the benchmark setup (e.g., additional processes running on the nodes and using up CPU time). Small deviations in performance can lead to results difficult to explain if only summary numbers are available. These deviations are quite similar to the ”noise” effects observed in large parallel computations in which a single lagging process delays the entire calculation [TEFK95].

Unfortunately, the common techniques used to obtain summary result calculations, including those described below, lead to the complete loss of information about many events.

Global throughput approach

The simplest way to calculate a ”result number” is to divide the total number of operations by the wallclock time required to complete it. While this is a perfectly valid approach for a single-threaded, single-process benchmark such as Postmark, it is problematic in a parallel environment. Consider example (b) in Figure 3.2, where the three processes P1, P2 and P3 each execute 1,500 operations, and process P3, which finishes after 15s, is slower than P1 and P2. The average number of operations per second would be (3*1,500)/15s = 300 op/s. Unfortunately this single number does not give any indication of a problem and is indistinguishable from example (a), where all processes work at the same speed.

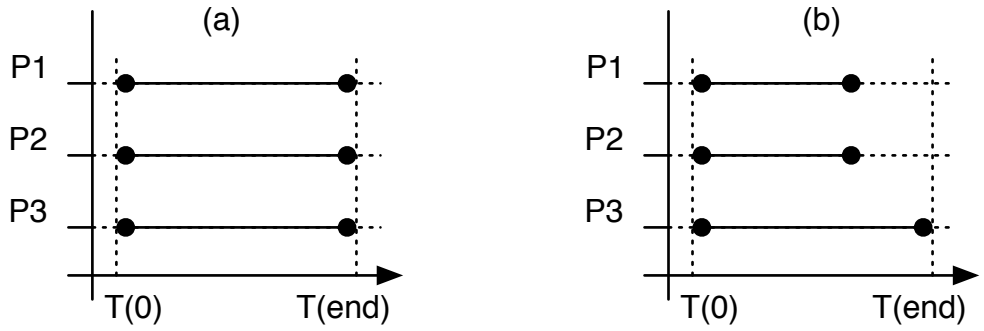

Figure 3.2: Average throughput measurement

The ”stonewalling” approach

”Stonewalling” is a term introduced by the IOzone I/O-benchmark [NC06]:

> Stonewalling is a technique used internally to IOzone. It is used during the throughput tests. The code starts all threads or processes and then stops them on a barrier. Once they are all ready to start then they are all released at the same time. The moment that any of the threads or processes finish their work then the entire test is terminated and throughput is calculated on the total I/O that was completed up to this point. This ensures that the entire measurement was taken while all of the processes or threads were running in parallel.

Stonewalling is very useful if the objective is to estimate the total performance of the file system without concern for unfairness among particular processes. IOzone also displays the minimum and maximum amount of data that was processed, which is the equivalent of the number of operations completed for metadata and, optionally, the time needed by every process; however other information about differences in speed is lost.

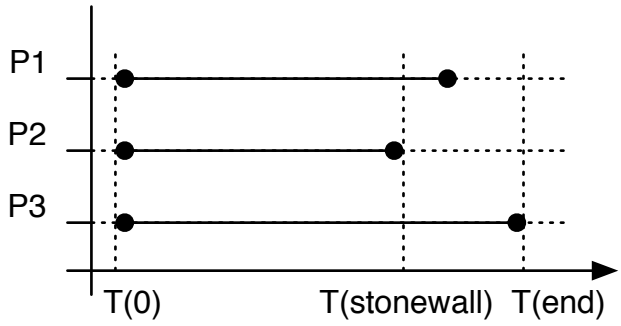

Figure 3.3: The stonewalling approach

Proposal for a time-interval logging approach

A time-interval logging approach is proposed to retain information about the behavior of every process. In this approach, every single process records the number of operations already completed within fixed time intervals until the entire task is completed (see Fig. 3.4).

This method preserves both the performance of every single process as well as the function of time vs. the number of operations completed. Based on this data the two averages described above, as well as total performance numbers for all or a subset of processes can be computed. In this manner a single benchmark run can provide much more information than the compressed and averaged results of existing benchmarks.

The example in Fig. 3.4 shows three processes that each performs 30 operations. A wallclock-time average of 18 operations per time unit is the result of 90 operations performed within five time units (90/5=18). The stonewalling approach would yield a result of 23.3 operations per time unit (70 ops/3 time units=23.3). The axis labeled ”Total” in the

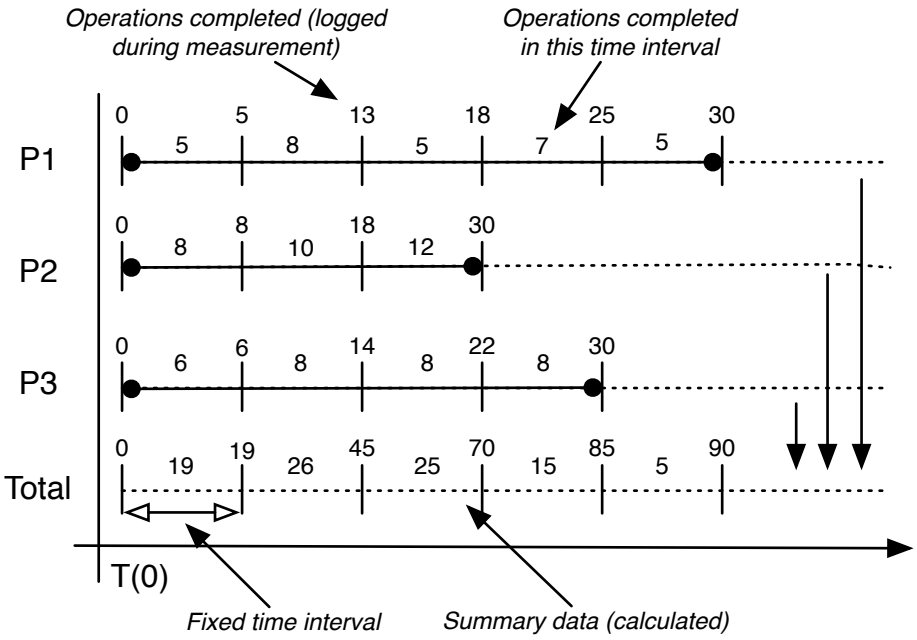

Figure 3.4: Time-interval logging *(Time interval enlarged for illustrative purposes)*

figure indicates exactly how performance changed between each time interval. For illustrative purposes, few intervals are used in the figure. In reality, the time interval would be kept much smaller than the total runtime, but still long enough so that a larger number of operations are completed within each time segment.

Obtaining the progress data requires running additional execution threads parallel to the benchmark itself. An alternative approach tested in [Str04] that is simpler to implement is to record the time elapsed when a process has completed a fixed quantum of operations (e.g., every 100 operations). However, in the context of file systems, a fixed time interval provides a significant advantage: Side effects that emerge based upon the time frame can be observed in detail, regardless of operation performance. It is also easier to compare different measurements if they share a common time grid.

The method presented above makes use of time intervals that all start at the same time and requires a suitable synchronization mechanism (e.g., an MPI barrier). Alternately, every process could keep track of time using a global time reference, for example an NTP-synchronized clock.

Time-based logging and scaling

When fine-granular data on the number of operations completed per time unit is available, strong scaling can be simulated. If the performance for a fixed number of operations N is needed, the first time interval within which the number of operations completed is larger than N can be identified and the average performance then computed from T(0) to that point

in time. In a similar way, the 'stonewall' average can be calculated by identifying the time interval within which the first process has completed its operations.

3.2.6 Environment profiling and result reproduction

A very important consideration in scientific research is the ability to reproduce experimental results; in this case, the results for benchmark runs. However, exact reproduction of results is often difficult or even impossible in large distributed systems for a variety of reasons.

The difficulty of reproducing results

The difficulty of reproducing results can be illustrated by considering the *testbed configuration* of large computer clusters, which is often globally unique. Technological advances may make it impossible to purchase a sufficient quantity of the same hardware as that used in the original experiment, even if only weeks have elapsed since the experiment was conducted. Therefore, rebuilding an identical cluster setup can be prohibitively expensive in terms of time and money.

Ongoing *changes to system software* in production systems, which are necessary for system security and stability, must also be considered. Software updates in large systems, which occur every few weeks and can be very difficult to reverse, can influence measurement results in unpredictable ways.

Unintended configuration differences are possible even with access to central configuration management tools, such as cfengine [Fri02], because the default parameters of the operating system or the file system must first be considered relevant before they are managed centrally. Thus unexpected differences can falsify assumptions about the runtime environment of a benchmark.

Finally, while *good scientific practice* requires the use of precise protocols for obtaining every measurement, making manual recordings in distributed environments is so labor intensive that doing so is often avoided.

Reproduction by retrospective analysis

To avoid the fundamental problems discussed above, a different approach is proposed: If it is not always possible to reproduce a benchmark run, it should at least be possible to trace the exact system configuration during the original benchmark run. This means that it is mandatory to comprehensively record the static and dynamic system state of the runtime environment.

Static properties include hardware type, software versions and configuration data while dynamic properties describe running processes, and CPU or file system utilization before a benchmark run. Using both kinds of data and the runtime performance protocols, it is

possible to analyze reasons for variations in performance – even after the benchmark run. This method has similarities to regression testing in software development.

3.3 DMetabench: A metadata benchmark framework

DMetabench is a new metadata benchmark framework which implements many of the requirements above and provides a lightweight platform for experimental benchmarking of metadata in distributed file systems.

3.3.1 Overview

The main objective of DMetabench is to execute metadata-related operations using varying numbers of nodes and parallel processes. An *operation* is a short sequence of metadata calls that simulates a real-world operation (e.g., creating a file using `open()` and `close()`).

DMetabench first executes a configurable number of operations using one or multiple processes that run in parallel and then provides detailed information about global and per-process performance. Depending upon the number of available nodes, different combinations of the number of nodes and processes per node are tested automatically.

Benchmark results from DMetabench can be automatically processed and compared using built-in charts. Additionally information about the runtime environment is recorded for later analysis.

3.3.2 Architecture

To provide execution parallelism for metadata operations DMetabench uses MPI [For03] and implements a master-worker architecture with multiple processes (see Fig. 3.5). The benchmark itself is written in Python (see section 3.4.1).

Master process

A master process coordinates the benchmark run, analyzes the runtime environment, distributes subtasks to worker processes and assumes responsibility for writing out result sets. It does not take part in the benchmark process itself.

Worker processes

Worker processes receive commands from the master process and react to them by starting a corresponding operation, such as an analysis of the runtime environment, an idle loop or a benchmark task. In every valid DMetabench configuration there is at least one master and one worker process.

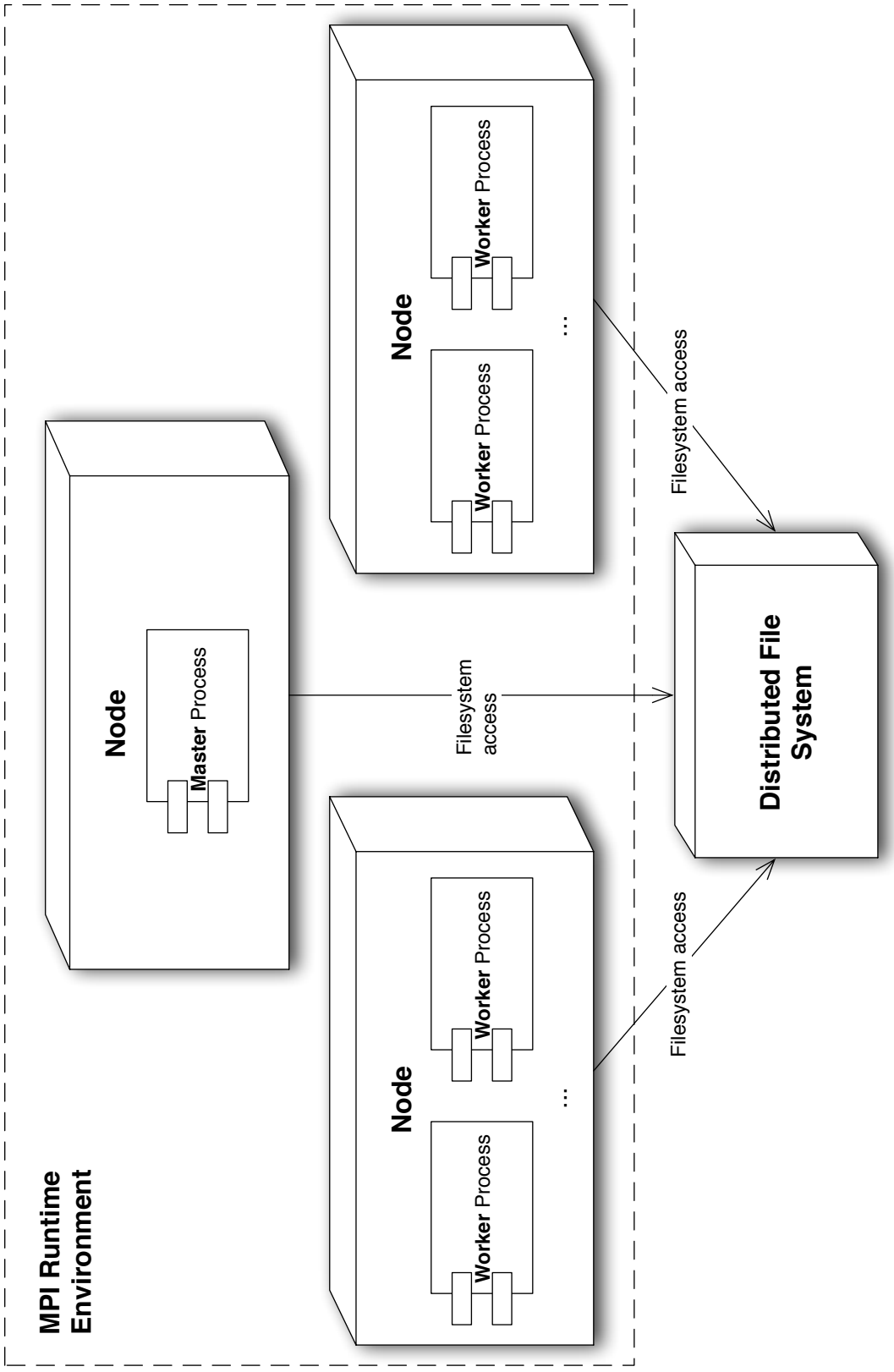

Figure 3.5: Sample DMetabench deployment diagram

3.3.3 Benchmarking workflow with DMetabench

DMetabench was created to facilitate rapid benchmarking. The following section describes how a typical benchmarking session with DMetabench appears.

Choice of operations and parameters

Before performing the benchmark, the operator must choose operations and parameters. DMetabench comes with several pre-defined operation types (e.g., *MakeFiles* to create files), but custom operations can also be defined. Multiple operations can be tested in a single run of DMetabench.

Important parameters include the number of operations to perform (*problem size*) and a target file system path (*working directory*) where all operations will be performed. Parameters are discussed in more detail in section 3.3.5.

Startup and placement discovery

Because DMetabench is an MPI program, it must be started inside an MPI environment (e.g., using an `mpirun` command). The MPI environment provided to DMetabench imposes natural limits on the number of nodes and processes. For example, assume that the MPI runtime environment uses the three nodes *A, B* and *C* and starts three processes on each node numbered globally from *0-8*.

After startup, the DMetabench master, which always has an MPI process rank/ID of 0, will begin an environment discovery process and instruct the workers (all other processes) to report the node on which they are running. Using this information, a mapping table between available nodes and process IDs is built, which enables the master to identify limitations and possible combinations of processes and nodes.

In a sample configuration, a mapping table could appear like that in Table 3.2. Regarding the nine processes, process 0 is (as always) the master, workers 1 and 2 are placed on node A, workers 3,4 and 5 on node B and workers 6,7 and 8 on node C. It must be kept in mind that DMetabench *cannot* influence the placement because it is unmutable and predetermined by a given MPI environment, but it can intelligently choose placements for benchmark tasks from the status quo. The MPI environment can of course be influenced using the `mpirun` command and the operator can choose a suitable combination of nodes and MPI slots per node.

From the placement discovery table, DMetabench infers that it is possible to run benchmarks with one or two processes per node on one, two or three nodes, and three processes per node on one or two nodes. This results in an execution plan for the benchmark, such as that in Table 3.3, and allows the benchmarking of multiple possible combinations in a single benchmark run.

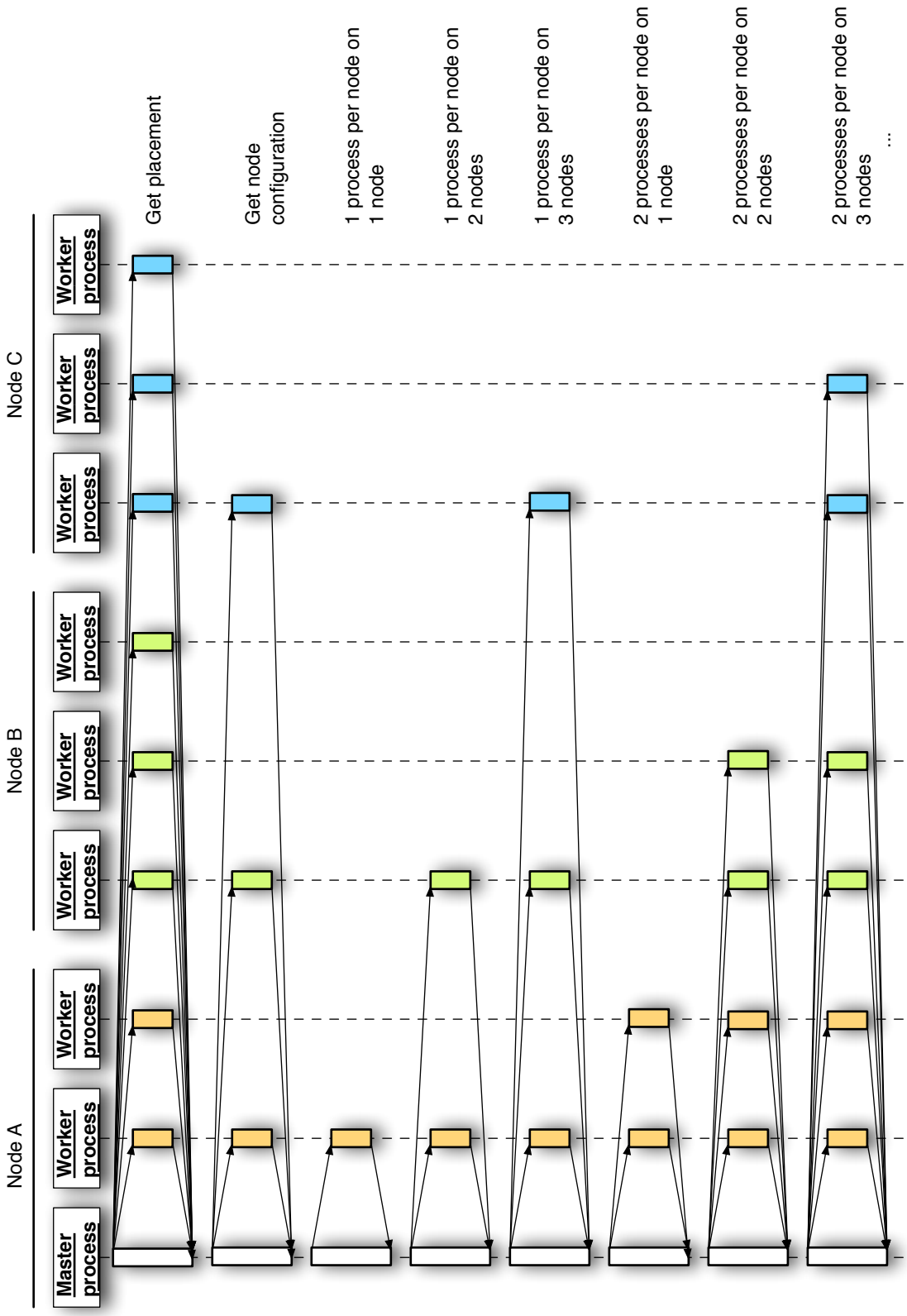

Figure 3.6: DMetabench workflow sequence diagram for the sample configuration in Table 3.3

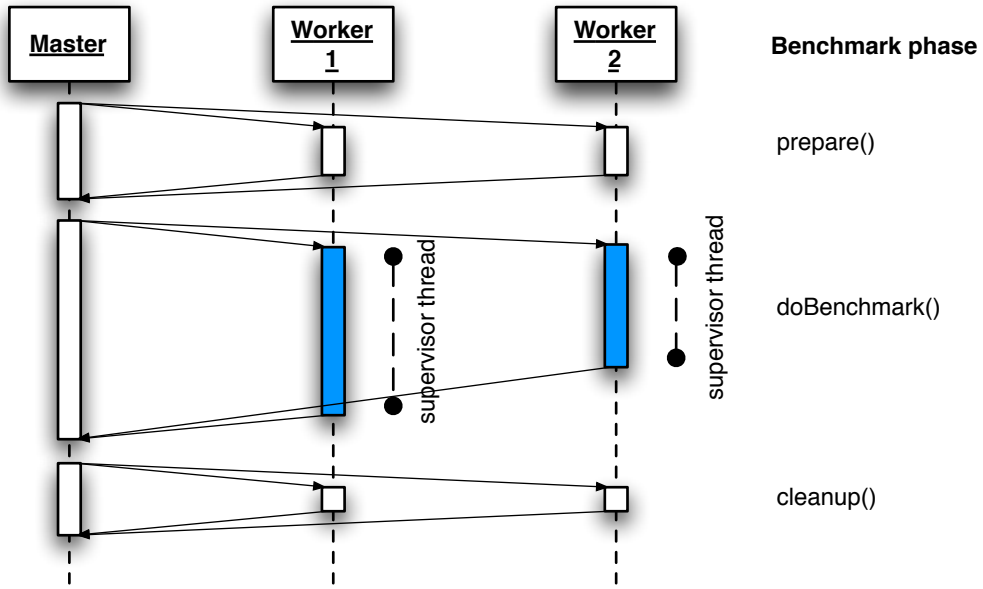

Figure 3.7: Detailed sequence of the three benchmark phases. Operations are performed by worker processes in the section marked in blue. A supervisor thread runs in parallel and logs the progress at regular intervals.

Table 3.2: Sample results of placement discovery

Node	IDs of available worker processes
A	1, 2
B	3, 4, 5
C	6, 7, 8

Table 3.3: Example benchmark execution plan derived from Table 3.2

Number of processes per node	Number of nodes	Total number of processes	Process placement (worker IDs)
1	1	1	1
	2	2	1,3
	3	3	1,3,6
2	1	2	1,2
	2	4	1,2,3,4
	3	6	1,2,3,4,6,7
3	1	3	3,4,5
	2	6	3,4,5,6,7,8

System environment profiling

To preserve information about the system environment, DMetabench instructs one worker on each node to collect hardware, software and configuration information. Additionally, it runs a system statistics process (`vmstat`) for 15 seconds to obtain an overview of system load before benchmark execution. It then stores the collected information together with the benchmark results.

Benchmark execution

Using three nested loops, the master process iterates possible node options, process per node options and operations, and commands the chosen workers to run their subtasks. The remaining workers idle until the next benchmark iteration.

Every subtask consists of three phases: a setup phase (`prepare`), the main benchmark phase (`doBench`) and a cleanup phase (`cleanup`). In this way, any dependencies between different operations can be eliminated, if required.[§] At the beginning and end of every phase, an MPI barrier is used to ensure that all processes start and complete simultaneously. In this manner, all time intervals begin at the same time.

In the setup phase, the system is prepared to fulfill any required pre-conditions. For example, a number of test files for the following operations can be created (see Listing 3.1): the main benchmark phase executes a given number of operations. A dedicated thread running in parallel to the benchmark measures and logs the progress over time for each worker with a default time interval of 0.1 seconds. Finally, a cleanup task is executed that can remove any unneeded test data.

Every operation has its own benchmark plugin represented by a separate Python class. This plugin is called dynamically (by name) from the framework code.

3.3.4 Process placement in mixed clusters

After the operator starts a sufficient number of processes in the available system, DMetabench determines the node-process relationship automatically. It builds a matrix of nodes and the number of processes on each node, and then determines the maximum number of nodes and the maximum number of processes per node possible. In this way, it is possible to test several different placements with only one benchmark call.

Depending upon the number of nodes and CPUs per node, several configurations are possible (see Fig. 3.8).

[§]This means that is not necessary to run a benchmark that creates files before running another one that deletes them, thus differing from IOzone where read tests can only be performed after performing a write test that creates a test file.

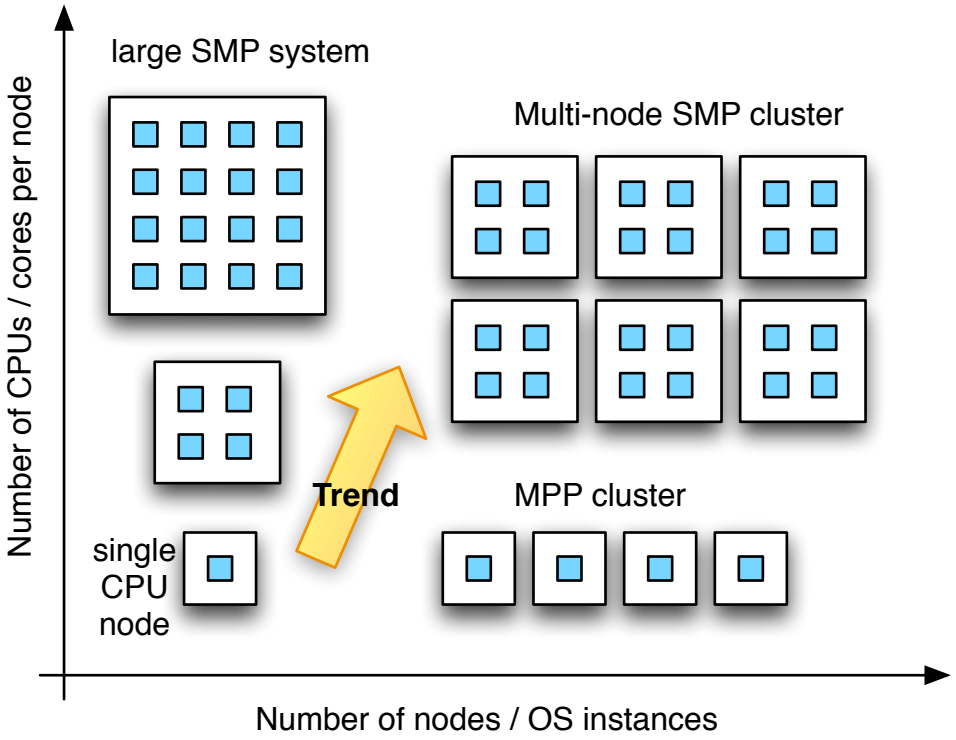

Figure 3.8: Differentiation between multi-core and multi-node setups is essential for metadata benchmarking. The trend is toward more cores per node.

Listing 3.1: Example code plugin for the StatFiles benchmark

```
class StatFilesBenchmark(ManipulateObjectsBenchmark):
  def prepare(self):
     # call parent class prepare
     super(StatFilesBenchmark, self).prepare()
     # create enough test files
     for i in range(0, self.param_dict["problemsize"]):
        f = open (str(i), "w+" )
        f.close()

  def doBench(self):
     self.progress = 0
     for i in range(0, self.param_dict["problemsize"]):
        # perform one operation
        os.stat (str(i))
        # update progress indicator - supervisor thread will read this value
        self.progress = i

  def cleanup(self):
      # remove all test files
      for i in range(0, self.param_dict["problemsize"]):
        os.unlink (str(i))
      # call parent class cleanup to delete working directory
      super(StatFilesBenchmark, self).cleanup()
```

Single-node and SMP

In a single-node setup, the master and all worker processes run on the same node and are managed by a single operating system instance. When running DMetabench in this configuration the ability to submit file system operations in parallel and the caching and locking mechanisms inside of the operating system can be examined. Additionally, local file systems can be tested.

MPP clusters

The other extreme is an MPP cluster where every node has only one CPU and its own instance of the operating system. In this instance, DMetabench will stress the ability of the distributed file system to coordinate operations from multiple nodes.

Multi-node SMP configurations

Multi-node SMP configurations are the most versatile environment for DMetabench because they enable the simulation of setups with multiple nodes and several processes on each node. In fact, the single-node SMP and the MPP cluster configurations are special edge cases of the multi-node SMP setup. The multi-node SMP setup is typical for current HPC clusters.

Process ordering

DMetabench first counts the number of processes on each node and then selects the one (if any) with the highest number of processes to host the master process (see Fig. 3.9). In this way, the largest possible number of processes per node can be utilized because one less MPI slot is available on the node hosting the master.

The remaining processes are then ordered. First, one process from each node is selected while iterating over all nodes, then a second process and so forth. This corresponds to the order in which the benchmark is later run. The exact process order is only significant for the selection of working paths during manual data placement.

DMetabench relies on services provided by MPI to provide barriers and communication between master and worker tasks. Because MPI is inherently process-based, every worker requires a separate operating system process and it is not possible to use threads instead of processes.

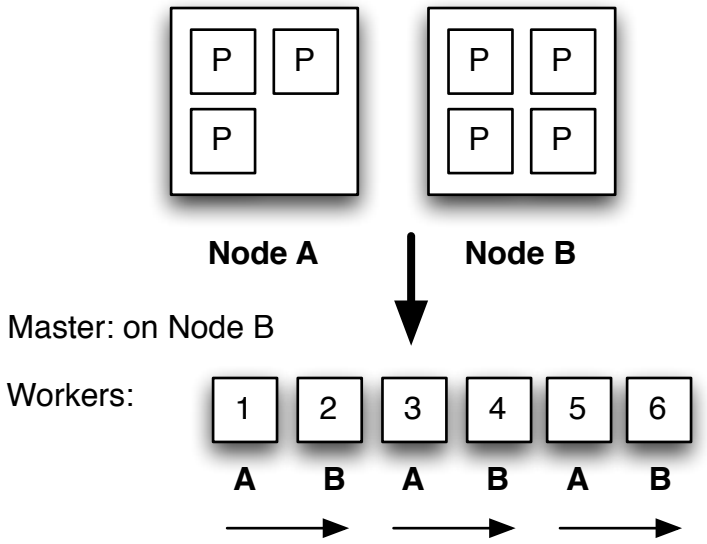

Figure 3.9: Example of a process order for seven MPI processes started on two nodes

3.3.5 DMetabench parameters

Table 3.4 provides an overview of parameters which specify details of a DMetabench benchmark run.

Implicit parameters, which relate to the environment provided by MPI, include the number of processes and their location on (possibly) multiple nodes. Practically, the number of processes for most MPI implementations can be chosen when calling the `mpirun` com-

Table 3.4: Implicit and explicit parameters in DMetabench

Implicit parameters
Number of available MPI slots (processes)
Placement of MPI processes on nodes
Explicit parameters
Node step
Process-per-node-step
Target directory or per-process path list
List of operations to perform
Problem size

mand.[¶] The location is defined in a *hostfile* that contains the mapping of slots to available nodes. Again, this placement cannot be influenced by DMetabench after startup.

As a program, DMetabench takes additional parameters. Two *step* parameters allow reduction of the number of node/process combinations tested. For example with a step of 5, DMetabench would measure on 1,5,10,15,.. nodes instead of 1,2,3,...,15,16,.. etc., and thus greatly reduce the runtime of a benchmark. The target directory is described in section 3.3.6. Every benchmark run can test multiple operations. The problem size parameter is often used inside the benchmark code itself as the number of operations (see Listing 3.1).

Listing 3.2 shows a sample invocation of DMetabench using an MPI environment.

Listing 3.2: Sample invocation of DMetabench using MPI

```
mpirun -np 15 dmetabench.py \
            --ppnstep=5 \
            --problemsize=10000 \
            --operations MakeFile,StatFiles \
            --workdir=/mnt/nfs/testdirectory \
            --label=first-nfs-benchmark \
```

3.3.6 Test data placement

As a default DMetabench uses a test directory specified by a parameter. This directory is then filled with test files and subdirectories, depending upon the particular benchmark. By this means, all processes share a common ancestor directory in the file system hierarchy.

To properly test namespace-aggregated file systems, it is also possible to specify a path list where one directory is specified for each process. In this case, DMetabench matches path entries from the list to the processes by iterating over the ordered list of processes created by the placement technique described above (see Fig. 3.10).

[¶] For example `mpirun -np 25 dmetabench.py...` for 25 MPI slots

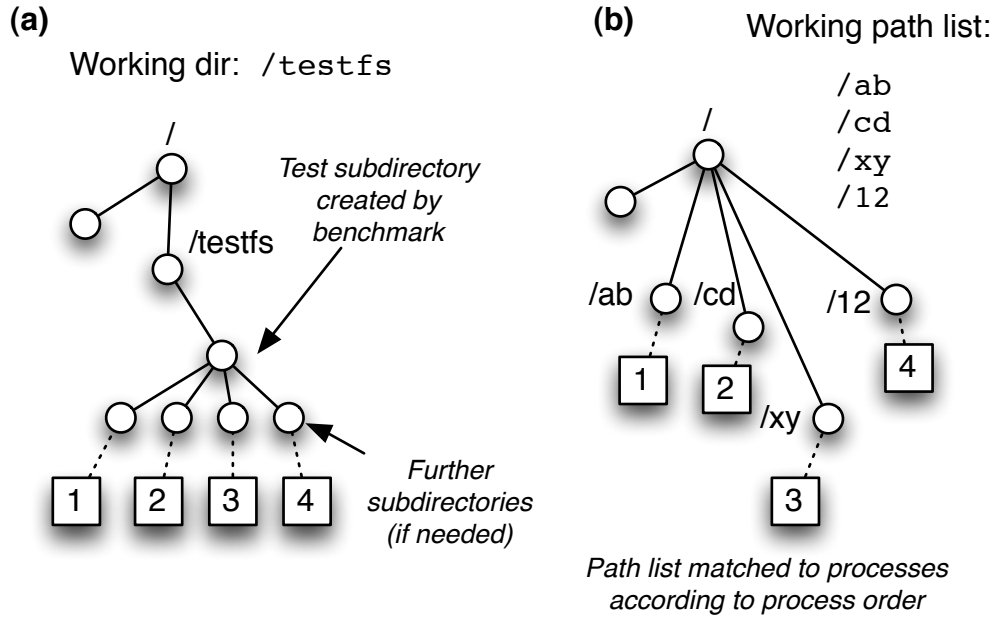

Figure 3.10: Default (a) and explicit (b) specification of working paths in DMetabench

3.3.7 Problem size and benchmark runtime

Section 3.2.3 discussed some of the problems that can appear if a fixed problem size is linearly scaled up with a large number of processes: If one process performs 5,000 operations, then 160 processes would perform 160*5,000=800,000 operations. Therefore there would be a large difference in runtime between 1 and 160 processes.

Depending upon the benchmark, it may be possible to run the measurement for a fixed amount of time (e.g., 60 seconds). This time period can be supplied to and supervised by the supervisor thread, which would simply interrupt execution after the time had elapsed. In this way, time-based effects (cache flushing) could be fully observed. Additionally, every measurement would have approximately the same runtime[‖] and depend upon neither file system performance nor the number of processes used. This would make planning benchmarks (e.g., allocating time for submission in batch queuing systems) much easier.

This method can only be used if the measurement does not depend on a precondition, as it is in the MakeFiles benchmark, which starts with no data. In contrast, a DeleteFiles operation requires a certain number of test files as a prerequisite, and there is no easy way to estimate how many files will be deleted within a fixed period of time. Here DMetabench would use the normal approach with a fixed problem size per process.

Of course, a combination of both methods (e.g., creating test files during a fixed period of time and then deleting all of them) could also be implemented in the future. Here it comes

[‖] The time needed for cleanup can still differ.

as an advantage that it is not necessary that all processes perform the same amount of work, as intermediate progress and point-in-time performance could be easily calculated using the time-log data.

Internal metadata scaling

An additional problem arises with large problem sizes, particularly when the total number of operations rises into the hundreds of thousands. Section 2.4.2 discussed the how putting large numbers of directory entries into a single directory can impact performance because of the scaling properties of the directory itself.** Typically, combining this effect with scaling effects is undesired. To avoid it, the size of directories could be limited. For example, the MakeFile benchmark could run for 60 seconds and use the *problem size* parameter as a size limit for a single directory. If 12,000 files were created during the 60 seconds with a given *problem size* of 5,000, there would be two directories with 5,000 files each and one with 2,000 files. The additional overhead of creating one directory every few thousands of files can usually be neglected.

3.3.8 Pre-defined benchmarks available in DMetabench

DMetabench comes with the pre-defined benchmark plugins listed in Table 3.5. While most are quite simple, the tests on real-life filesystems presented in Chapter 4 demonstrate that even primitive operations can be a rich source of information about the capabilities of a file system. It is easy to add custom operations to DMetabench or modify how existing plugins work. Listing 3.1 shows the complete implementation of a benchmark plugin.

3.3.9 Data preprocessing

The original benchmark data recorded by DMetabench is multi-dimensional: For every combination of a number of nodes, a number of processes per node and every operation there is a complete performance trace for every process participating. Data for all the processes of a particular run is collected after every run and written by the master process to a correspondingly named†† common result file that contains the operation, a set of timestamps and the number of operations processed at the corresponding point in time for every process that was part of the measurement (see listing 3.3). This data can be subsequently processed and compared.

The summary data for all processes is computed in a preprocessing step. The basic algorithm sums all operations completed by particular processes using the principle shown in

**This effect is shown in section 4.3.3.

††For example if the StatNocacheFiles operation is performed with four processes on two nodes the filename is `results- StatNocacheFiles-2-4.tsv`.

Table 3.5: Pre-defined benchmarks in DMetabench

Name	Description
MakeFiles	Every process creates its own working subdirectory or uses the path list given as a parameter. MakeFiles runs for 60 s and creates as many empty files as possible using `open()/close()` system calls. The *problem size* parameter is used to control the maximum number of files after which a new subdirectory is created.
MakeFiles64byte	Similar to MakeFiles but writes 64 bytes into the file (special benchmark for WAFL allocation tests).
MakeFiles65byte	Similar to MakeFiles but writes 65 bytes into the file (special benchmark for WAFL allocation tests).
MakeOnedirFiles	All processes create files in a common directory. The problem size is the total number of file created and for n processes, every process creates $1/n$ of the problem size.
MakeDirs	Analogous to MakeFiles but creates directories with `mkdir()`.
DeleteFiles	Creates a number of test files (according to *problem size* for every process) and then measures the delete performance using `unlink()`.
StatFiles	Creates a number of test files (according to *problem size* for every process) and then measures the attribute retrieval performance using `stat()`.
StatNocacheFiles	Similar to StatFiles but drops operating system caches caches (see section 3.4.3).
StatMultinodeFiles	Similar to StatFiles but measurement is performed on another node (needs at least two nodes, see section 3.4.3).
OpenCloseFiles	Creates a number of test files according to *problem size* for every process and then performs an `open()/close()` pair.

Listing 3.3: Example result file from a benchmark run with four processes on two nodes (the corresponding filename is `results-StatNocacheFiles-2-4.tsv`)

```
Hostname        Operation         ProcessNo  Timestamp  OperationsDone
lx64a153        StatNocacheFiles        0        0.1     1
lx64a153        StatNocacheFiles        0        0.2     569
lx64a153        StatNocacheFiles        0        0.3     1212
[...]
lx64a153        StatNocacheFiles        0        0.8     4411
lx64a153        StatNocacheFiles        0        0.9     5000
lx64a153        StatNocacheFiles        1        0.1     1
lx64a153        StatNocacheFiles        1        0.2     550
lx64a153        StatNocacheFiles        1        0.3     1163
[...]
lx64a153        StatNocacheFiles        1        0.9     4977
lx64a153        StatNocacheFiles        1        1.0     5000
lx64a140        StatNocacheFiles        2        0.1     1
lx64a140        StatNocacheFiles        2        0.2     547
lx64a140        StatNocacheFiles        2        0.3     1166
[...]
lx64a140        StatNocacheFiles        2        0.8     4351
lx64a140        StatNocacheFiles        2        0.9     4995
lx64a140        StatNocacheFiles        2        1.0     5000
lx64a140        StatNocacheFiles        3        0.1     24
lx64a140        StatNocacheFiles        3        0.2     624
lx64a140        StatNocacheFiles        3        0.3     1266
[...]
lx64a140        StatNocacheFiles        3        0.8     4475
lx64a140        StatNocacheFiles        3        0.9     5000
```

Figure 3.4). The result is the number of total operations completed for every time interval and the total performance during this time interval. The total performance in a time interval, given in operations per second, is the difference between the total number of operations completed during two adjacent time intervals (see Listing 3.4).

To determine the differences in performance between particular processes, the standard deviation and coefficient of variation (COV) of the per-process performance is calculated. The COV, a measure of how much the performance between particular processes differed during a time interval, is defined as the ratio of the standard deviation to the mean of the per-process performance.

Finally a total performance average is calculated using the stonewall principle. The first point in time during which at least one process had completed is identified and the average performance is then calculated up to that point. In the example listing 3.3, two processes complete after 0.9 seconds. At the point in time in question in the example 19,972 operations had completed, so the stonewall average performance is 19,972 operations divided by 0.9 seconds = 22,191 operations per second. In this particular example, the total runtime of the benchmark is extremely brief in comparison to the time interval: for most measurements, the

Listing 3.4: Summary runtime data computed for the input data of listing 3.3. The colums from left to right contain the operation, number of nodes, number of processes, the timestamp, the total number of operations completed, the standard deviation of per-process completed operations and the coefficient of variation of per-process completed operations.

```
StatNocacheFiles        2       4       0.1     27      0       0.0     0.000
StatNocacheFiles        2       4       0.2     2290    22630   24.8    0.044
StatNocacheFiles        2       4       0.3     4807    25170   15.5    0.025
StatNocacheFiles        2       4       0.4     7316    25090   16.2    0.026
StatNocacheFiles        2       4       0.5     9866    25500   2.6     0.004
StatNocacheFiles        2       4       0.6     12443   25770   1.0     0.001
StatNocacheFiles        2       4       0.7     14994   25510   2.5     0.004
StatNocacheFiles        2       4       0.8     17568   25740   0.6     0.001
StatNocacheFiles        2       4       0.9     19972   24040   57.1    0.095
StatNocacheFiles        2       4       1.0     20000   280     10.9    1.561
```

ideal time is approximately one minute, which provides approximately 600 time intervals of 0.1seconds. In addition to the stonewall average, additional averages can be calculated, such as the average for a fixed number (e.g., 10,000) of operations, in a manner that resembles the 'strong scaling' model discussed above. This summary result is stored in a file common for all operations and combinations of nodes and processes (see Listing 3.5).

Listing 3.5: Performance summary obtained for input data from listing 3.3. The columns from left to right contain the operation, the number of nodes, the number of processes per node, the total number of processes, the stonewall average performance and the performance for 10,000 resp. 25,000 operations.

```
StatNocacheFiles        2       2       4       22191   20738   0
```

3.3.10 Graphical analysis

DMetabench is able to visualize preprocessed data using three different types of charts. To review, there are five basic dimensions in DMetabench of which the performance results are a function:

1. **Result set:** Each run of DMetabench generates a new result set
2. **Operation:** Different operations can be measured
3. **Node number:** The number of nodes used in a particular measurement
4. **Per-node process number:** The number of processes per node used in a particular measurement
5. **Time:** The performance varies over time.

DMetabench charts present a two-dimensional plane based on the multi-dimensional result data. All other dimensions must be kept constant to be able to plot the data.

Combined time chart

Based on the summary data, DMetabench can generate a chart for a particular operation using a certain number of nodes and processes. This chart shows the operations completed as a function of time corresponding to a selection of fixed values for dimensions 1,2,3 and 4. Additionally, the total performance and the COV, also depending upon time, can be presented graphically. The example in Fig. 3.11 uses the data from Listing 3.4.

Section 3.2.5 discussed different possibilities for obtaining performance averages. The wallclock time throughput in this example is 20,000 operations in 1.0 seconds which yields 20,000 ops/s. While the stonewall performance number (22,191 ops/s @ 0.9s) is better, the bottom chart clearly shows that we clearly see that for most of the period, the performance reached almost 25,000 ops/s with a 'slow start' at the beginning. Neither number is incorrect per se, but this graphical representation gives a more precise indication of the changes in performance, which is precisely the advantage of the method presented.

The middle chart demonstrates the coefficient of variation of per-process performance. At 0.9 seconds two processes have completed, so the COV increases as two processes continued working while the other two have a performance value of zero. Using the COV, similar differences can be visualized for a larger number of processes.

The combined time chart can also be used for detailed analysis of a particular benchmark run. When a summary performance number differs from expectations (e.g., based on one of the two following types of charts), this chart type will help to identify the reasons for this discrepancy (e.g., external disturbances such as other processes competing for I/O). Section 4.2.3 provides several practical examples.

Performance vs. number of processes chart

The second type of chart, the performance vs. number of processes chart, allows an assessment of scaling for a changing number of processes (see Fig. 3.12). Of the input measurement dimensions *result set*, *operation*, *number of nodes* and *number of processes per node*, the first two are selected and then the stonewall average performance (average value of dimension 5) is plotted against a variable number of processes. In this way, each performance value in the chart is the stonewall average of this particular measurement.

This chart type is useful for comparing scaling on a single SMP-type machine. Multiple measurements, possibly obtained from different operations and configurations, can be shown together to facilitate direct comparisons. Some possibilities for comparisons include:

- the same operation on different file systems

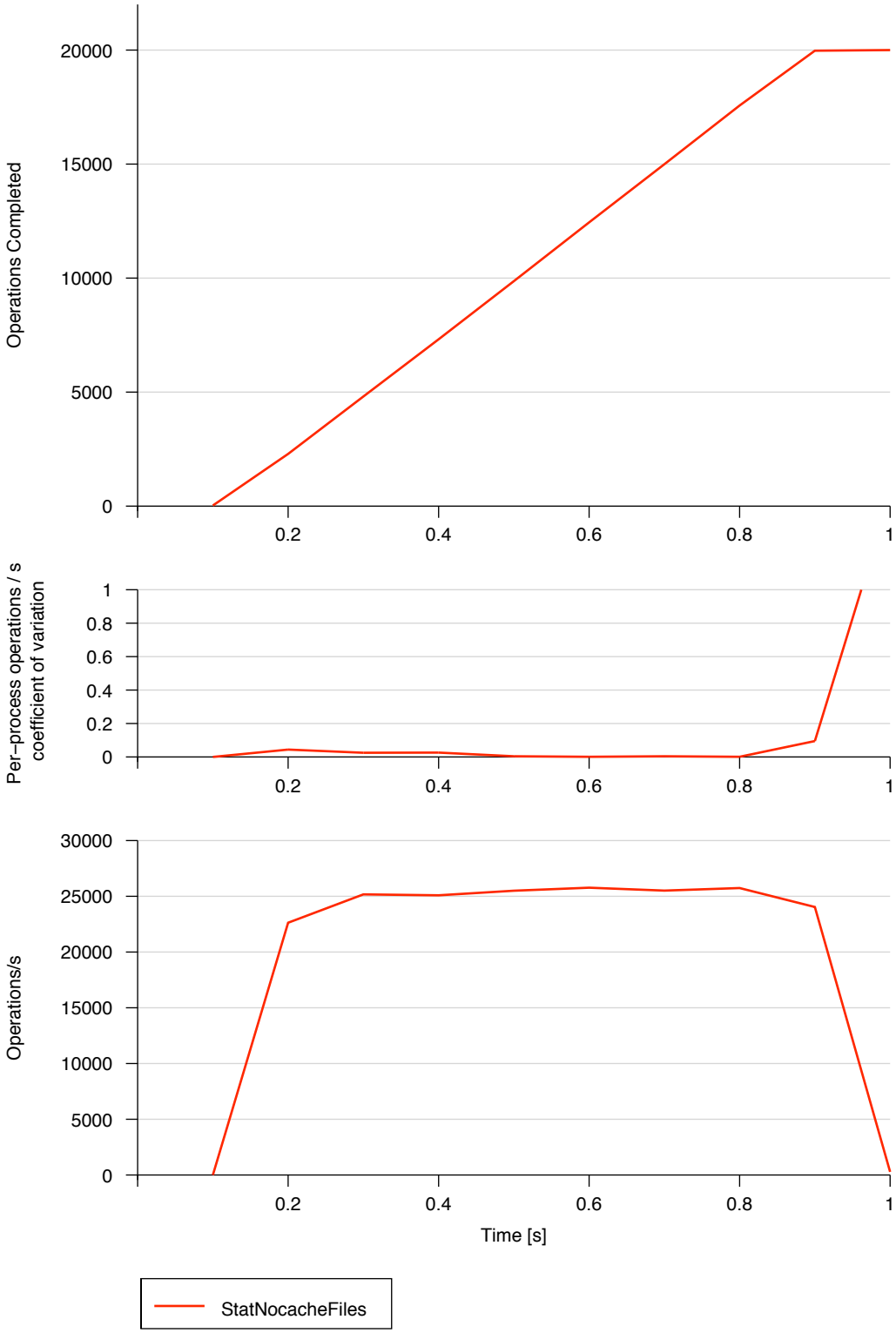

Figure 3.11: Graphical representation of data from listing 3.4 in an automatically generated time chart. The top chart shows the total operations completed, the middle the COV and the bottom the total performance for a given point in time.

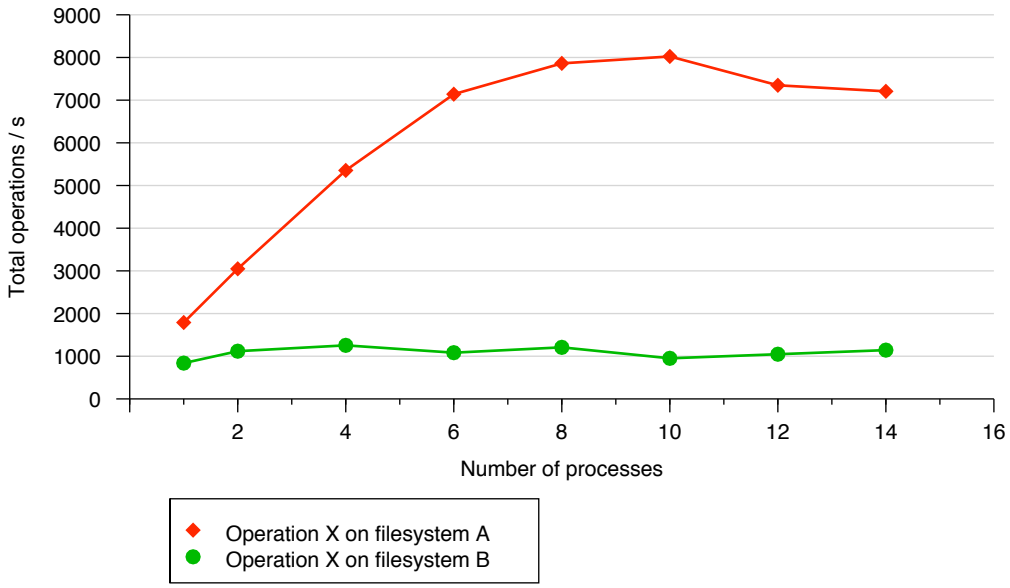

Figure 3.12: Sample chart showing performance for a variable number of processes

- different operations on the same file system
- multiple measurements of the same configuration using identical or different tuning parameters.

Performance vs. number of nodes chart

The third type of chart, the performance vs. number of nodes chart, is quite similar to the previous type, but shows how performance depends upon the number of nodes in an MPP measurement (see Fig. 3.13). Here the result set, the operation and the number of processes per node is fixed. Using this type of chart, the scaling of performance can be compared for multiple node configurations.

Strong scaling comparison

Normally, the performance vs. number of processes chart and the performance vs. number of nodes chart use the stonewall average performance from the complete measurement. Alternatively, it is also possible to use average performance for a fixed number of operations, which is also available after the preprocessing step (see Listing 3.3). In this way, the focus of a chart can shift from a global file system view (weak scaling) and instead enable conclusions regarding the advantages of parallelism for a problem of a fixed size (strong scaling).

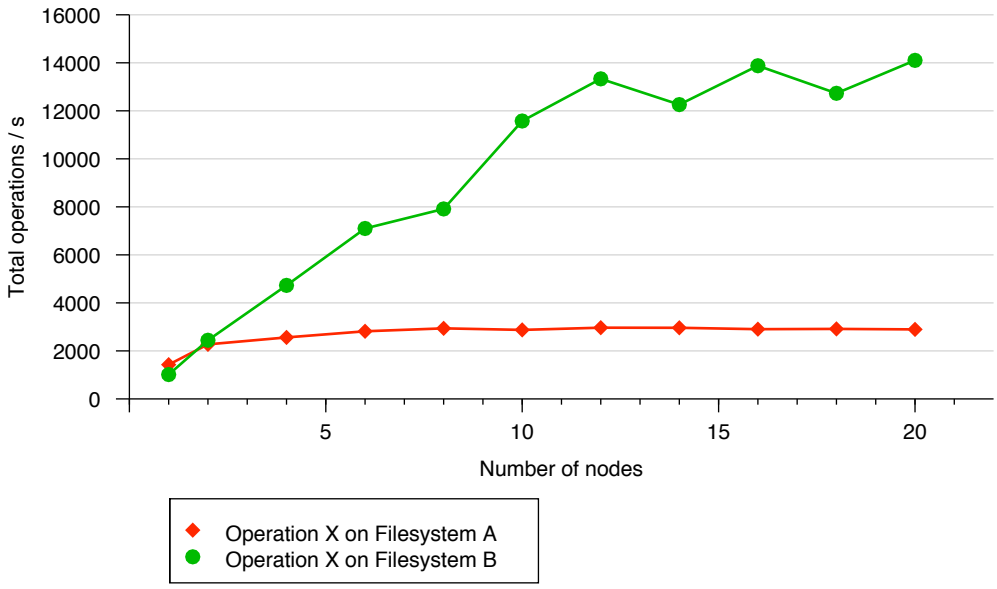

Figure 3.13: Sample chart showing performance of an operation on two different filesystems as a function of the number of nodes

3.4 Implementation details

3.4.1 Programming language and runtime environment

DMetabench has been implemented using Python and a matching Python-MPI binding called PyPAR. While this combination is less common in parallel environments, it offers clear advantages in extendibility, portability and ease of use compared to languages such as C.

Python

The programming language Python, developed by Guido Rossum during the 1990s, offers a clean and concise syntax, powerful data structures and an interface to C [RD03]. Python is an interpreted language and ports of the interpreter are available for all common operating system platforms like Linux, Mac OS X, AIX, Solaris and Windows.

PyPAR: A Python-MPI binding

To enable interaction with the MPI environment PyPAR was used. PyPAR is a thin wrapper layer for the MPI library that makes basic MPI functions available to python. In addition to the MPI barrier function, it is possible to easily send and receive complex Python data structures, used for control and result processing, between processes. PyPAR is the only

platform-specific component of DMetabench that must be compiled for the target MPI platform. Details of the PyPAR implementation are discussed in [Nie07].

Parallelism in Python

The current 2.4 version of the Python interpreter uses a Global Interpreter Lock (GIL) which effectively guards against any modification to Python structures and prevents real parallelism when using Python threads [Pil90]. Fortunately, this is not a problem with DMetabench because it uses completely separate Python interpreters running in different operating system processes. The only place where Python-level thread parallelism is needed is between the benchmark and the supervisor thread. As the GIL is released before potentially blocking operations external to Python, such as I/O operations, the supervisor thread is not hindered upon running.

3.4.2 Chart generation

All DMetabench charts are generated using Ploticus [Plo08] which is essentially a charting software quite similar to gnuplot [GNU08]. To create the three chart types, three Python programs (`compare.py`, `compare-process.py` and `compare-node.py`) take the fixed dimensions (e.g., the location of the result set) for every measurement to be compared and subsequently generate a control file for Ploticus (see Fig. 3.14). A sample invocation is shown in Listing 3.6: the fixed dimensions of the measurement selected and labels for the charts are specified on the command line.

Using the control file Ploticus reads the summary data generated in the preprocessing step and produces an image file in one of the supported file formats (e.g., EPS). Ploticus handles axis scaling automatically.

Listing 3.6: Sample invocation of `compare-process.py` that could have been used to generate Figure 3.12

```
compare-process.py "result-dir-of-test1:MakeFiles:1:Operation X on filesystem A" \
                   "result-dir-of-test2:MakeFiles:1:Operation X on filesystem B"
```

3.4.3 Controlling caching

Understanding caching is necessary to obtain meaningful results in file system benchmarks. Both write and read caches can potentially accelerate operations by several orders of magnitude and with larger systems it is often impossible to influence all caches. The following section explores caching opportunities in a distributed file system and presents several options for dealing with undesirable cache effects.

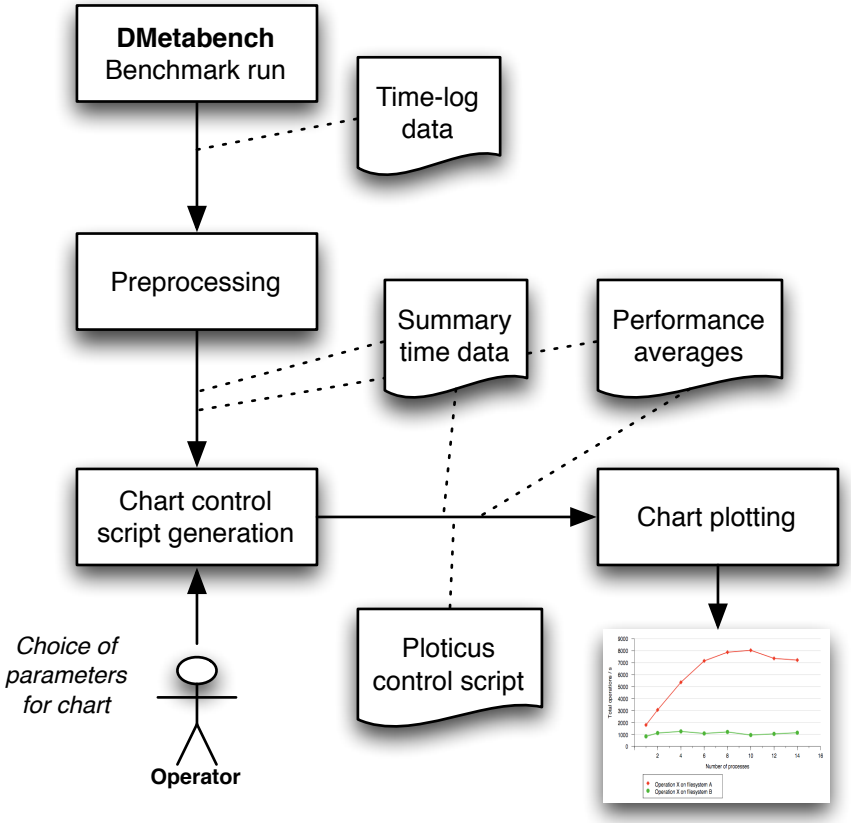

Figure 3.14: Result processing workflow in DMetabench

Storage layers

The goal of caching is to avoid the need to perform expensive operations, such as those related to network communication or physical disk access. Cache memory is usually filled when writing and accessed when reading data, but the caching of read and write operations is not necessarily connected. Caching semantics for write operations include *write-through*, which signals completion when the write operation has been completed in the next layer, and *write-back*, which reports an operation as completed as soon as the data has been written to the cache. A read cache can be filled by means of write operations, explicit read operations or implicit reads (e.g., through read-ahead).

In a distributed storage system consisting of clients, servers, storage controllers and disks data can be potentially cached in all layers of the storage stack. Hard disks already include a small (tens of MB) data cache. In professional storage systems, write-back caching is usually disabled at the disk level. The next layer involves storage and RAID controllers that use cache sizes from one to hundreds of GB. Here the cache is often required to aggressively optimize redundancy calculation for RAID. When a distributed file system includes storage servers, these servers can also maintain write and/or read caches. Finally, there is an operating system cache on each client that, depending upon the filesystem, can hold both data and/or metadata.

Benchmarking and caches

When data is written to a distributed file system (e.g., new files are created), write-back caching occurs at least in the storage system. This process is more a question of data persistence semantics than performance, as the same effects also occur for real applications; therefore, it is reasonable to measure them with a benchmark. On the other hand, when artificial test data is read back, there is a strong chance that it is still inside a cache as it had been written shortly before being read back. In this case, any operations accessing the data can run much faster than would be expected from real applications.

To decrease the impact of this effect, a manner of removing cache contents or avoiding their use must be identified. As a software application running on file system clients, DMetabench cannot influence the caches of disks, storage systems or file servers. This means that a large storage system could, at least theoretically, perform an entire benchmark run inside its cache with no guarantee that the metadata generated will ever hit the disks. While this is a known limitation, its consequences are justifiable: Firstly, using its time-based logging, DMetabench is able to identify the effects of time-based mechanisms which flush write-back caches independently of the amount of data. Second, the forthcoming introduction of solid-state storage will remove the mechanical latency of disk-based storage in the very near future, accelerate the storage hierachy and thus shift the focus back to client-file system interaction.

Caching in the operating system is a different problem, as it might lead to a case whereby operations are performed from cache without any interaction with the distributed storage. This hinders benchmarking metadata performance, because the distributed file system would no longer be involved in the measurement. To prevent this situation, the use of operating system caches for read benchmarks (e.g., StatFiles) should be avoided.

Removing cache contents

One way to clean caches is to re-mount (unmount and mount) the distributed file system, which can be performed on most filesystems (AFS, with its persistent disk cache being an exception). However it requires not only superuser privileges but also that no other processes are allowed to access the filesystem: otherwise, the filesystem will be marked as busy. The latter problem makes this option difficult for larger environments.

In Linux kernel versions beginning with 2.6.16, there is a dedicated sysctl parameter (`/proc/sys/vm/drop_caches`) that can be used to explicitly remove the contents of the page cache, cached dentries and inodes. It still requires superuser access, but this can be overcome using a suid wrapper program. In DMetabench, this technique is used in the StatNocacheFiles-Benchmark wherein the caches are dropped after all test files in the `prepare` phase have been created. In comparison to the simple StatFiles benchmark (as shown in listing 3.1), only a single call to the suid wrapper "dropcaches" program is added at the end of the `prepare` phase.

Bypassing caches

With a multi-node setup, it is also possible to bypass OS cache contents by creating test files on one node and performing operations on another node. As the other node does not 'see' these files, they are not in its operating system cache and must be fetched from the distributed file system. This method is implemented in the StatMultinodeFiles benchmark, where every worker process is matched with a peer process running on another node and both processes exchange their file sets between the `prepare` and `doBench`-phases.

File system aging

File system aging describes the effect that occurs when filesystem metadata is not written and read sequentially, as seen in the lifecycle of real production file systems [Tan95]. Unlike in DMetabench, creation and removal operations are interchanged so that the spacial locality of internal metadata information on the storage system differs from a benchmark situation.

DMetabench takes no explicit measures to simulate the aging of a file system. The author of this thesis argues that with the increasing use of large NVRAM caches and solid-state

storage devices, the spacial locality of data on a storage medium will not play a significant role in the near future.

3.5 Summary

DMetabench is a simple but effective parallel metadata benchmark framework that permits measurement of typical metadata operations at the operating system API level. Its distinctive feature is its ability to monitor operation progress during measurement to obtain additional time-based information not available in existing benchmarks. Using custom code plugins the DMetabench framework can be simply extended or modified for use with rapid, experimental benchmarking or for tuning file systems.

Chapter 4

File system analysis using DMetabench

DMetabench, a parallel metadata benchmark framework presented in Chapter 3, was used to assess different aspects of several distributed file systems. The main objective was to both demonstrate the manner in which DMetabench differs from existing benchmarks and identify concepts and implementation details that influence metadata performance. For these purposes, benchmarks were run on large production file systems at the Leibniz Supercomputing Center (LRZ) in Munich.

This chapter starts with a system-level evaluation of DMetabench and then demonstrates how DMetabench can be used to evaluate the metadata performance of real file systems using six examples selected to highlight its novel capabilities:

- A comparison of NFS and Lustre in a cluster environment
- Testing priority scheduling and metadata performance
- Intra-node scalability on SMP systems
- Influence of network latency on metadata performance
- Intra-node and inter-node scalability in namespace aggregated file systems
- Write-back caching of metadata.

4.1 System environment used for benchmarking

All measurements presented were have run on production files systems available at LRZ. As an European supercomputing center providing services to local and national researchers from different scientific fields, the LRZ offers a heterogenous multi-architecture Linux cluster

for low- to midrange applications and the HLRB2, a world-class Linux-based supercomputer for grand-challenge type high performance computing.

4.1.1 Storage system design rationale

Experience with previous HPC systems has shown that there are two different types of data used in HPC environments (see Table 4.1). On the one hand, input- and output files as well as application binaries and source files are smaller, but often accessed interactively by users. On the other hand, enormous amounts of data are generated automatically by HPC applications in the form of checkpoint or intermediate files. These types of data differ greatly regarding data and metadata requirements as well as expected reliability. It is for these reasons why HPC systems at LRZ have traditionally used two different file systems.

Table 4.1: Data set classification at the LRZ

Data set	**Application software** **User source codes** **User input-/output files**	**Intermediate results** **Checkpoint files**
Number of files	High	Low - medium
Size of files	Small	Medium - Large
Access bandwidth	Less important	Highly important
Metadata operations	Highly important	Less important
Main type of access	Sequential (single-node)	Parallel (many nodes)
Reliability	Files must not be lost	Files can be re-calculated
Space allocation	Per-User-/Per-Project quotas	High-watermark deletion
Backup	Snapshots and backup to tape	No backup

4.1.2 The LRZ Linux Cluster

As of 2008, the LRZ Linux cluster contains approximately 800 compute nodes grouped in pools of identical machines. Different CPU architectures (IA32, IA64, x86_64) and hardware types are available (see Fig. 4.1). Several pools are dedicated to serial (single-CPU) tasks and parallel applications using MPI, OpenMP or multi-threaded programming models. The allocation of computing resources is managed using Sun Grid Engine (SGE) as a batch-queuing system. Interactive login nodes, usually one from each hardware model, allow users to compile, test and submit their programs to SGE. All nodes are connected using a 1-Gigabit or 10-Gigabit-Ethernet-based network with a 10-Gigabit-backbone infrastructure.

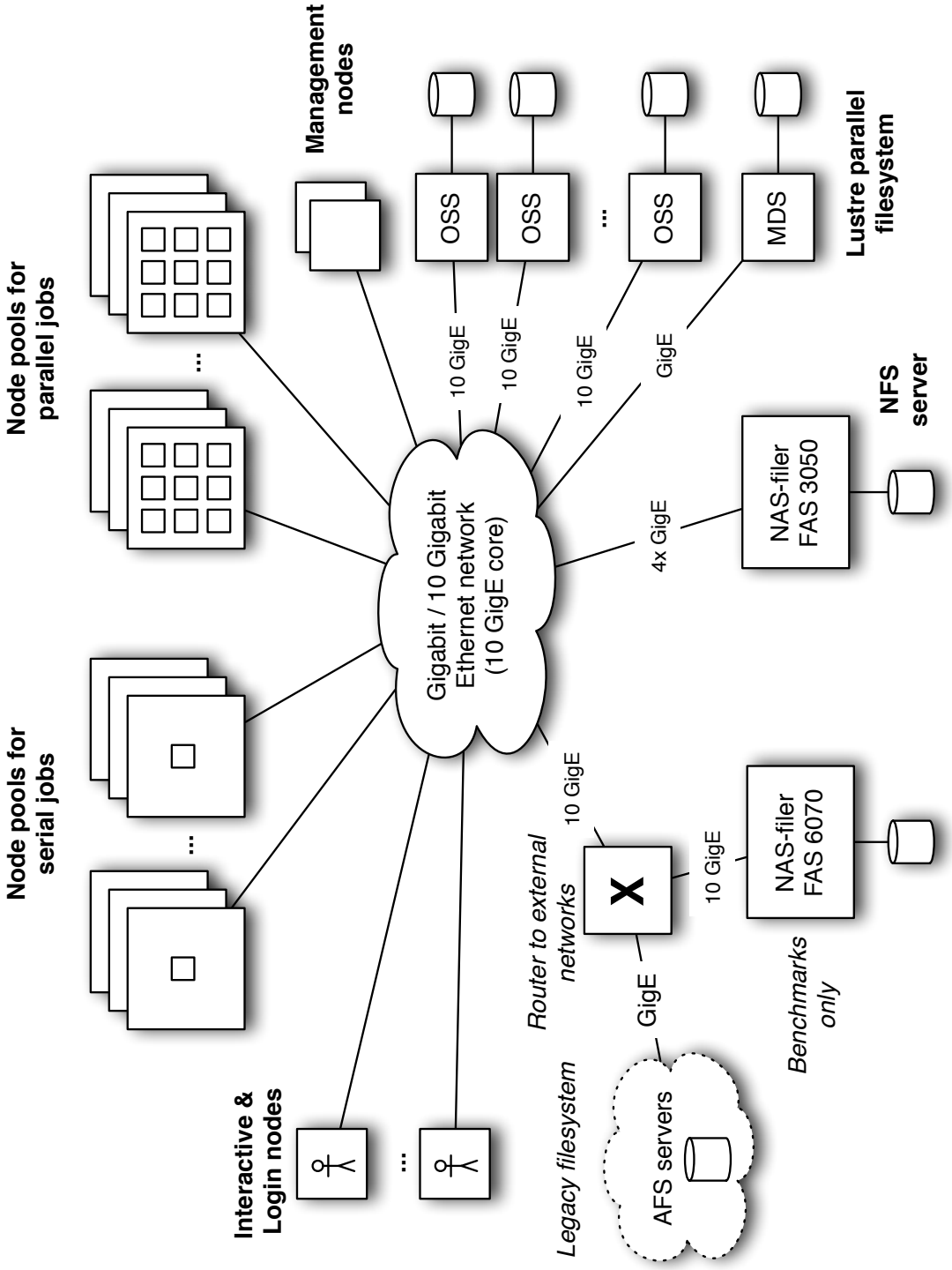

Figure 4.1: Storage and file system configuration in the LRZ Linux cluster

NAS storage

On the Linux cluster system software, user home and project directories are stored on a single dedicated Netapp file server that exports the volumes to the cluster using NFS v3. The Netapp system is a single FAS 3050 server using the WAFL file system, dual Xeon 2.8 GHz CPUs, 512 MB of NVRAM and 3 GB of RAM. Connection to the cluster network is enabled using four trunked Gigabit Ethernet connections. The disk configuration consists of 60 FibreChannel disks grouped in four RAID-DP* aggregates with 3x16 and 1x12 144 GB 10k RPM disk providing a total useable capacity of approximately 7 TB. During the benchmark measurements Data OnTAP version 7.2 was used.

To protect data, all file systems are asynchronously mirrored to a separate filer and then backed up to tape. Users can access multiple snapshots of all filesystems to recover data erroneously deleted or damaged. The default user quota on the home file system is 10 GB and 100,000 files, sufficient for program sources, binaries and input/output data and typical workloads including program compilation and application startups. This NFS server has a maximum sequential throughput of approximately 100 MB/s.

For several of the benchmarks presented here, an additional FAS 6070 NAS filer was used, which was faster than the production system of the Linux cluster. This file contains four Opteron CPUs running at 2.4 GHz, 2 GB of NVRAM and 32 GB of RAM. It is connected to the network using 10-Gigabit-Ethernet. Tests were performed on a 3x16 disk RAID-DP aggregate using 300 GB 10k FC disks.

Parallel file system

Large intermediate and temporary files generated by compute jobs are stored in a Lustre file system based on twelve Object Storage Servers (OSS) and a metadata server (MDS). During the benchmarks Lustre version 1.6.2 was used. The total net system capacity is 130 TB and the maximum sequential data throughput for reads or writes is approximately 5 GB/s.

There are no quotas on the Lustre file system, but a high-watermark based deletion is enabled as soon as the file system becomes more than 80% full to delete old and large files until only 60% remain.

Legacy file systems

For many years, the Linux cluster had been using AFS for home directories and applications. Due to multiple factors† AFS was phased out as the primary user file system, replaced by

*RAID DP is a variant of RAID 4 with two parity disks. In an aggregate with, for example, 2x16 disks there are 2x14 data and 2x2 parity disks.

†These relate primarily to problems with building AFS client software for current kernels and hardware architectures but also include the inability to transparently continue file system access after network problems.

NFS and then completely removed from the Linux cluster. However, some benchmarks presented later were run while AFS was still available.

Compute nodes

While the Linux cluster contains many different types of hardware, most benchmarks were run on dual-quadcore Intel processor nodes with 32 GB of RAM and a 10-GigE network connection. The operating system used was SuSE Linux Enterprise Server (SLES) 10 using the Linux kernel 2.6.16 with suitable kernel modules for Lustre.

4.1.3 The HLRB II

The SGI Altix 4700-based supercomputer at the Leibniz Supercomputing Center is called 'HLRB 2'. It consists of 19 large SMP partitions with 512 Itanium2 CPU-cores and 2 TB of RAM each. The NUMAlink high-speed interconnect (6.4 GB/s per link) connects both different CPUs in a partition, which creates a NUMA architecture, and also different partitions that enable multi-partition jobs. Eighteen partitions are dedicated to batch jobs and one is reserved for interactive work. To make process scheduling and memory allocation easier every batch job runs in dedicated cpusets.[‡] Every batch partition has a two-core cpuset responsible for running system daemons and the remaining 510 cores are available for batch jobs.

Ontap GX storage cluster

In a manner similar to those on the Linux cluster, project data and applications on the HLRB2 are stored in NFS-based storage. On the HLRB 2, a dedicated eight-node server cluster running Netapp Ontap GX is used in which every node itself is a FAS 3050. Although a single NFS namespace is presented to the supercomputer, every project (out of more than 120) has its own data volume that can be located on any of the eight servers.

Detailed concepts of Ontap GX are presented in [ECK+07]. All filers in a GX cluster are interconnected with a cluster network in addition to the client-side network. Data volumes that resemble local file systems are managed by D-Blade software components (see Fig. 4.3). At any time, there is exactly one D-Blade responsible for a volume. To form a common namespace and access a volume from other members of the cluster, another component, the N-Blade, translates client requests to an internal protocol and forwards them to the correct D-Blade, which can be either on the same or another filer. Similar to AFS [Cam98], a volume location database (VLDB) is used to match volumes and their locations in the namespace, but the translation process is internal to the cluster and does not require software on the

[‡] A cpuset limits the migration of processes to a number of pre-defined processors with the intent of reducing interferences among multiple compute jobs on the same node.

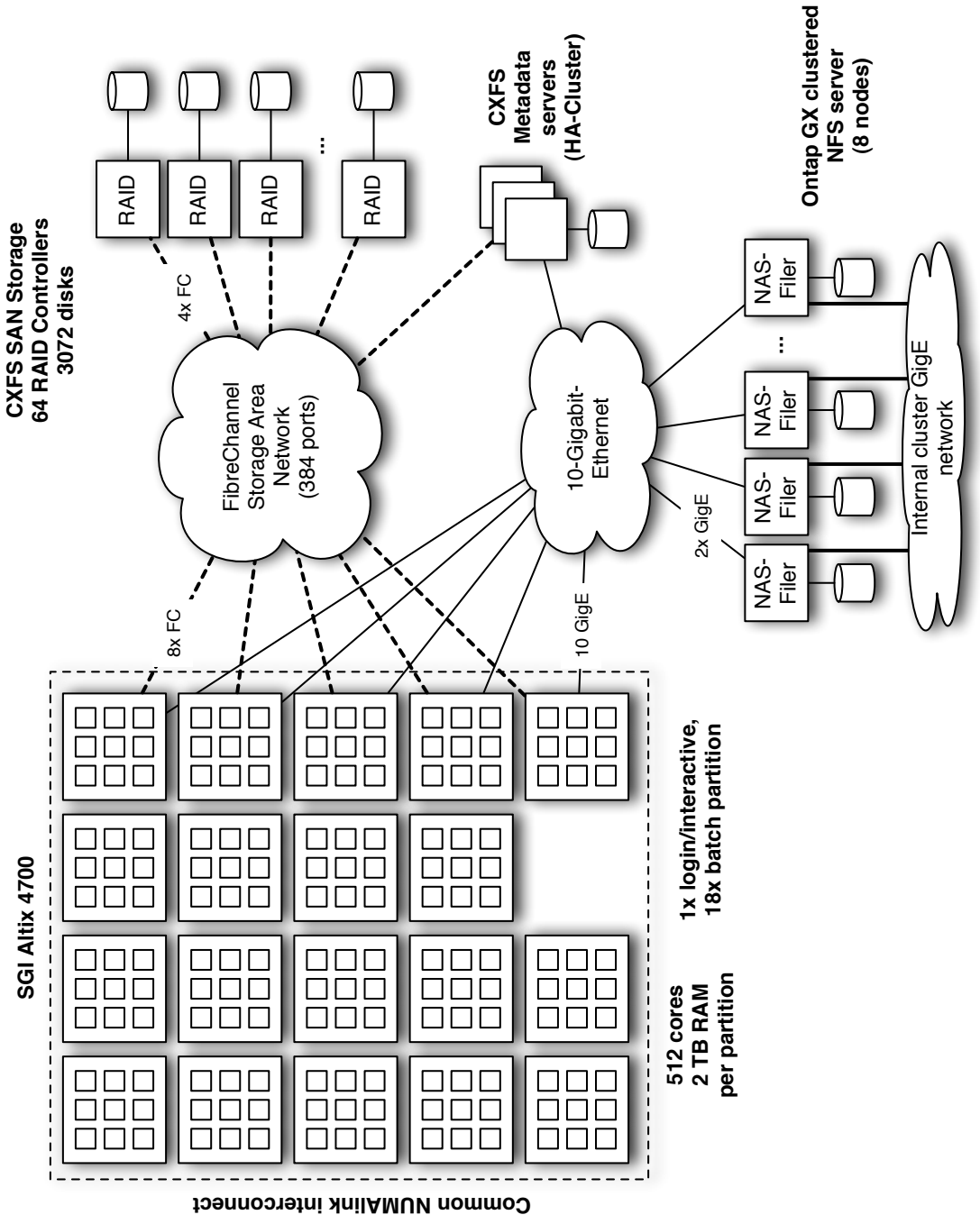

Figure 4.2: Storage and file system setup of the HLRB 2 supercomputer

clients. In measurements cited in [ECK+07], an efficiency of approximately 75% is claimed, even if all requests must be forwarded to another node. By design, a client request must pass at most two nodes: one with the N-Blade and one with the D-Blade.

In the HLRB2 configuration, there are 16 network interfaces (two per filer) with different IP addresses and the 19 partitions of the HLRB2 are almost uniformly distributed among these addresses by a separate mountpoint for every IP address. Every filer utilizes a single aggregate with 2x18 RAID DP disks (300 GB 10k FC).

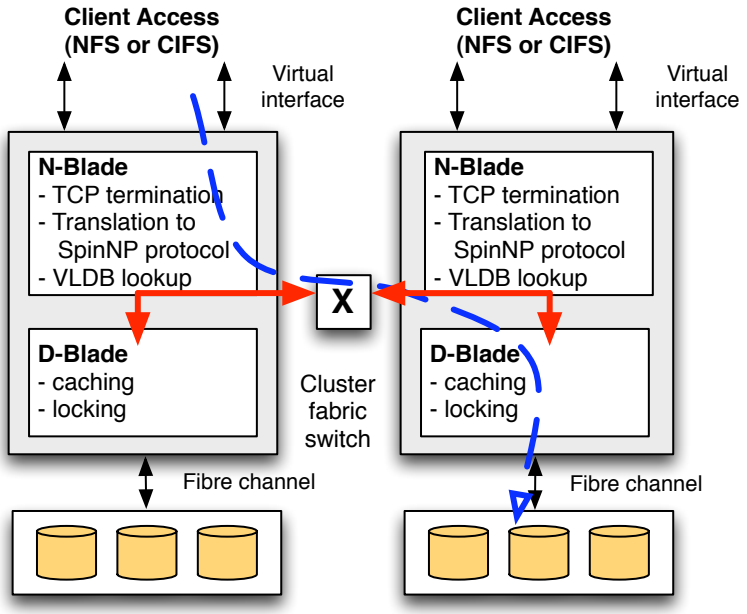

Figure 4.3: In Ontap GX, requests that cannot be satisfied from a volume local to the receiving node are forwarded to the volume owner using a dedicated cluster interconnect [ECK+07].

CXFS file systems

Temporary and intermediate files on the HLRB 2 are placed on one of two CXFS file systems. Every Altix partition is directly connected to a FibreChannel SAN. The storage itself consists of 64 RAID controllers and 3072 disks delivering a net capacity of 300 TB for each of the filesystems. Metadata is managed by a dedicated three-node failover cluster of metadata controllers but only one metadata controller per file system remains active at any one time. During acceptance tests running over both file systems, an aggregate write and read throughput of more than 45 GB/s has been achieved.

4.2 A system-level evaluation of DMetabench

An important initial issue to be determined experimentally is the impact of using a high-level interpreted language, such as Python, in a low-level benchmark framework. Two particular areas of interest are how Python function calls map to the API layer and how performance is changed compared to a pure C implementation.

4.2.1 File function processing

Python provides an object based interface to files (see Listing 4.1).

Listing 4.1: Basic object-style file function in Python

```
# create file object using filename and access mode
f = open ("filename","w")

# write to file object
f.write ("Hello World")

# close file object
f.close()
```

To investigate how these operations map to the corresponding C calls, such as `open()` or `close()`, the source code of the application (in this case Python) can be inspected or, if the source code is not available, tracing tools for system calls can be used. Such tools include `strace`, `ktrace`, `truss` or the more advanced `dtrace` framework available on Solaris and Mac OS X [CSL04]. Here a `dtrace`-Script was used that counted the number of issued system calls while a simple loop of Python `open()` and `close()` calls was running (listing 4.2).

Listing 4.2: Dtrace script tracing for object-style file creation in Python (partial results)

```
dtrace -n 'syscall:::entry { @num[probefunc] = count(); }'

[...]
fstat                                             13255
open                                              13255
close                                             13278
```

Surprisingly, in addition to an equal amount of `open()` and `close()` calls, an equal amount of additional `fstat()` system calls have been observed. This has occured during the initialization of the file object and before the `open()` call while Python checks whether the filename given is a directory (directories are not allowed as file objects[§]). These addi-

[§]See `fileobject.c` in the Python 2.5 source code.

tional calls skew measurements because in many file systems, `fstat()` needs significant time for non-existing files.

Fortunately, Python also provides an `os` module with thinly wrapped operating system calls that include `os.open()` and related functions that map directly to the C API and deliver a file descriptor. These functions do not issue additional system calls and were used in the example benchmarks obtained using DMetabench.

This experience made clear that measuring metadata performance requires a careful assessment of what exactly occurs at the system API layer. Particularly with distributed file systems, any additional system calls can have massive performance implications. Naturally, this is generally also true for normal user applications where high-level constructs can also change performance properties.

4.2.2 Performance comparison of Python and pure C programs

As an interpreted language, Python has some overhead in comparison to compiled languages, C in particular. This overhead was quantified in a simple experiment including a simple loop that created empty files in a file system.

The same loop semantic was implemented using pure C (see Listing 4.3) and Python (see Listing 4.4) and then the runtime was compared on the same system.[¶] To minimize the influence of the file system, the very fast `/dev/shm` in-memory file system was used. The runtime was measured from the command shell level using the `time` command. For Python, an additional time measurement inside the program was used to separate the Python runtime startup from the work loop. The results of both measurements are compared in Table 4.2.

Listing 4.3: Basic loop in C

```
#include<stdio.h>
#include "fcntl.h"
#include "unistd.h"

int i;
char filename[40];
int fd;

main()
{
    for (i=0; i<200000; i++) {
        sprintf(filename,"%d", i);
        fd = open (filename, O_RDWR | O_CREAT);
        close(fd);
```

[¶] Dual Xeon 3.0 GHz server running Linux kernel 2.6.16.

```
    }
}
```

Listing 4.4: Basic loop in Python

```
#!/usr/bin/python
import os,time

t1 = time.time();

for i in range(0,200000):
   fd = os.open(str(i),os.O_RDWR|os.O_CREAT)
   os.close (fd)

print "Time:␣", time.time() - t1
```

Table 4.2: Python vs. C loop runtime (create 200,000 files)

	C version	**Python version**
Wall-clock time	0.62 s	2.1 s
Loop run time		2.0 s

For this micro-benchmark the runtime difference is quite significant: The Python version needs 1.38 seconds longer than does the C version. As both programs perform the same amount of work, this overhead is fixed with respect to the number of operations. If several Python processes work in parallel, as they do in DMetabench, the overhead does not add up.

The implications for DMetabench are as follows: Comparative assessment of different file systems on the same hardware platform is not influenced in any manner, as all measurements are slowed down equally. Qualitatively, the difference depends upon the run time of the benchmark and the number of operations (e.g., with a loop runtime of 40 seconds and a file system which creates 5,000 files per second, the difference is below 4%).

For very precise measurements, the overhead could be eliminated by measuring the runtime of the loop without file system operations and subtracting this time from the total runtime. Alternately, a compiled C module could be included into the framework, but this would eliminate the rapid-benchmarking advantages of using a high-level language.

The benchmark plugins in DMetabench do not use these techniques because the distributed file systems tested were much slower, which reduced the number of operations performed and thus the overhead. Additionally, the main focus of this work was a more qualitative comparison of performance, as quantitative performance is too greatly influenced by continuous improvements in system hardware.

4.2.3 Evaluation of the time-tracing feature

One of the most important features of DMetabench is its ability to continuously document the progress of a benchmark run. The following benchmarks will show how this data can be used to identify different types of disturbances that may occur during a run.

Figure 4.4 shows a very basic measurement involving the creation of empty files using NFS with four compute nodes and one process on every node that has been running for 60 seconds[‖] on a non-idle production file system.

In the top graph, 'Operations Completed', it can be observed how the number of files created increases almost linearly with time. The data resolution is 0.1 seconds and there are approximately 600 data points that directly correspond to recordings from the benchmark. Here the number of operations completed is summed for all four processes.

The bottom graph 'Operations/s' shows the performance at every single point in time during the benchmark. Performance is calculated as the difference between two time intervals multiplied by the number of time intervals per second (in this case, 10 using 0.1 second intervals). Case (a) in the figure shows an unobstructed run while case (b) shows a run in which one of the nodes was purposely made slower. A workload generator (`stress` [Str07]) started several dozens of CPU-intensive calculations so that less CPU time was available for the benchmark process on this node. The graph clearly shows that performance decreased from approximately 5,500 to 4,000 operations per second, from *t=16s* to *t=22s*. The total number of operations completed was still higher in spite of the simulated "problem", because the benchmarks were run on an active production file server, which reduced the accuracy.

Recognizing disturbances

A sudden change in the performance of a multi-process benchmark may be due to one of two reasons: either the whole file system behaves differently or one (or several) processes are slower/faster than others. For this test case, one of the four processes was slower than were the other three. The middle graph shows the coefficient of variation (COV) of the performance of the particular processes. For every point in time, the performance of each process was computed (see above) and standard deviation calculated before being finally normalized to the mean per-process performance. In this graph, the artificial slowdown can be easily identified, because the constant difference in performance increased the COV to a higher level.

Figure 4.5 shows a similar measurement, but in this example the NFS file server started the creation of multiple file system snapshots at *t=9s*. Again the COV of per-process performance also becomes higher, but in a much more random manner.

[‖] A typical MakeFiles benchmark.

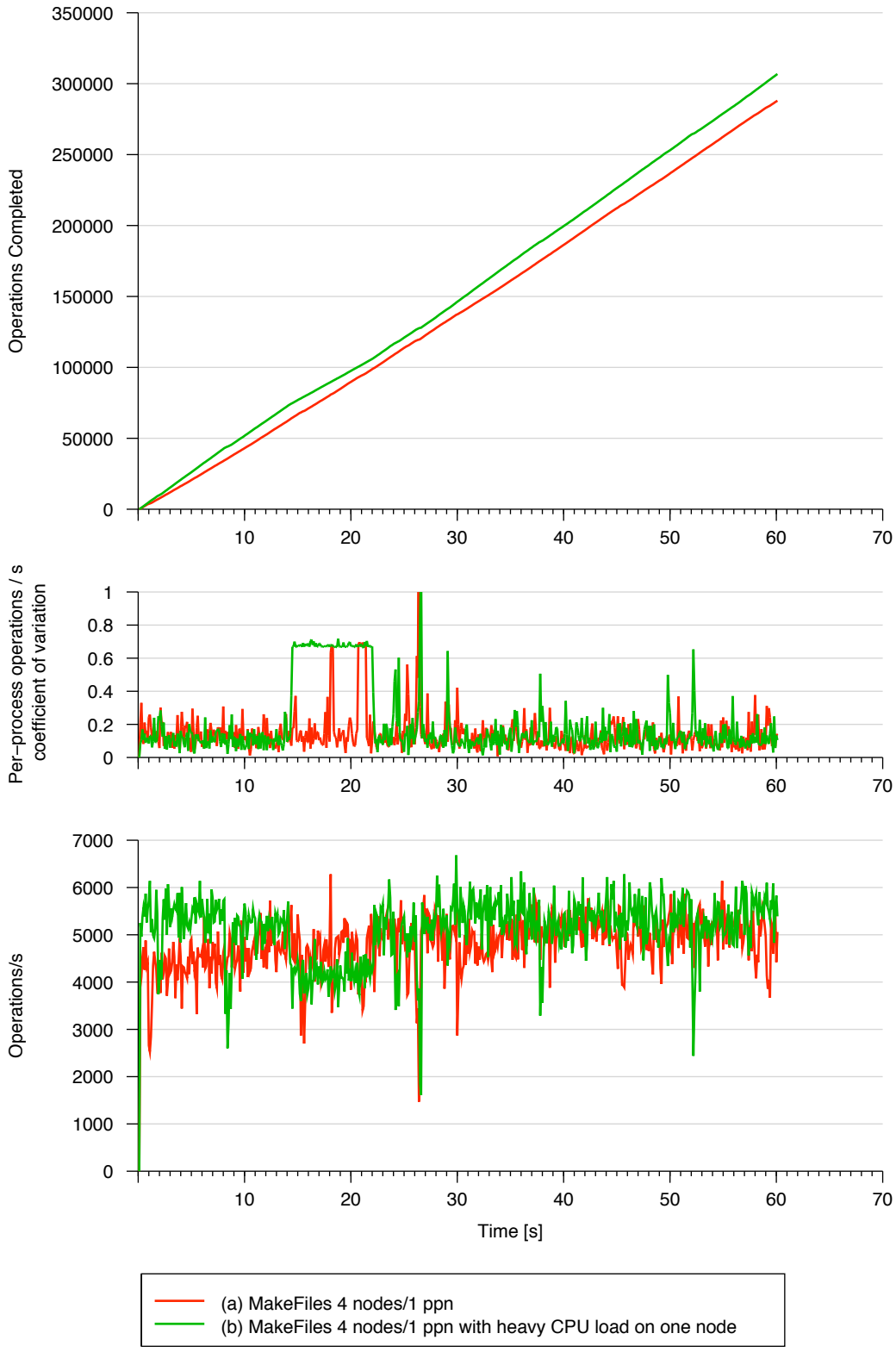

Figure 4.4: MakeFiles benchmark from four nodes to a Netapp NFS server using one process per node. (a) Shows an unobstructed run while (b) shows a CPU-intensive process was started on one of the nodes for 10 seconds

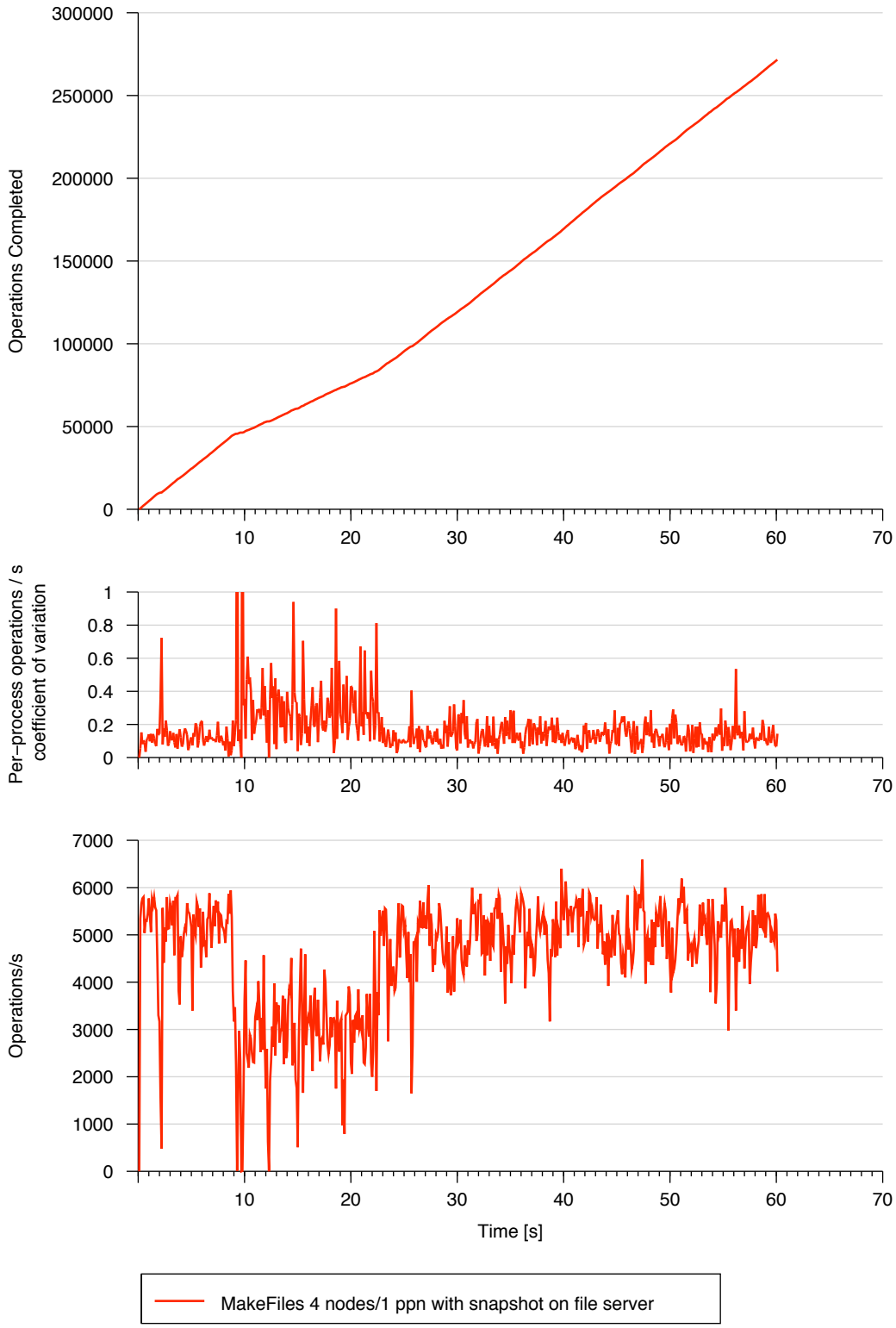

Figure 4.5: With a setup identical to that in Figure 4.4, the NFS server created multiple snapshots during the benchmark run. The COV of per-process performance changes in a very random manner.

Observing changes in performance

The measurement shown in Figure 4.6 is similar to the previous measurements described but uses 20 instead of four nodes. The additional load on the NFS file server brings it to saturation. The bottom performance graph reveals the internal consistency points of the WAFL file system [HLM02] that are triggered 10 seconds, at the latest, after the completion of the last consistency point. In spite of enormous changes in performance, the COV of per-process performance stays at the same level because all processes experience the same slowdown (see case (a) in the figure).

In case (b), a CPU-intensive workload was added on one node at *t=20s*. In this case, it was very difficult to observe any difference in total performance; in contrast to the four-node-measurement above, performance was now limited by the server and not by the number of processes. That means that even if one process is slower, the others will easily use the freed capacity. Therefore, although the difference in total performance cannot be observed, the middle graph showing the COV of per-process performance still clearly indicates the difference.

Based solely upon observation of the graphs, it is not possible to determine which process or which node behaved differently but, if doing so should become necessary, the original data recordings contain the required information.

The final example in Figure 4.7 again shows a global slowdown of the entire file system, this time caused by a large sequential file write to the NFS server that took away processing capacity from the metadata benchmark.

4.2.4 Summary

The system-level evaluation assessed internal performance of DMetabench compared to C a and demonstrated the time-logging functionality using artificial test cases. In contrast, the measurements in the following sections were performed to highlight the behavior of the file systems in question.

4.3 Comparison of NFS and Lustre in a cluster environment

Both NFS and Lustre are popular distributed file systems used for high performance computing. This section examines their characteristics regarding metadata performance.

4.3.1 Test setup

Figure 4.8 shows the setup for the measurements. One to 20 nodes from the LRZ Linux cluster were used as clients, with every node being a dual Quad-Core Xeon with 32 GB of RAM and a 1-GigE connection to the network. On the server side, the NFS server was a

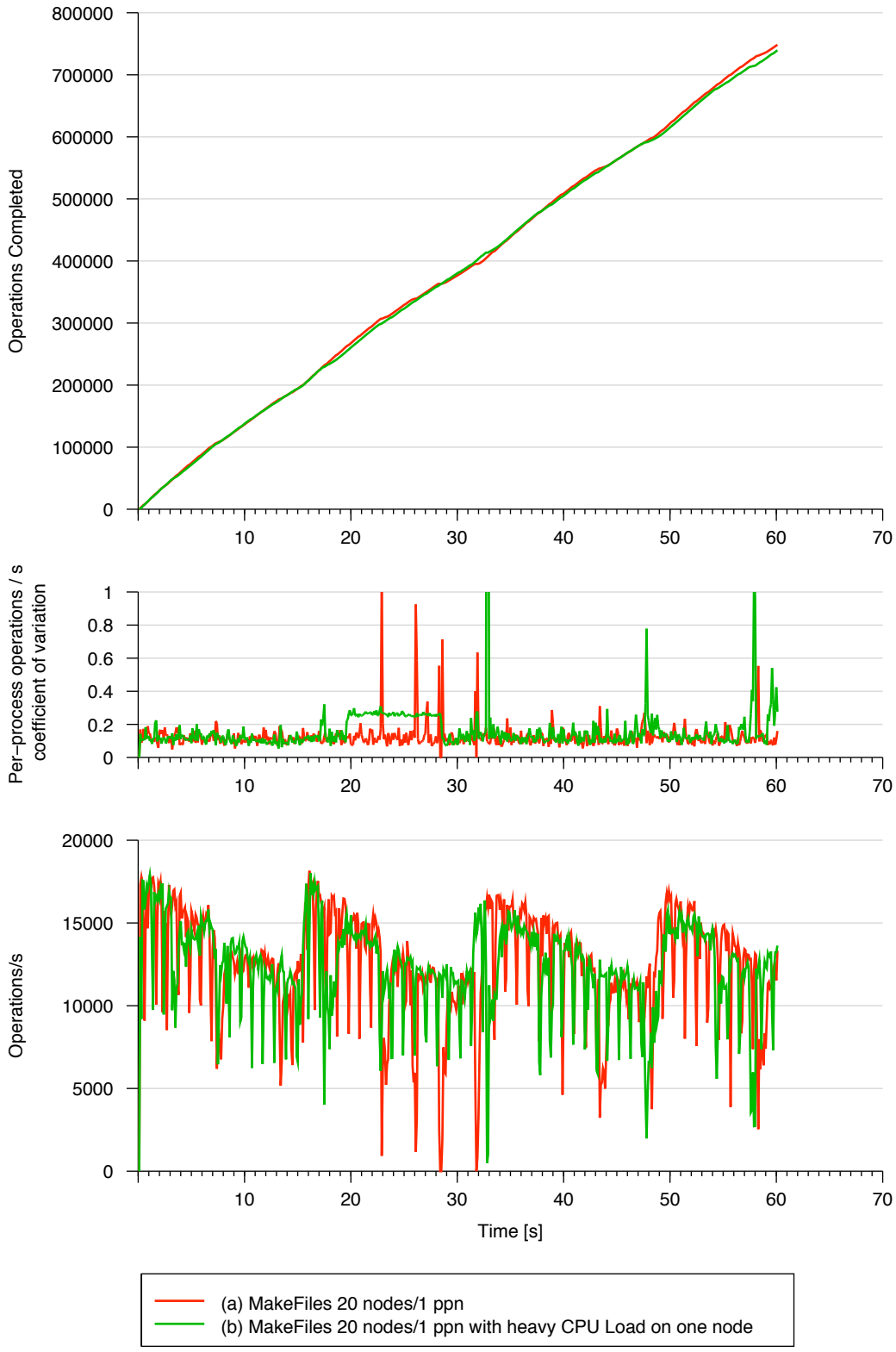

Figure 4.6: A setup is similar to that in Figure 4.4 but using 20 instead of four nodes. The consistency points of the NAS server can be observed (sawtooth form). In (b), one node was obstructed using the CPU hog process approximately 20 seconds after beginning.

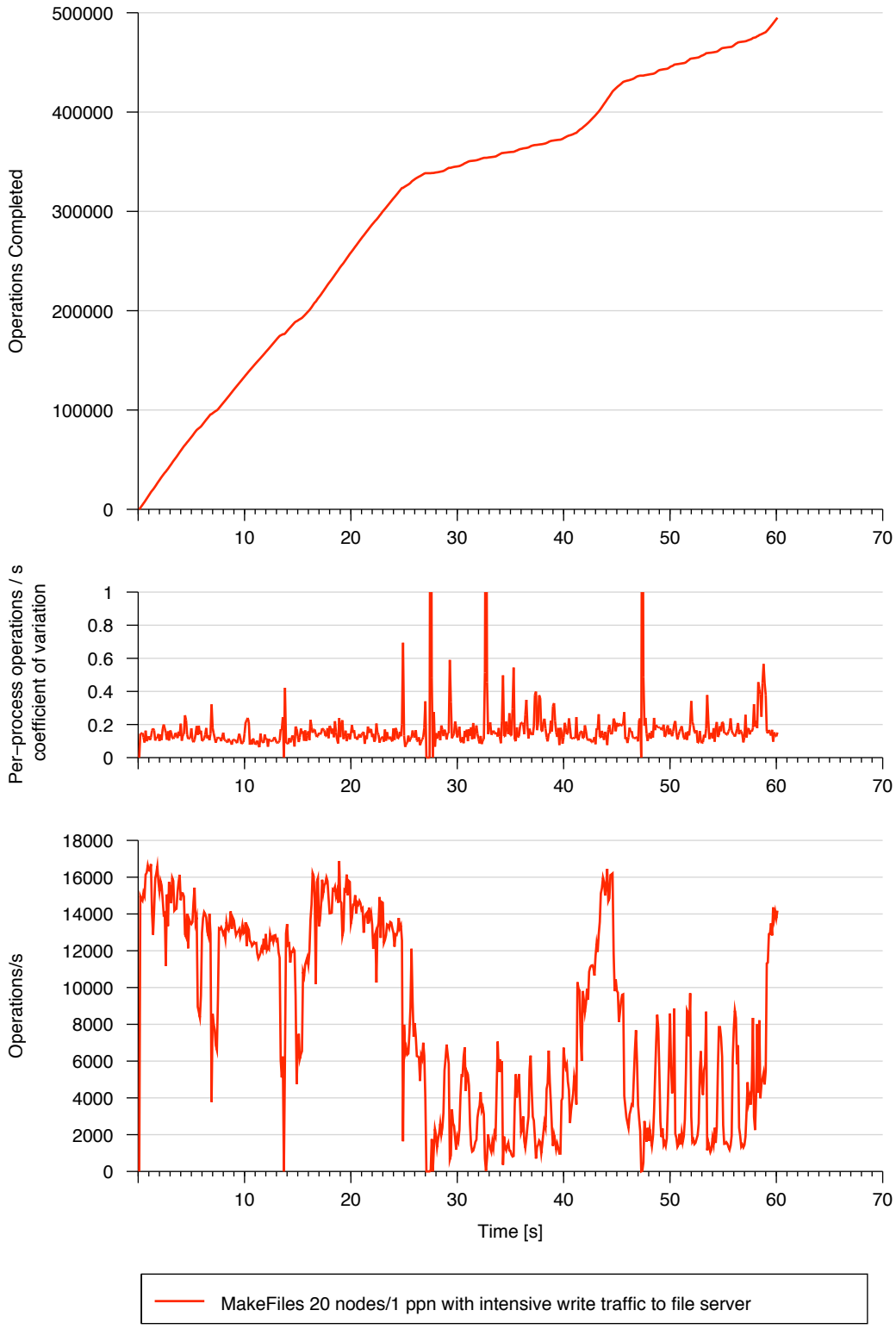

Figure 4.7: During this measurement, an external process writes a large file to the NAS server twice. While the MakeFiles throughput decreases, there is very little difference between the different nodes.

Netapp 3050 using the WAFL file system. The Lustre environment consisted of the metadata server (MDS) and the object storage servers (OSS).

It is very important to keep in mind that for NFS, the measurements represented the specific behavior of the Linux NFS-client and this particular type of NFS server.** Thus, it is not always possible to transfer the results to other operating systems or other server types. At present, Lustre is only available for Linux.

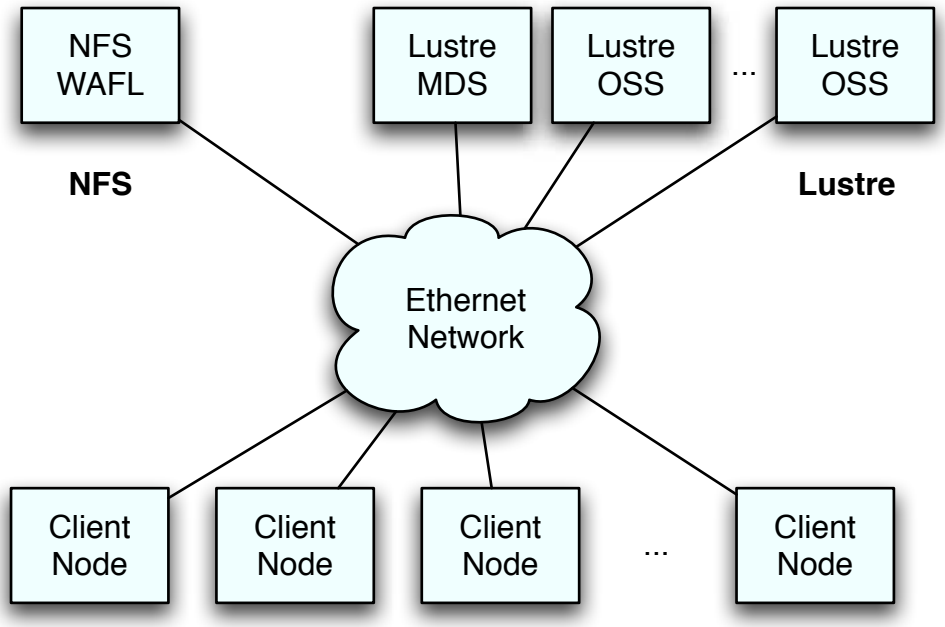

Figure 4.8: Test setup for NFS and Lustre benchmarks

All benchmarks were performed in a running production environment, which means that the remaining Linux cluster was also using both file systems. This is a very realistic situation for larger production systems.

DMetabench was invoked from a standard batch script that was submitted to the batch scheduling environment. When a direct comparison between Lustre and NFS was needed, DMetabench was called twice from the same batch job to ensure that exactly the same nodes were used.

**Netapp NFS servers implement the WAFL file system, which differs significantly from other file systems (e.g., those available in Linux.) with respect to internal metadata management [HLM02].

4.3.2 File creation

NFS

Figure 4.9 shows the scaling of file creation on the NFS file system. With an increasing number of processes, performance generally increases until it reaches a plateau when the NFS server is fully loaded (approximately 12,000-14,000 operations per second). In this benchmark, the server can be saturated with 10 to 12 processes. If the server is not fully loaded, every process brings an additional 1,000 file creates per second. This is true for both intra-node scaling (multiple processes on one node) and inter-node scaling (multiple nodes). For example, four processes on one node have approximately the same performance as do four processes on two nodes or four processes on four nodes.

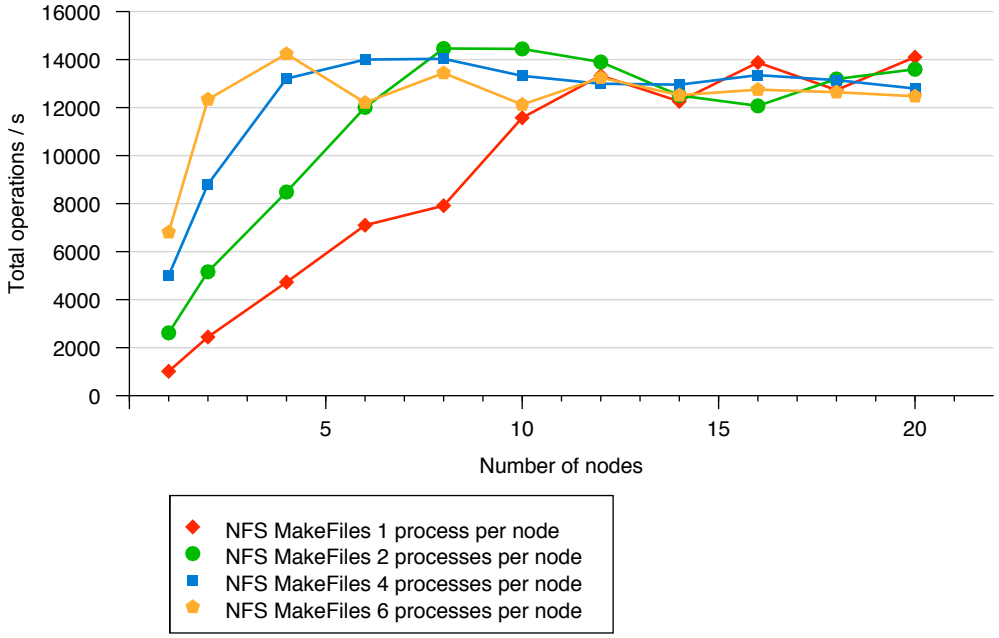

Figure 4.9: Creation of empty files with a varying number of nodes and processes and a separate directory for each process. Good scaling is observed until saturation of the NFS server (NetApp 3050). Comparable numbers for both intra- and inter-node scaling when using the same number of processes can be achieved.

In Figure 4.10, two aberrant data points from Figure 4.9 are shown in the timeline view. Additional disturbances were present during the measurement and lowered the average performance. For example, the data point for eight nodes and one process per node shows an average of 9,000 creates/s until the very end, when an other I/O-heavy process slowed down the server.[††] The benchmark could be run again to get a clean measurement but al-

[††]Keep in mind that the benchmarks were run on a production system.

ternatively the values could be corrected using data obtained graphically from the timeline view. The time-interval sampling feature of DMetabench preserves information normally lost in existing benchmarks, thus helping make better use of measurements that are not performed under perfect laboratory conditions.

Lustre

Lustre exhibits completely different scaling characteristics on the MakeFiles benchmark than does NFS (see Fig. 4.11). One difference is that its saturation level is much lower: 3,000 file creates per second versus approximately 12,000 for NFS. It is thus more difficult to observe scaling because the difference between one process and the maximum is smaller than that for NFS. On a single node, running more than two processes does not improve performance; in comparison multiple nodes are indeed faster.

The detailed timeline (see Fig. 4.12) shows very little performance variations when compared to NFS. One reason might be that for the normal usage pattern of Lustre in the LRZ environment, there is very little load on the metadata server compared to NFS, where data and metadata operations are intermixed. A very regular and easily recognizable pattern is a five second interval within which performance dramatically drops. It corresponds to the Ext3 journal commit interval on the Lustre metadata server. Every file created in the Lustre file system is stored in a local Ext3 filesystem on the MDS and the regular journal flush leads to a temporary drop in performance. Interestingly, it is also detectable when there is only one process and the MDS is not being fully utilized.

Scaling in NFS vs. Lustre

How can the differences in scaling between NFS and Lustre be explained? Creating a file in both file systems involves RPC calls between the client and the server. If there are multiple client nodes, their RPCs are issued simultaneously and the number of concurrent RPCs is only limited by the number of available nodes.

Inside a single client node, both filesystems implement a limit for the number of concurrent active RPC calls to a server. This can be modeled as a fixed number of RPC slots: If all slots are being used, additional RPCs will be queued (see Fig. 4.13). The limit can be tuned for NFS and Lustre using the Linux syscontrol parameters shown in Table 4.3. With Lustre, these parameters can be set for the MDS and the OSTs independently.

With NFS it can be argued that for intra-node scaling the number of processes used is lower than the default number of concurrent RPCs; thus all requests are issued in parallel, which allows linear scaling. With multiple nodes RPCs coming from different clients are independent anyway, so linear scaling is expected, as long as the server is able to handle the load.

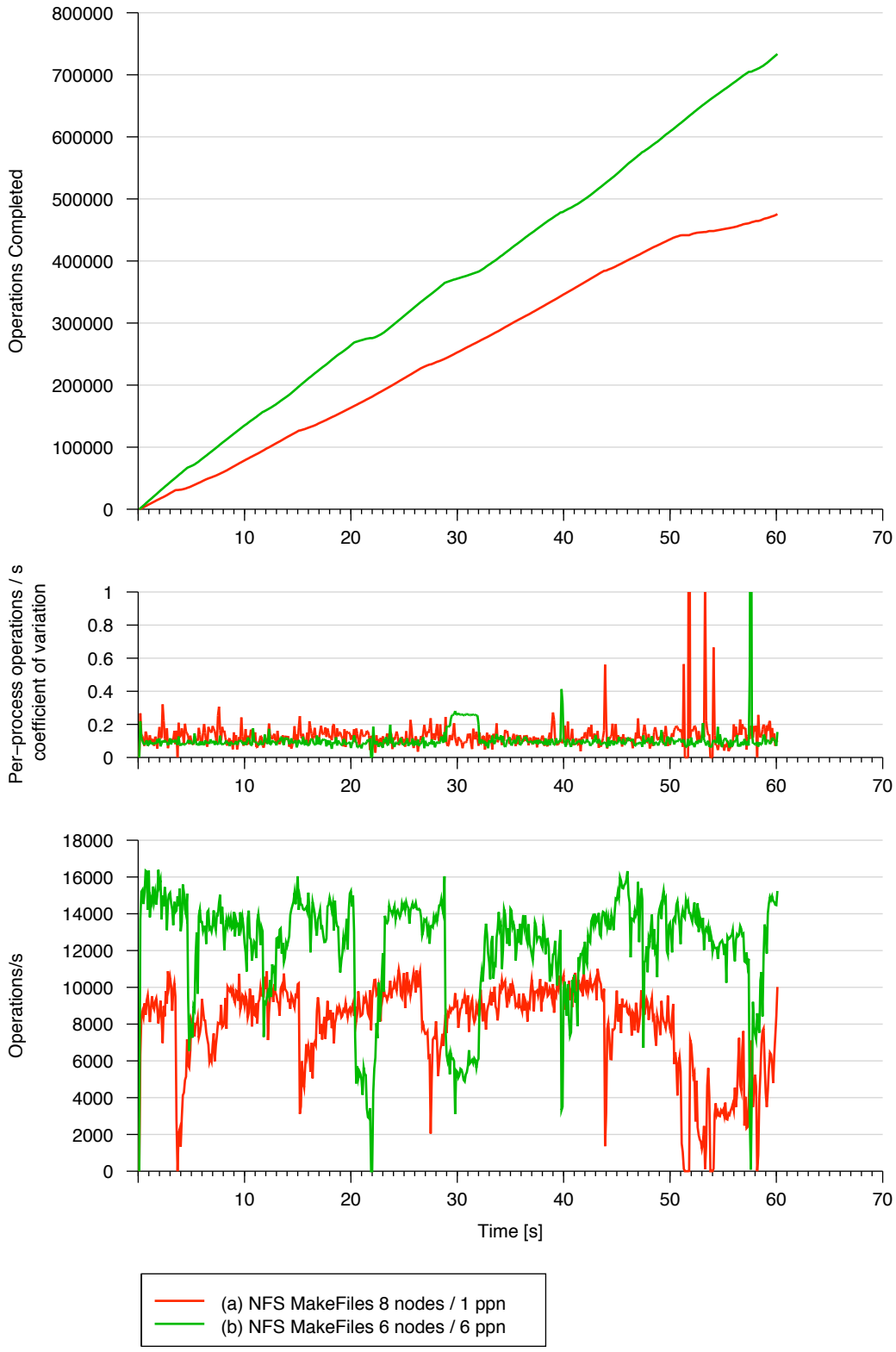

Figure 4.10: Detailed runtime data for two datapoints from Figure 4.9: (a) shows that total performance was significantly lower during the last 10s of the run. (b) shows that two areas (20-25s and 29-34s) has a lower throughput. The benchmark was run on an active production file system of an 800-node cluster.

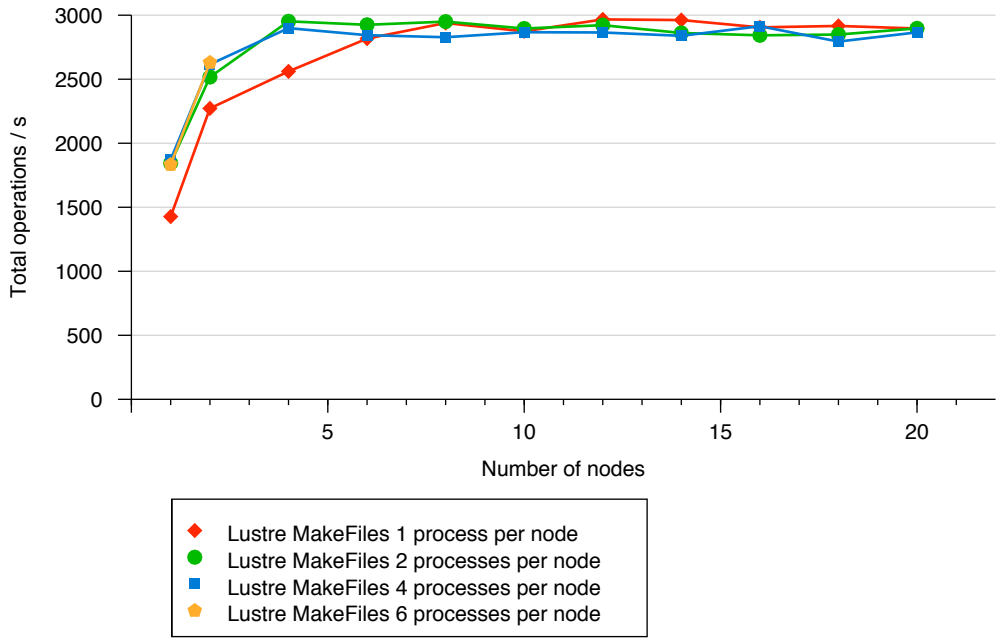

Figure 4.11: Scalability of file creation in Lustre

Table 4.3: Limits for concurrent RPC requests

Filesystem	Parameter	Default value	Max. value
Lustre	`max_rpcs_in_flight`	8	32
Linux NFS client	`tcp_table_slot_entries`	32	128

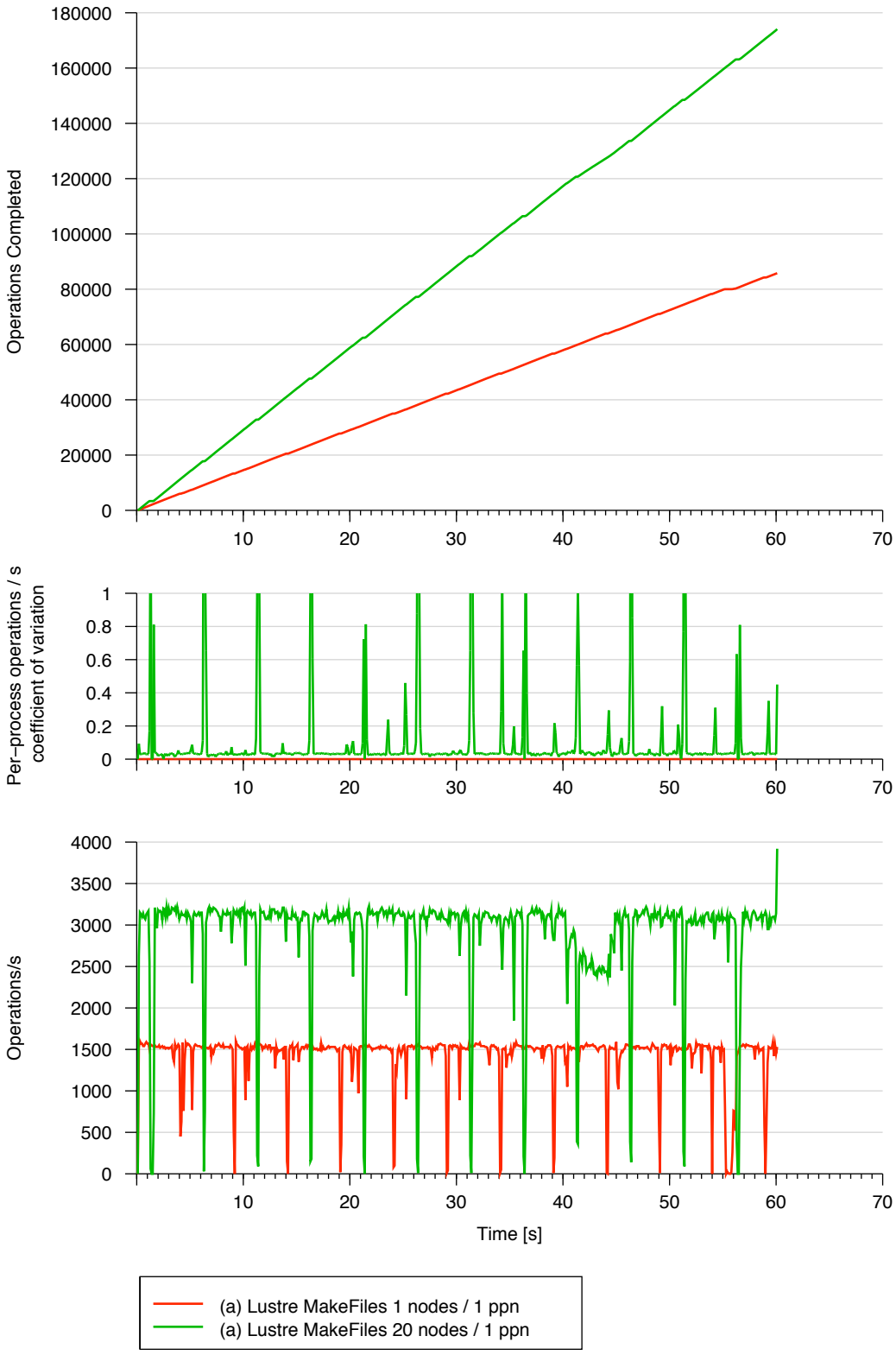

Figure 4.12: Detailed timeline view of two data points from Lustre shown in Figure 4.11. There are muchfewer performance variations than with NFS and the five second Ext3 journal commit interval on the metadata server is clearly visible.

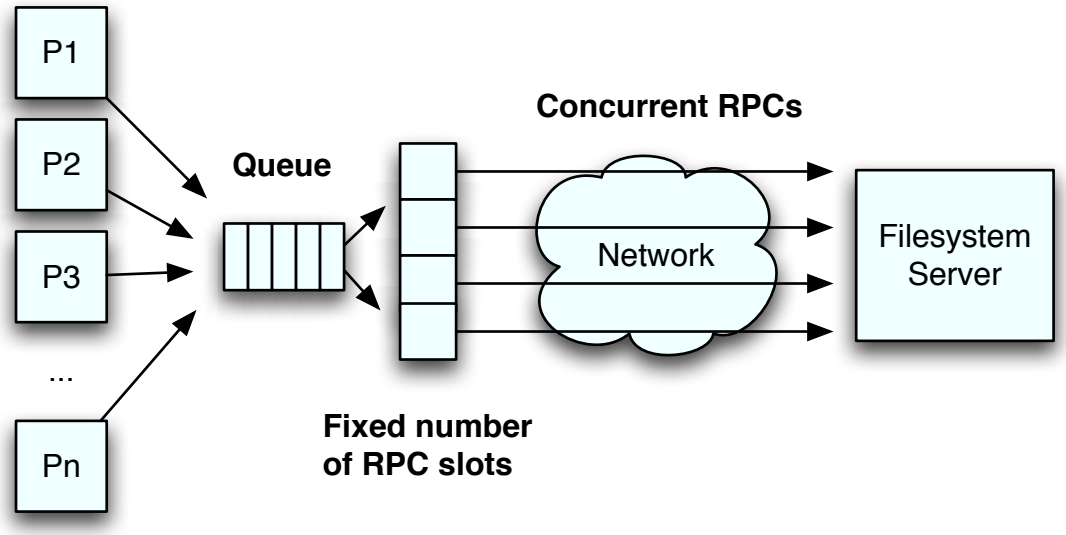

Figure 4.13: Model for limiting the number of concurrent RPCs by using request slots

The Lustre graph does not show real intra-node scaling despite the fact that the number of processes that create files is still well below the default concurrent RPC limit of eight. A discussion of this issue with Lustre developers revealed that the current 1.6 version of Lustre imposes a limit of *one modifying call* (e.g., `open()`, `create()`,... but also `close()`) to the metadata server for recovery reasons. Nonmodifying operations, such as `getattr()`, can operate in parallel up to the limit defined by `max_rpcs_in_flight`. This asymmetric behaviour explains why there is no visible intra-node scaling for the MakeFiles-Benchmark in Lustre. The design choice itself seems plausible because Lustre was designed for large clusters of smaller nodes. For currently available large SMP machines with several hundreds of cores in a single system image, such as SGI Altix, this limitation is quite significant. This issue and its implications on the WAN usage of a Lustre file system are discussed in more detail in section 4.6.

4.3.3 Sequential and parallel file creation in large directories

In tis context, large directories are defined as directories with more than several thousand directory entries. Section 2.4.2 presented several techniques that improve the performance of operations on large directories, like hash-indexes or tree structures, compared to the usual O(n) linear list structure used in the original UNIX file system. A common practice among system programmers is to avoid using very large directories because they slow down per-

formance. By using DMetabench the effect of having thousands of files in a directory can be investigated.[‡‡]

Sequential file creation

To find out how file creation performance varies with the number of directory entries, DMetabench was told to create a single directory with 400,000 files using a single process and Figure 4.14) shows the result timeline for NFS/WAFL and Lustre.

The graph shows that file creation on NFS/WAFL is initially faster than that on Lustre, but creation speed decreases linearly with the number of exiting files. Lustre, in contrast, does not exhibit noticeable performance variations and its creation performance remains constant. Here again it is imporant to remark that NFS behaviour is specific for the WAFL file system used on the particular NFS server. WAFL uses traditional linear directories augmented with a hash-based index. The hash selector, which has a size of one byte, does not fundamentally change the O(n) append complexity but rather helps improve performance for normal-sized directories. In Lustre, the metadata server uses a local Ext3 file system with the `dir_index` option enabled. For directories with between 70,000 and 16 million directory entries, which is the maximum allowed, the implemented H-Tree structure has exactly two levels of indirection in the tree and thus an O(1) append complexity [Phi01].

In practical applications, directories of this size are not very common, one of the reasons being that it is almost impossible for one person to inspect the content of a directory with thousands of entries, even if the system itself performs well. One application for larger directories is e-mail servers using the *maildir* format, for which there are several folders (=directories) with many e-mails (=files) for every user. Here a directory with several thousand files/e-mails is indeed possible, but in general these files are created one at a time and then accessed by a single machine process that corresponds to the owner of the mailbox. Other similar applications include automatic imaging equipment (e.g., microscopes).

Parallel file creation

Another question in the context of large directories is the manner in which multiple processes influence each other when writing to the same directory. To determine the answer, the MakeOnedirFiles benchmark was again used, but this time with a 50,000 file directory and multiple processes on multiple nodes (see Fig. 4.15 and Fig. 4.16).

In a previous measurement (see Fig. 4.9 for NFS and Fig. 4.11 for Lustre) a similar file creation was performed, but every process accessed its own directory. In comparison,

[‡‡]The number of files and the number of subdirectories allowed in a directory may be quite different, depending upon the file system. For example, in Linux Ext2, maximum of 32,766 subdirectories may be created because every subdirectory needs a hardlink for '..' to the parent directory, and the internal metadata allows a maximum of 32,768 hardlinks.

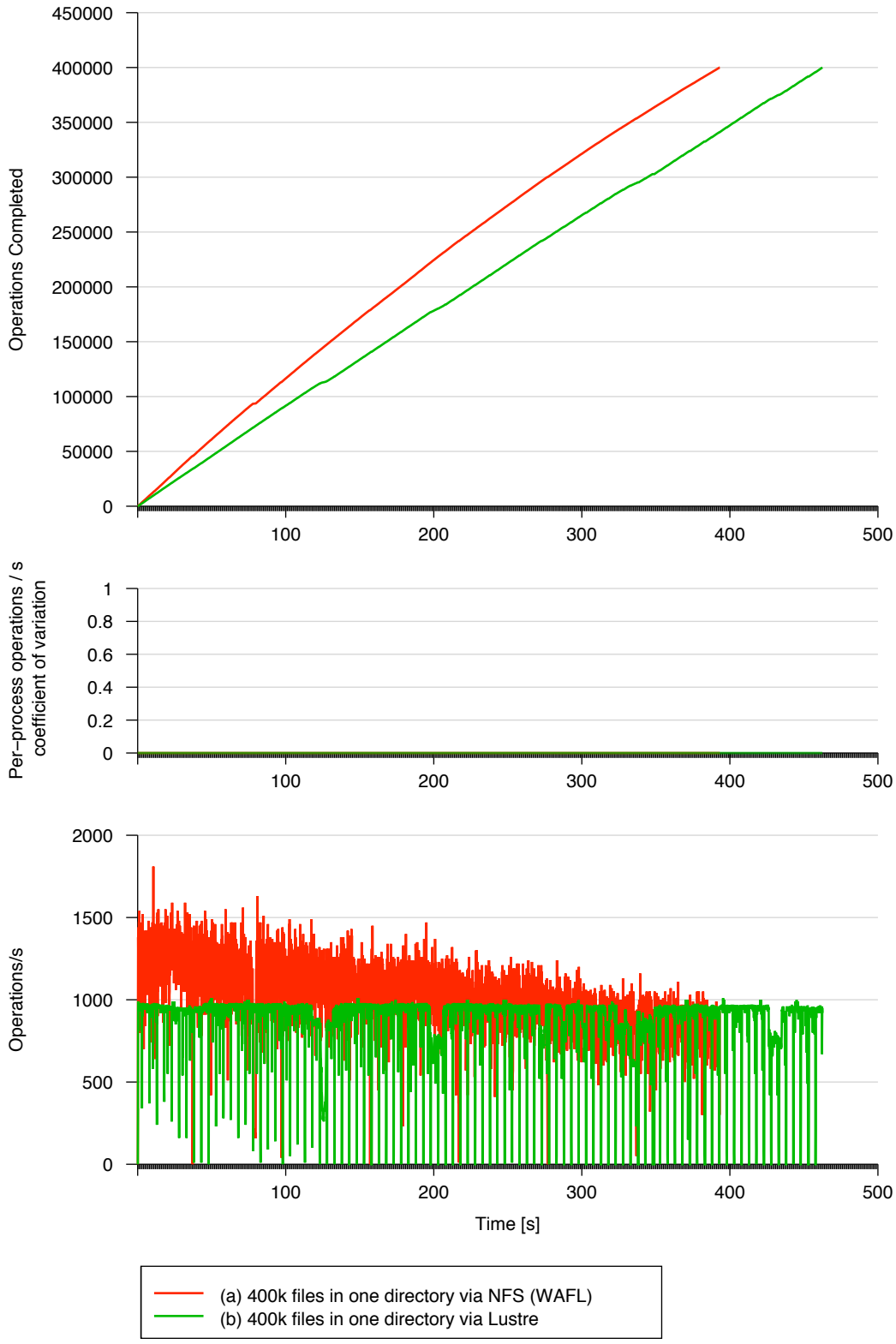

Figure 4.14: Creation of a large directory with 400,000 empty files in NFS/WAFL and Lustre. Create performance decreases with the number of files in WAFL but remains at the same level in Lustre. WAFL uses a simple index structure for directories and Lustre relies on ext3 `h_tree` B-tree-like structures on the metadata server.

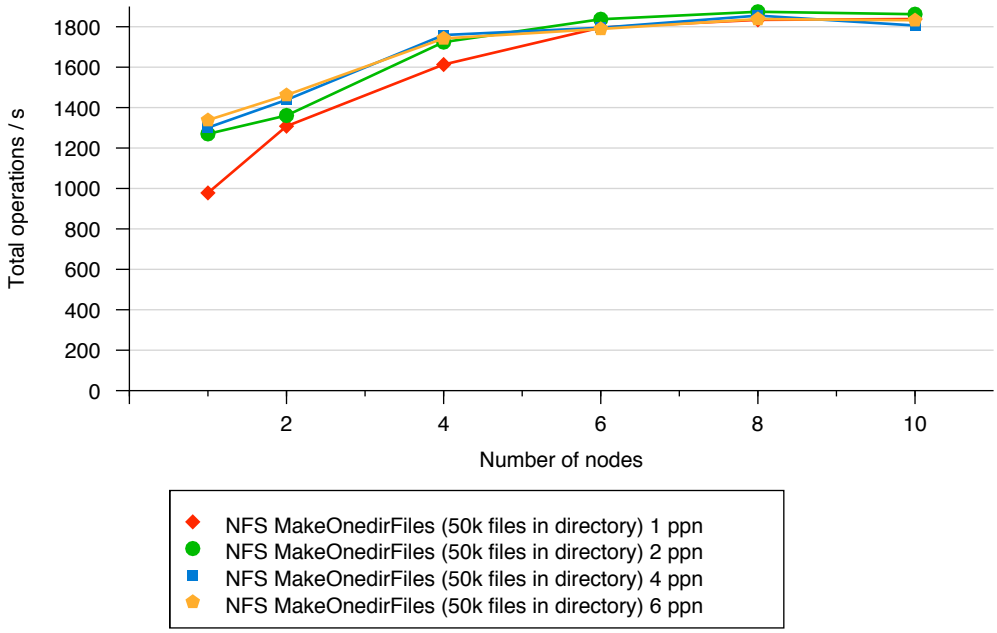

Figure 4.15: File creation with multiple processes in a single directory in NFS/WAFL

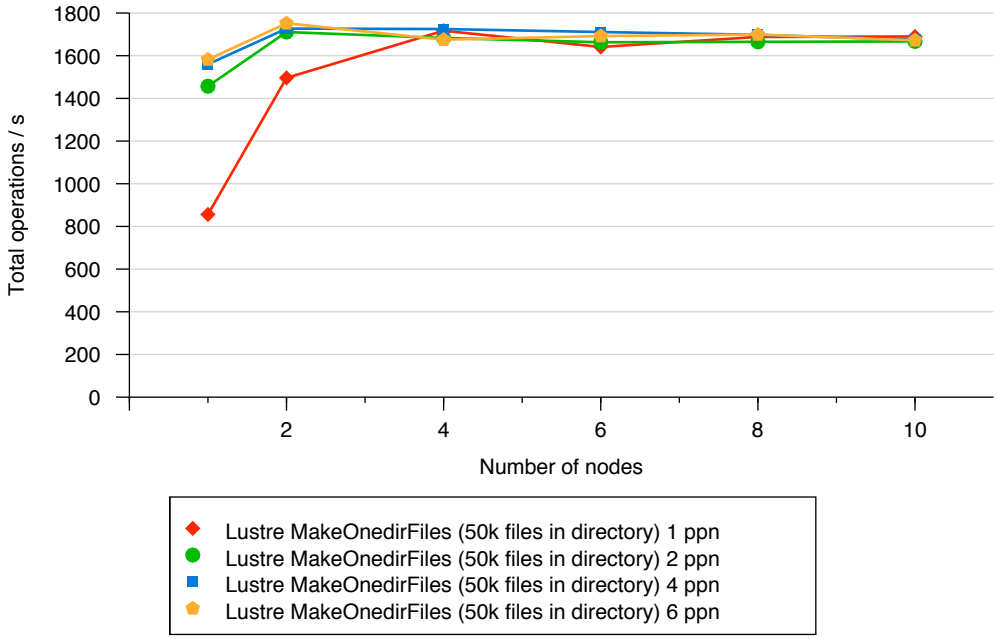

Figure 4.16: File creation with multiple processes in a single directory in Lustre

performance for a single directory is much lower both in NFS and Lustre, and beyond four nodes there is practically no scaling at all.

The root cause of this behavior is locking: On the Linux client side any modification in a directory locks the entire directory in the VFS for the duration of the operation. For distributed file systems using RPCs, this means that access if effectively serialized. The same problem arises on the server that must provide exclusive access to the common directory for all RPCs. Cao et al. presented a solution prototyped during design work for Lustre that involves fine-granular locking in VFS on a per-filename basis instead of on the entire directory [CTP+05]. Additionally, the filesystem itself (Ext3 in case of the Lustre MDS) locks its directory data structures with the granularity of a tree leaf.

Unfortunately, this *pdirops* patch has not found its way into the official Linux kernel as of 2008 since its development in 2005. Commercially available NFS servers are also unable to process operations on a single directory in parallel. Therefore, it remains advisable to avoid large directories when simultaneous file creation is involved.

This type of access can also occur in mailserver applications in the *spool directory*, which contains e-mails received by a mail server that have not yet been processed. In contrast to the example above, access to a spool directory is highly parallel because many e-mails can be delivered at the same time, even if there is only one server accessing the directory. Additionally, delays in the processing can cause the spool directory to quickly grow up to hundreds of thousands of messages. Practical experience with production mail servers at the LRZ in 2004 have showed that is it not feasible to place a single, common spool directory on an NFS share: Exactly as occurred in the benchmark, locking slows down operations to a point where the mail server is unable to keep up with incoming messages.

Figure 4.17 shows a detailed timeline of the measurements from Figures 4.15 and 4.16. Because there are no recognizable patterns of WAFL consistency points in graph (a), it can be concluded that the performance achieved is not limited by the saturation of the NFS server, as it is in the example in Figure 4.10 (b). When using Lustre and the underlying Ext3 file system it is, unfortunately, not possible to reason in the same manner: Flushing the Ext3 joural seems to briefly stall all operations, independent of the load on the server. In the case of WAFL processing, a consistency point uses up resources but does not interrupt processing.[§§]

Although issues with large numbers of files in a single directory are well-known to most UNIX administrators, most comparisons have been anecdotal. However, a very informative paper written by Jeff Turner discussed file creation latency in a Solaris 10 environment [Tur06]. The author used simple bash scripts to generate either one large or multiple smaller

[§§]Technically, NFS operations are logged into non-volatile RAM (NVRAM). The available NVRAM is divided into two sections so that one half can be used for logging operations while the other half is undergoing a consistency point [DMJB98].

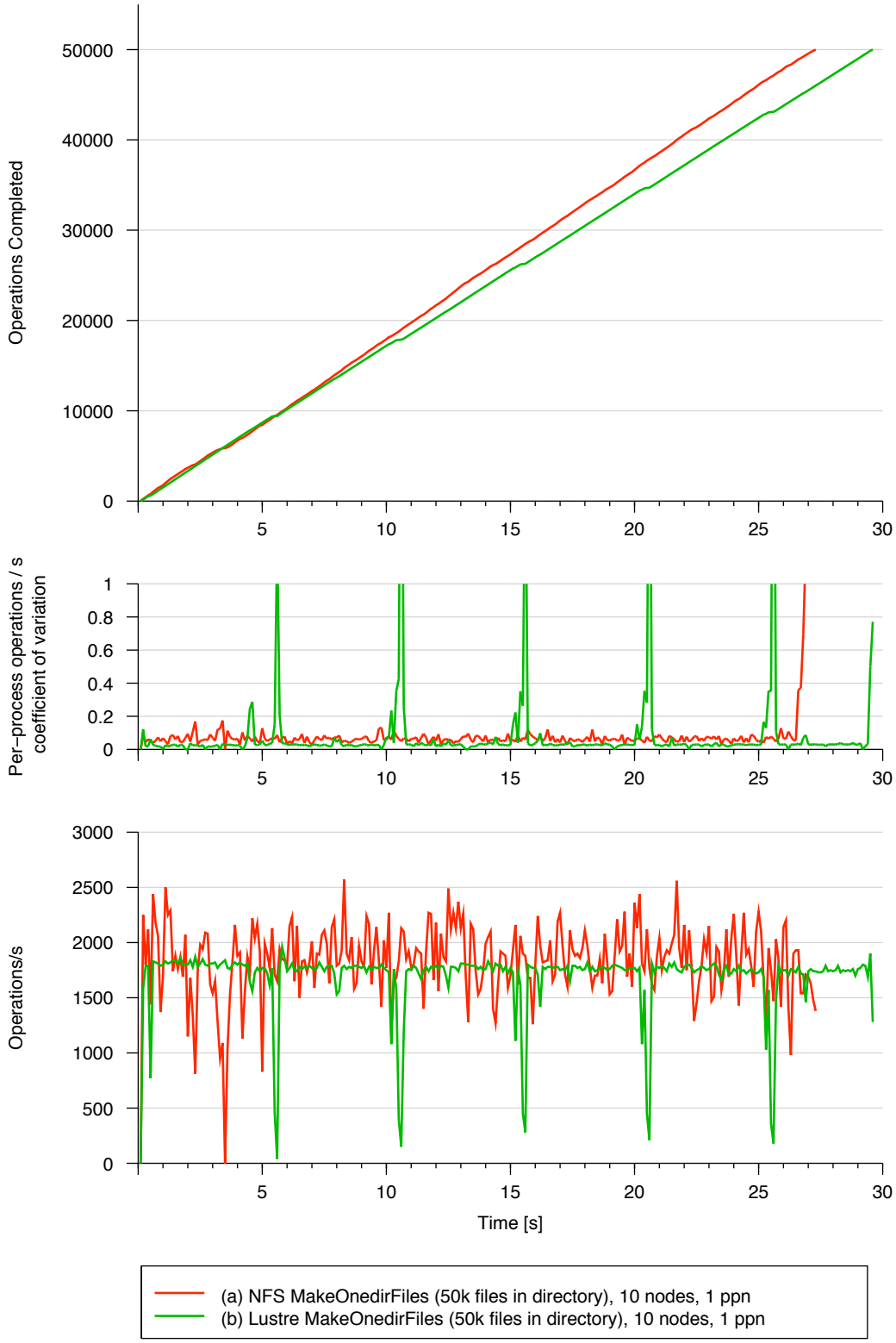

Figure 4.17: Detailed timeline for two data points from Figures 4.15 and 4.16. Regular CP paterns are absent in the NFS/WAFL graph while the regular 5s journal commit in ext3 on the Lustre MDS remains visible.

directories and the dtrace-Toolkit available in Solaris 10 to measure the duration of every single operation.

4.3.4 Observing internal allocation processes

The WAFL file system used on LRZ NFS servers is able to acommodate up to 64 bytes of data into the inode, avoiding the need to allocate a complete 4kB data block. This means that only files larger than 64 bytes require the allocation process that provides the first data block. Using DMetabench, it is possible to show this difference. Two special micro-benchmarks based on MakeFiles (*MakeFiles64byte* and *MakeFiles65byte*) were created that write 64 and 65 bytes, respectively, into each file.

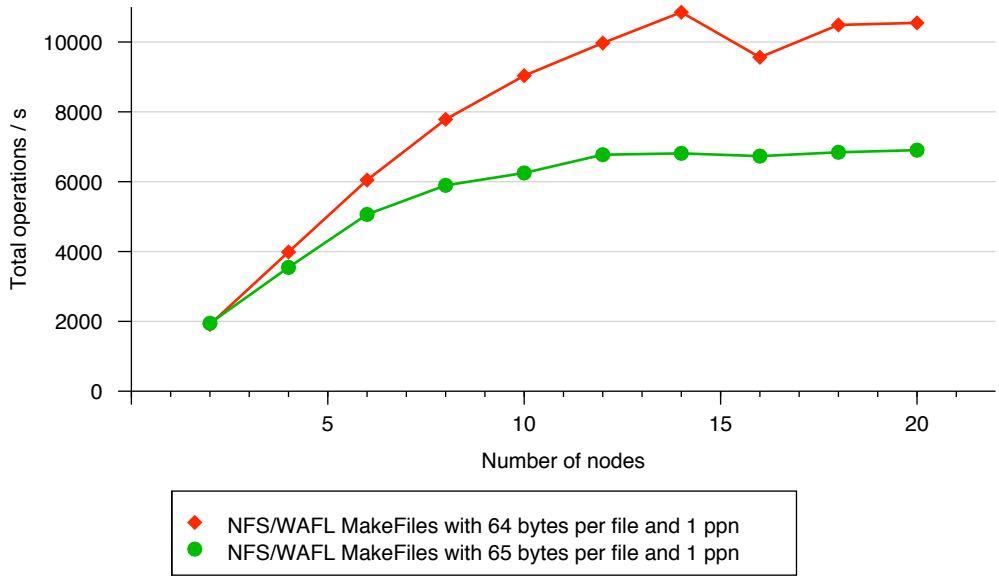

Figure 4.18: Effects of internal WAFL metadata allocation

In Figure 4.18, two processes on two nodes initially perform at the same speed, but with more processes, the variant that writes 64 bytes becomes faster. One 4 kB block per file at a rate of almost 7,000 files per second is equivalent to an additional write load of 28,000 kB/s. In the timeline graph with 20 processes, the regular drops in performance from consistency points have an interval of less than 10 seconds because the NVRAM of the NAS server becomes full in less than 10 seconds and earlier consistency points are induced (see Fig. 4.19).

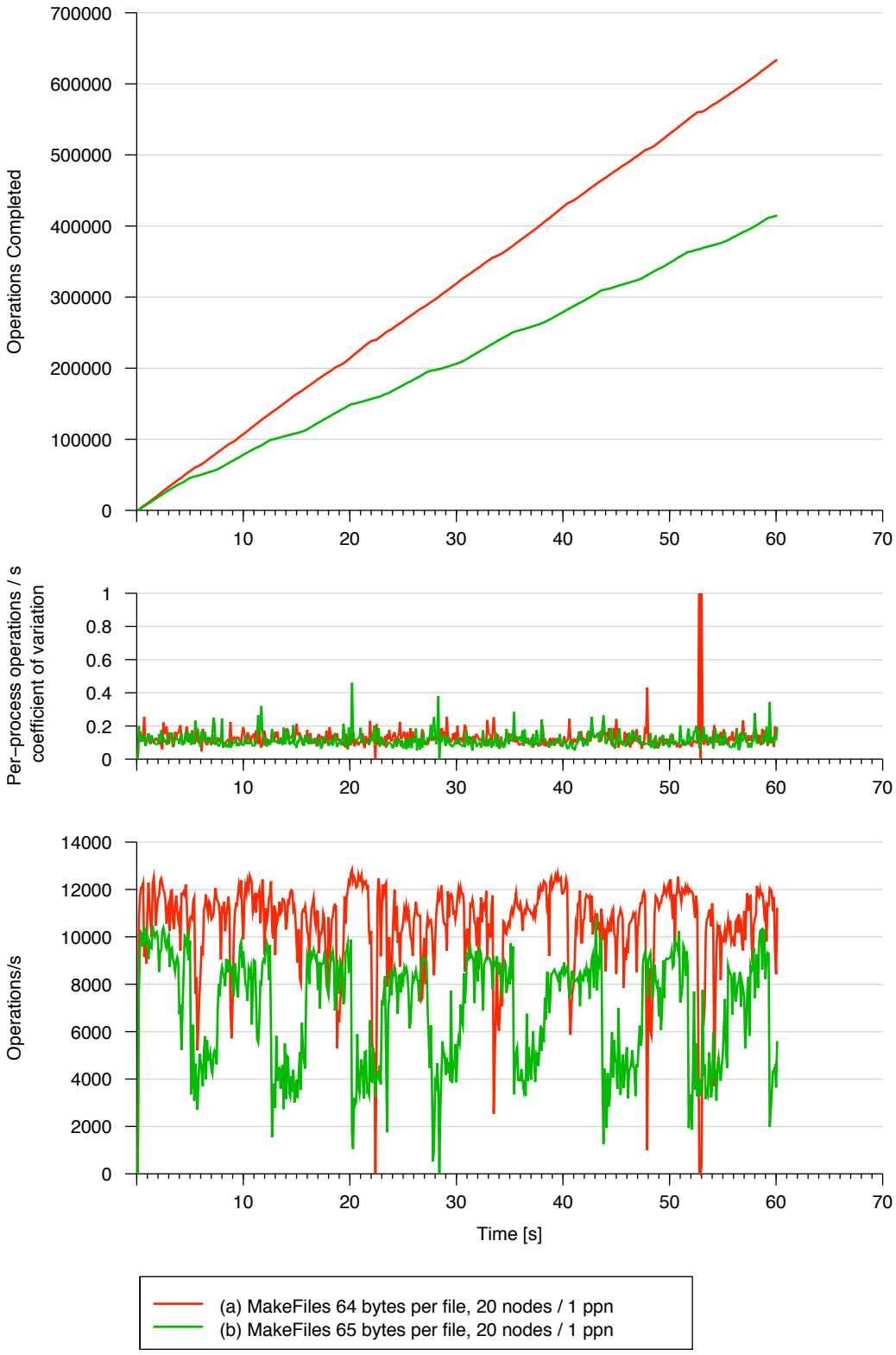

Figure 4.19: Detailed results of the WAFL metadata allocation benchmark

4.4 Priority scheduling and metadata performance

This section describes an experiment that, although regarded a failure, is nevertheless interesting and therefore described in the hopes of motivating further research.

NetApp NFS file servers utilized at the LRZ allow for the differentiation of I/O priority between different volumes using a priority scheduler called "FlexShare". In this scheduler, every server volume can be marked with a priority level (Very High, High, Medium, Low and Very Low) and separate processing queues are used for I/O requests from different priority levels. The documentation explains that 'once a WAFL operation is dispatched to execute, FlexShare' work with the WAFL operation is complete. If there is a WAFL operation that has been dispatched or is already in progress, FlexShare will not interrupt that WAFL operation even if higher priority WAFL operations arrive in the system. FlexShare only controls the order in which WAFL operations are dispatched to be processed, but once they are dispatched they are out of the control of FlexShare' [Bha06]. Example in the documentation focus on *data* and do not mention *metadata* operations.

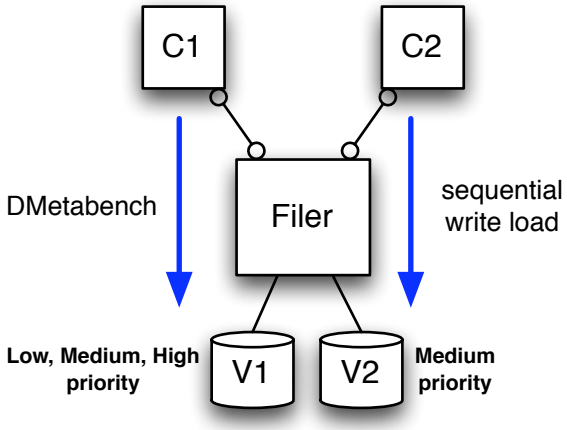

Figure 4.20: Setup for test of priority scheduling

The objective of the test of priority scheduling was to determine its impact on scheduling metadata operations and whether different settings lead to changes in performance. The test setup consisted of the NAS server configured with two volumes and two client nodes (see Fig. 4.20). As the priority scheduler only influences the system behaviour under high load when there are resource constraints, a sequential I/O load was applied to one of the volumes, which almost saturated the NAS server. The DMetabench *MakeFiles* benchmark was run on the other volume.

The volumes where sequential data was written and DMetabench operated were initially both set to *medium* priority. Then, the DMetabench volume priority was changed to *very high*

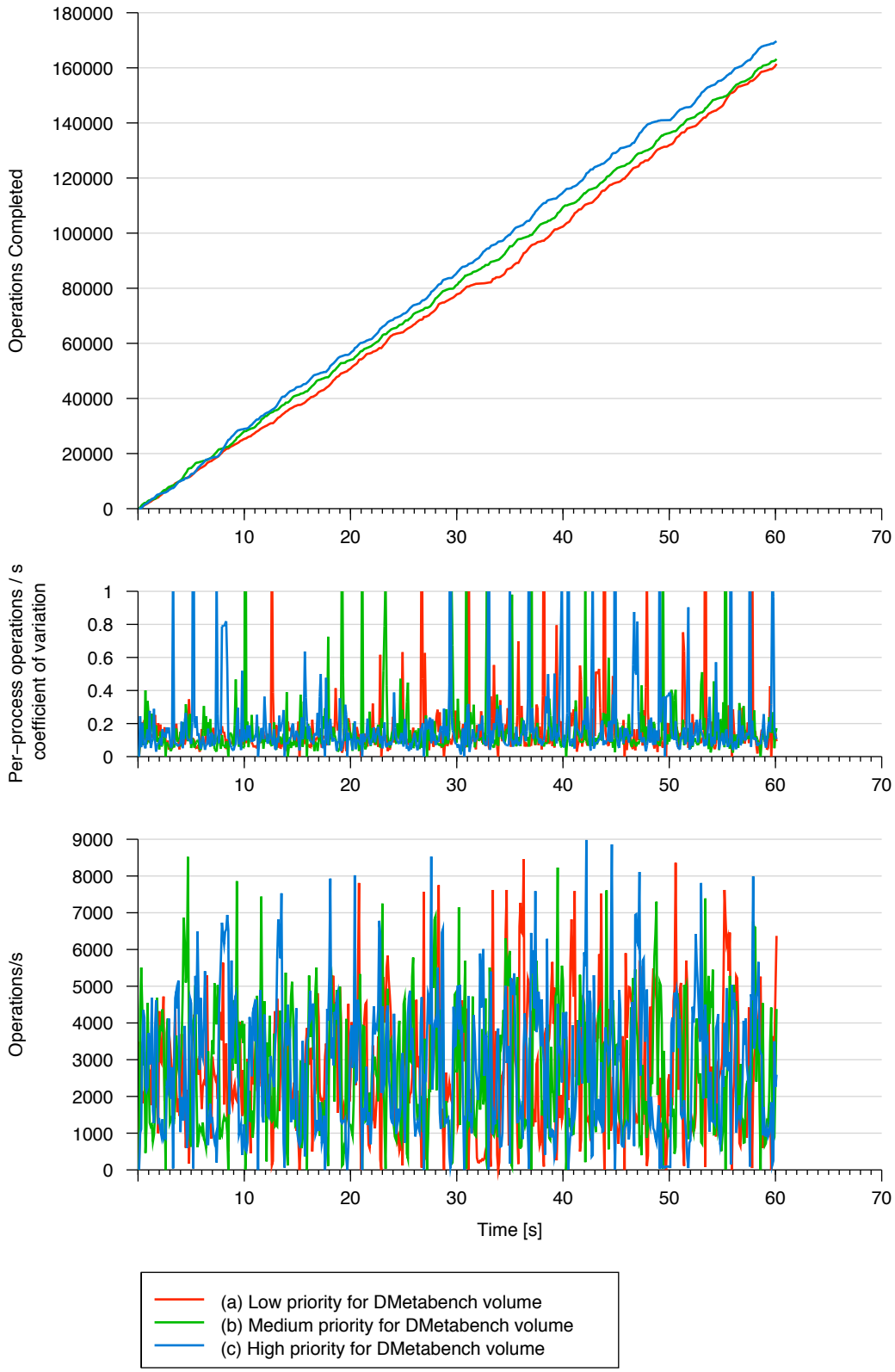

Figure 4.21: *MakeFiles* from a single node to a NFS server which is busy with an (external) sequential write load writing to a volume with *Medium* priority. The impact of changing the priority of the DMetabench volume is negligible.

and*very low*, respectively, for two additional runs. The results of DMetabench, using 6 client processes, are shown in Figure 4.21.

One expectation was that changing the priorities would have a significant impact on file creation performance. However, this was found to be untrue: All three settings offer an almost identical performance that is, by the way, much lower than the performance that can be achieved when the NAS server has no additional load (not pictured). It seems that the current implementation of FlexShare does not influence metadata-intensive workloads effectively.

4.5 Intra-node scalability on SMP systems

The increasing availability of SMP systems with multi-core processors has expanded the number of users for parallelized applications. While MPI-based parallelization is almost exclusively found in scientific applications, SMP-type servers are commonly used for business applications, such as messaging services or web servers. It is therefore useful to identify how different distributed file systems perform in comparison to each other and in comparison to a local file system.

4.5.1 Test setup

The client machine used to perform the following benchmarks is a Linux-based x86_64 server with eight Dual-Core CPUs (16 cores total), 128 GB of RAM and connections to multiple file systems: the CXFS and Ontap GX filesystems of the HLRB II described in section 4.1.3, the Lustre and NFS filesystems from the Linux cluster described in section 4.1.2) and finally a local high-performance storage system using XFS.

A partition of the HLRB2 with 512 cores has been used as an example of a large SMP system.

4.5.2 Small SMP system measurements

On the smaller 16-core SMP system three different distributed file systems were compared to a local storage system with XFS (see Fig. 4.22). In a test with MakeFiles, file creation performance of XFS was not dependent upon the number of parallel processes. Particularly, a single process performs as fast as multiple processes together.

The behavior of the three distributed file systems is irregular: NFS scales up to 6 to 8 processes, reaches a plateau with 10 processes and then performance slowly deteriorates. Lustre and CXFS do not scale at all.

With more than four processes NFS is faster than the local XFS file system running on a dedicated mid-range RAID storage array. From the perspective of the client node, creating

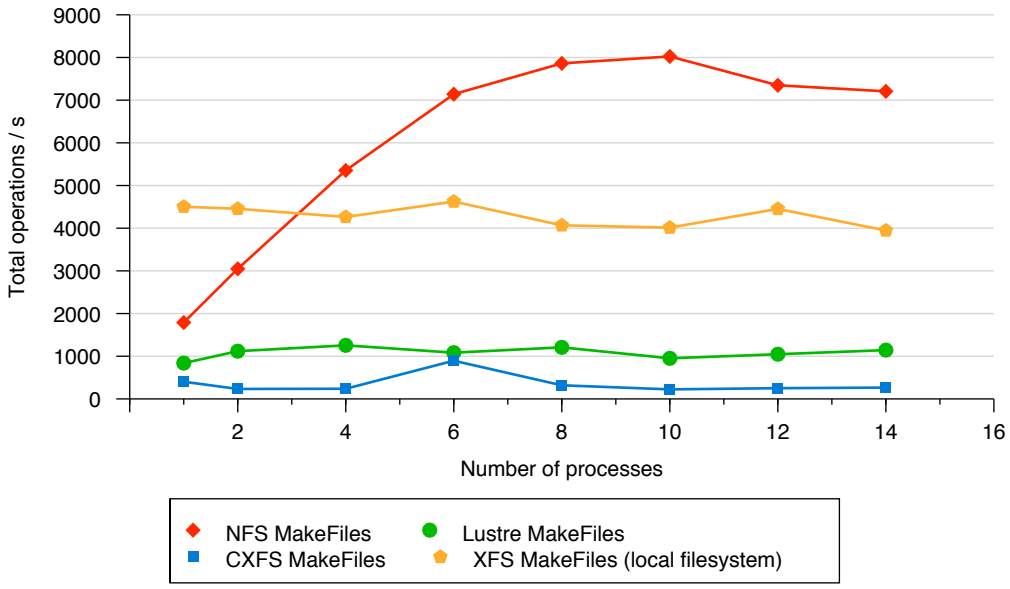

Figure 4.22: MakeFiles performance on a 16 core Opteron SMP system using NFS, CXFS, Lustre and a local XFS file system

an empty file is a very simple and small operation over NFS. The local file system has to perform much more and do disk I/O itself. For applications with some degree of parallelism in the access patterns, NFS should be taken into consideration for handling metadata-intensive workloads. At LRZ there are multiple e-mail servers which regularly handle up to 4,000 simultaneous sessions with 70,000 user accounts using NFS file systems to store the messages.

4.5.3 Large SMP system file creation performance of CXFS and NFS

Figure 4.23 compares the file creation speed between NFS and CXFS on a single partition of the HLRB2.

For CXFS, performance increases only slightly – from 1 to 500 processes – with an average of approximately 500 file creations per second. NFS characteristics are different: the maximum is positioned below 32 processes, and after 128 processes there is a significant drop in performance. While NFS performance is an order of magnitude better than that of CXFS for lower process counts, at 500 processes the difference becomes quite small.

With NFS, the sudden drop in performance for more than 128 processes corresponds to the maximum number of outstanding RPC calls, also set to 128. It seems that if there are more processes than slots, performance slows down because of blocking.

It can be observed that approximately one file per process and per second can be created for the maximum number of processes. This is especially noteworthy for CXFS: while data

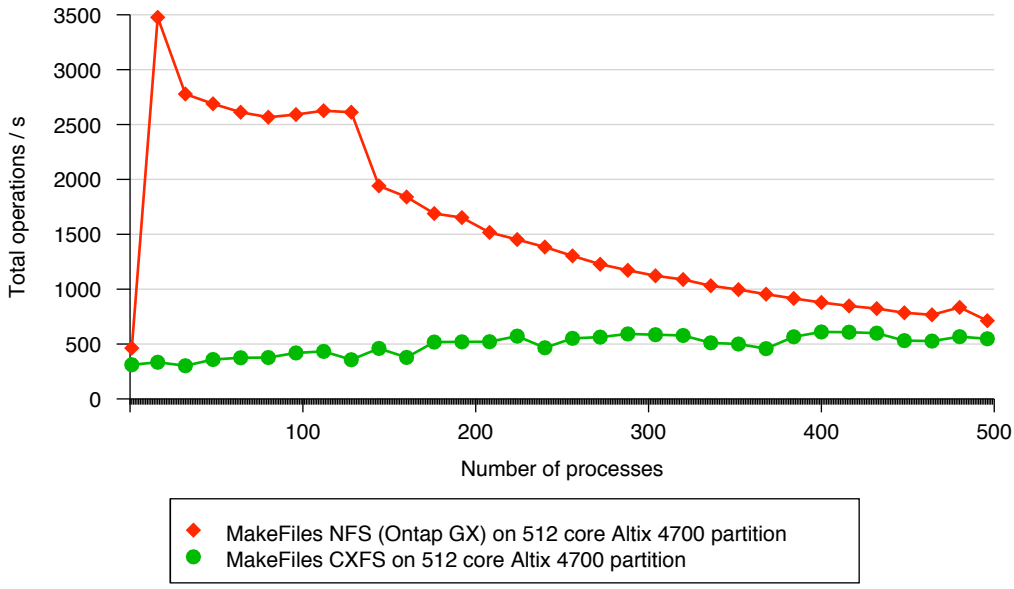

Figure 4.23: MakeFiles performance on a single 512-core partition of the HLRB2. For Ontap GX, only a single volume in the namespace is used. The measurements include 1 to 498 processes with data for every additional 16 processes.

throughput can be increased up to 3 GB/s from a single partition, at the same time only 500 files/s can be created. This clearly shows why CXFS in the existing configuration is better for large files and why NFS-based storage delivers a better user experience when performing interactive work (e.g., compiling software). The differences are consistent with the operational model at LRZ presented in section 4.1.

4.6 Influence of network latency on metadata performance

This section investigates how network latency changes metadata performance. While network bandwidth has been improving over the last years, there remain fundamental lower limits to latency in computer networks, such as the speed of signal propagation. Two common assumptions in communication networking are a latency of 1 ms for every 200 km of fibre cable and the addition of more latency by all all active network components.

Many trans-national, European and global grid projects in HPC rely on the ability to exchange data between sites many hundreds of kilometers apart. Table 4.4 shows examples of average round-trip ping times from the LRZ to chosen sites.

Table 4.4: Examples of round-trip latency from the LRZ (April 2008)

Site	Latency
University of Erlangen	5 ms
University of Heidelberg	6.5 ms
Research Centre Jülich	12 ms
ECMWF, Reading (UK)	28 ms
Barcelona Supercomputing Center (ES)	54 ms
Carnegie Mellon University (USA)	108 ms
Argonne National Labs (USA)	120 ms
University of Melbourne (AUS)	290 ms

Simulation of latency

To introduce a configurable latency in the test environment (see Fig. 4.24) a hardware simulator was used. The simulator consists of a dedicated node with two network interfaces running a FreeBSD kernel and the *dummynet* software module.

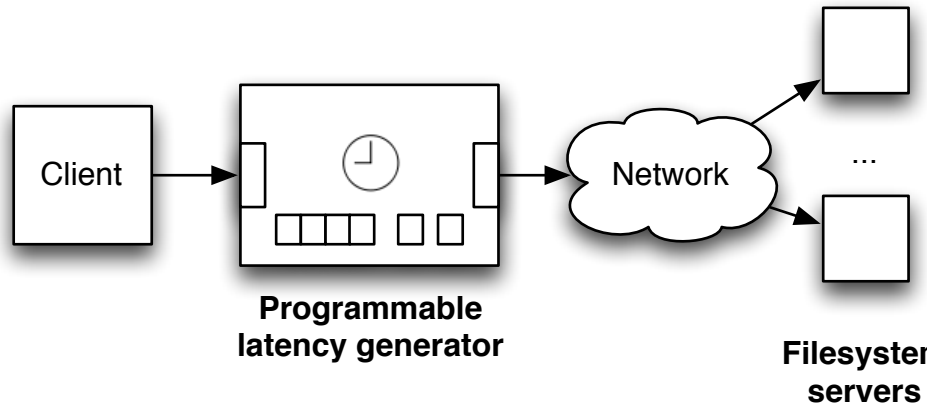

Figure 4.24: Setup for latency tests using a FreeBSD-based programmable latency generator

Both interfaces of the latency simulator were configured for transparent bridging, simplyfying the insertion of the simulator into an existing network link without making any changes to the environment. The dummynet module allows to selectively process packets of a specific type using *pipes* that can be parametrised with a delay, bandwidth or packet loss for simulation purposes. For the simulation, all IP and ICMP packets were delayed. The granularity of a delay depends upon timer setting in the BSD kernel. In this case, the kernel timebase was set at 1,000 Hz, which allows a precision of approximately 1 ms. The functionality of the latency simulator was validated using the standard UNIX *ping* tool with 64 byte messages. The measured roundtrip latency was double the configured one-way latency less

0.5 ms with a mean deviation of 0.3 ms. For example, when a delay of 5 ms was configured, the roundtrip latency as measured by ping was 9.5 ms. When comparing a direct connection to a simulator configured with 0 ms delay, the setup introduced a delay of 0.2 ms.

A comparison of AFS, NFS and Lustre with high network latencies

The performance of file creation in AFS, NFS and Lustre with 10 ms and 50 ms of round-trip latency are shown in Figure 4.25. It can be observed that all three file systems are very sensitive to latency changes: With a single process and a five-fold increase in RTT, file creation speed is reduced by a factor of five in all three file systems, clearly showing that servers are contacted for every single file create.

When more processes perform file creations in parallel, the results begin to differ: NFS shows an almost perfectly linear scaling from 1 to 10 processes, but both AFS and Lustre reach a plateau with four processes. This effect appears for both latency values of 10 ms and 50 ms, albeit at different performance levels. Thus, a performance limitation of the filesystem servers can be excluded from consideration.

A possible hypothesis for this bottleneck is the presence of a limitation that hinders more than 4 processes from performing their operations in parallel. All three file systems use RPCs to communicate with file system servers, so a possible limitation includes the number of simultaneous requests. With higher latencies, RPCs need more time to complete, thus with an increasing number of processes, it must be possible to issue more parallel requests to scale properly.

As discussed in section 4.3.2, with Lustre and NFS, the number of simultaneous RPCs can be tuned using parameters. Using the latency simulator, it is quite easy to verify their influence: Lustre allows only one modifying operation per server, so tuning the `max_rpcs_in_flight` parameter does not influence MakeFiles performance (see Fig. 4.26). For non-modifying operations such as StatFiles (see Fig. 4.27), performance changes as expected depending upon the number of simultaneous RPCs.

Using this server in a WAN environment, we could expect for example 1,000 / 6.5 = 153 file creations per second from LRZ to Heidelberg, 18 per second to Barcelona but only 3 per second to Melbourne. This is a *per-client-node* limitation that, unfortunately, also applies to large SMP machines, including the HLRB2. On an MPP cluster, the typical Lustre environment, the impact would be much less, as every node could execute one operation in parallel.

With AFS, the limitation is caused by a design decision in the underlying Rx protocol [Mit00]: Every Rx connection from a client to a server is limited to four simultaneous calls. AFS uses a connection to contact the server responsible for a particular volume. This means that all AFS volumes on a single server share one connection and the four simultaneous calls, while volumes on other servers use separate connections. Figure 4.39 shows that using additional volumes on additional servers does increase performance.

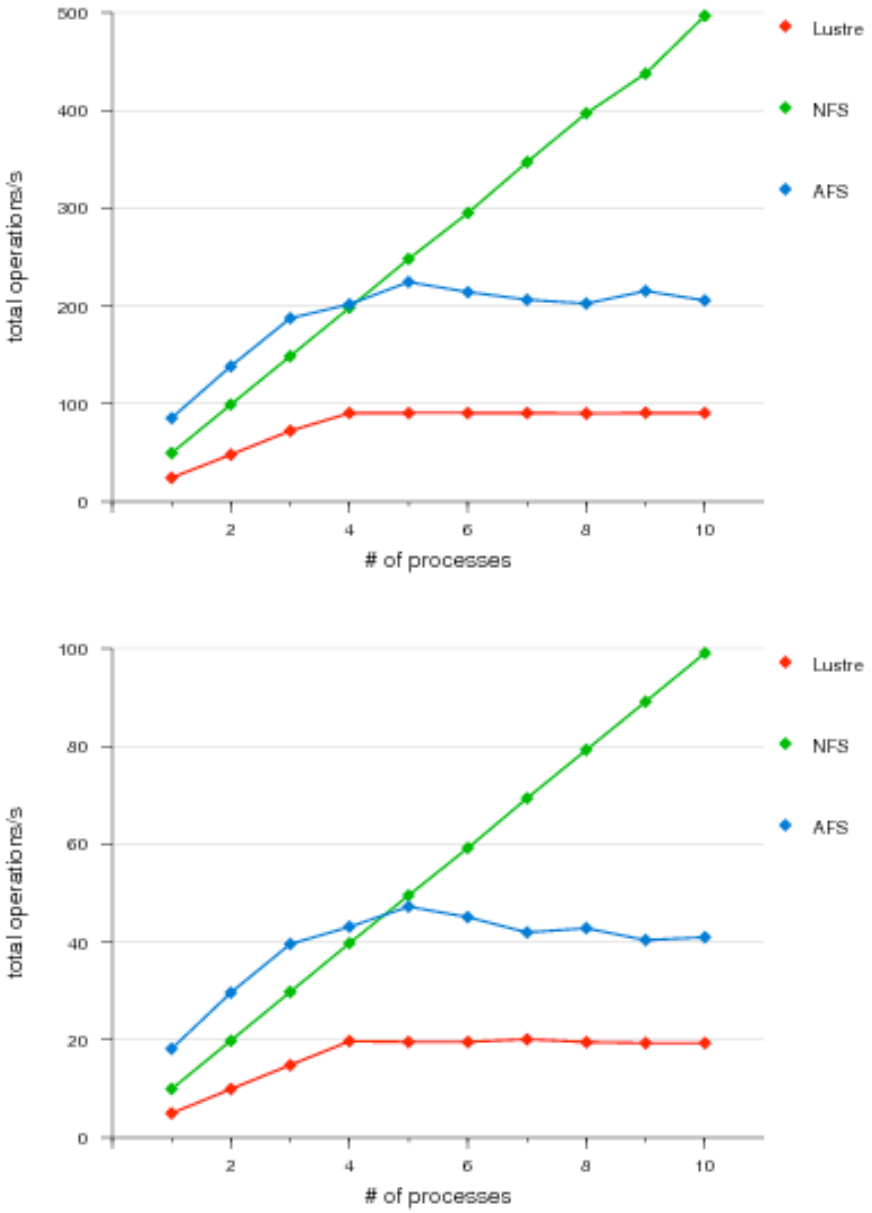

Figure 4.25: MakeFiles performance on a single Linux client using Lustre, NFS and AFS file systems and RTT values of 10ms and 50ms

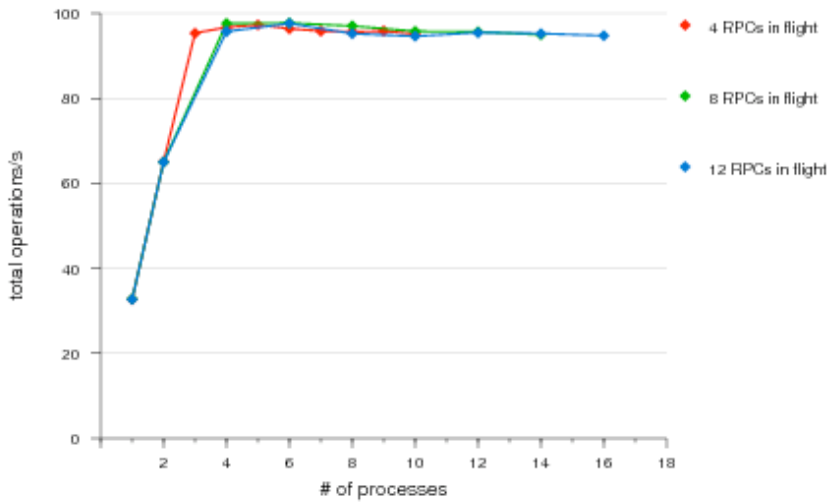

Figure 4.26: Lustre MakeFiles performance with 10 ms of latency shows the limitation to one simultaneous modifying call, which corresponds to 100 operations per second.

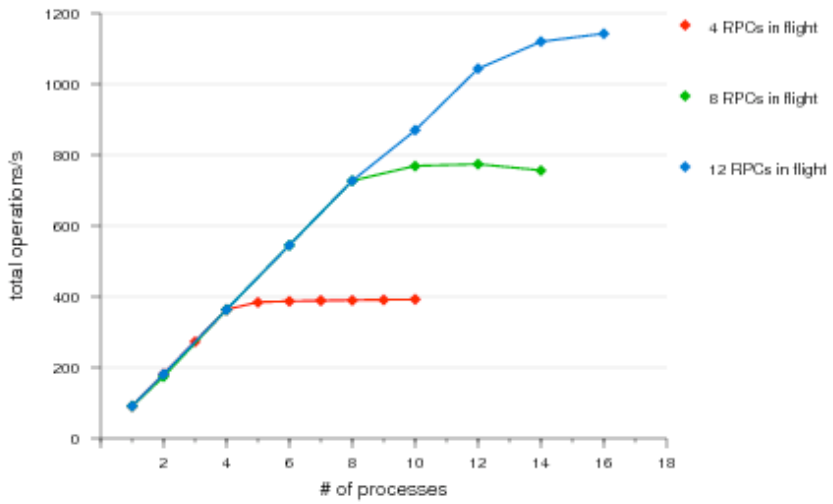

Figure 4.27: Scaling of non-changing metadata operations such as *stat()* depends on the maximum number of RPCs in flight.

How can the apparently linear scaling of NFS in Figure 4.25 be explained? The default value of 16 for the Linux NFS client tuneable */proc/sys/sunrpc/tcp_slot_entries* does not impose a limit on 10 processes. Figure 4.28 shows that this parameter does in fact significantly influence scaling. While its theoretical limitations are the same as with Lustre and AFS, NFS is the only file system that can be effectively tuned for higher metadata parallelism, because it supports concurrent modifying operations and up to 128 simultaneous RPCs.

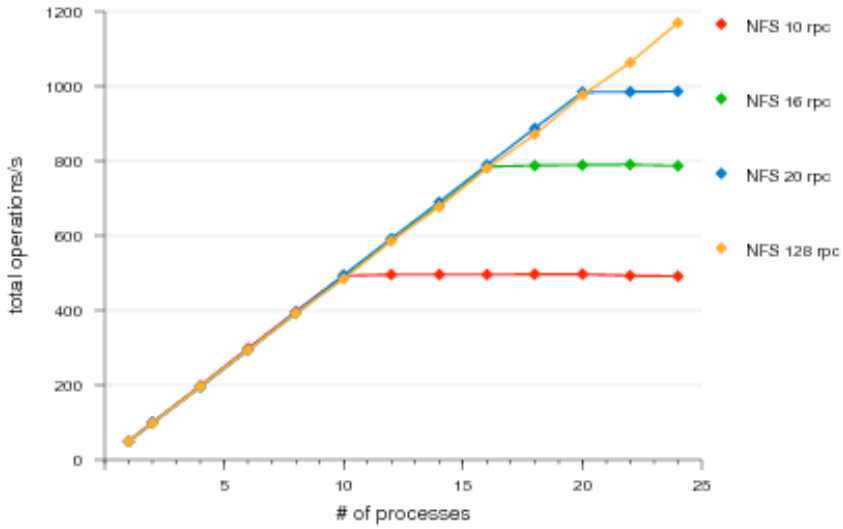

Figure 4.28: NFS MakeFiles performance with 10 ms latency, depending upon the number of allowed request slots

Summary

As the amount of international cooperation in high performance computing grows rapidly, network latency becomes a significant factor that must be considered when planning file systems for HPC grid environments. Artificially introducing latency is useful for examining the mode of operation and limitations of a file system. The default settings of popular file systems are optimized for neither high-latency environments nor high metadata concurrency as needed on large SMP nodes. Depending upon the type of the file system, concurrency may be the only option to improve performance.

4.7 Intra-node and inter-node scalability in namespace aggregated file systems

Filesystems with namespace aggregation present a single hierarchical namespace where parts of the namespace called *volumes* are potentially placed on different storage systems.

This promises – at least in theory – scalability in terms of both capacity and performance because different parts of the namespace hierarchy are independent.

Depending upon the the file system, the client is aware of the existence of different servers and contacts them directly (as does AFS) or file system operations are forwarded inside the storage system (as does Ontap GX). This creates a multitude of options for parallelizing access. This section presents several tests that aid in assessing potential performance improvements.

4.7.1 Single-client measurements on Ontap GX

From a client viewpoint an Ontap GX storage system appears exactly like a normal NFS file server in which the client node mounts exactly one node in the storage cluster using a single mountpoint.

For the following single-node measurements, a 16-core SMP machine[¶¶] that mounted the Ontap GX storage system from the HLRB2 (see section 4.1.3) using NFS version 3 was used. In all tests, the number of simultaneous RPC requests was set at 128 (see section 4.6).

Direct access and forwarding

The first consideration in a system such as Ontap GX is whether the internal forwarding process in the storage cluster leads to performance differences compared to the performance using a direct volume access. On the one hand, forwarding a request to another storage node could induce a performance penalty. On the other hand, the total workload is then divided between two storage nodes, which might improve performance.

Figure 4.29 shows that client C1 has an NFS mount to storage node S1. If volume V1 is accessed, all operations can be processed directly by S1 because it is the owner of this volume. In comparison, client node C2 mounts storage node S3. If C2 works with data on volume V2, then storage node S3 must send all operations to S2 for processing.

The first benchmark setup closely follows this model: Three plugins of DMetabench (MakeFiles, OpenCloseFiles and StatFiles) were run on a single client node using up to 64 processes. In the first case (”direct access”), the data target was a single volume placed on the same storage node that the test client had mounted. In the second case (”forwarding”), a volume on another storage node was chosen. Figure 4.29 shows that there is generally very little difference between the two setups. However, there is a slight advantage with up to 16 processes for the StatFile measurement only.[***]

It should be noted that this only means that the difference between direct access and forwarding is not obvious when using *a single client node*. Therefore other factors might limit performance before a difference becomes significant.

[¶¶] Sun X4600 with eight dual-core Opteron CPUs.

[***] The lower value for StatFiles with 36 processes is an outlier that can be checked using the time-based graph.

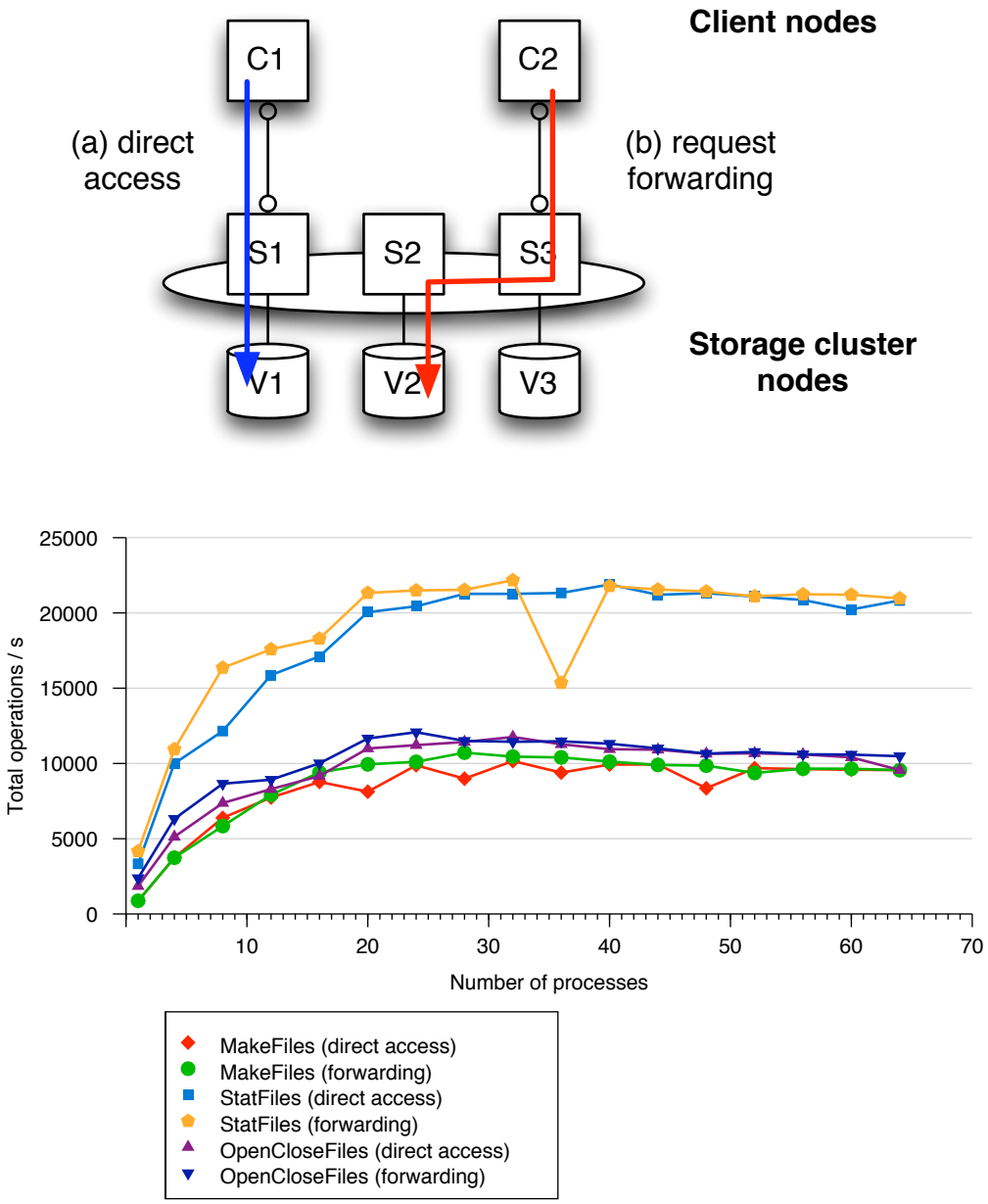

Figure 4.29: Request processing for direct and forwarded requests in Ontap GX

One vs. multiple volumes

Can performance be improved further by distributing operations to more volumes placed on multiple storage nodes? In a setup such as that in Fig. 4.30, client C1 still mounts only a single storage node (S1), but uses multiple volumes.

As the Ontap GX cluster tested has eight nodes, the same number of volumes was created and one volume placed on every GX node. The chart in Figure 4.30 shows the same three operations with one volume compared to eight volumes; again, very little difference can be observed so the testing of different numbers of volumes (2,3,4,...) is unnecessary. At this point, there is no performance gain with multiple volumes for a single node.

Multiple mountpoints and storage nodes

Because single-node performance does not appear to change with the number of volumes, attention should be directed to the client side. Although a client normally mounts only one storage node with Ontap GX, the accessing of multiple storage nodes can be simulated by adding more NFS mounts. Figure 4.31, in which the client has one mountpoint for every node in the storage cluster, shows this configuration. Naturally, this contradicts the idea of a single namespace on the client but, in a benchmark scenario, allows determination of which part of the setup is responsible for the performance plateau. The same volume is accessed through all mountpoints.

Figure 4.31 shows significant performance improvement for StatFiles when the metadata load is distributed over multiple mountpoints and multiple storage nodes are contacted. For OpenClose and MakeFiles, the difference is smaller but still clearly noticeable.

Finally, it is also interesting to observe the combination of multiple mountpoints and multiple volumes. The test volume configuration for this measurement is more complex: There are eight NFS mount points (`/mnt1` to `/mnt8`) where the eight Ontap GX nodes are mounted. Because the path configuration file for DMetabench specified that a volume on *another* GX node is accessed in every case, all operations must be forwarded (see Table 4.5).

Table 4.5: Mountpoint and volume assignment with multiple mountpoints and volumes

Client mountpoint	Ontap GX node mounted	Node hosting accessed volume
`/mnt1`	Node 1	Node 2
`/mnt2`	Node 2	Node 3
...	...	...
`/mnt8`	Node 8	Node 1

The results in Figure 4.32 appear very similar to those obtained with a single volume, leading to the conclusion that with this particular client, the communication between client

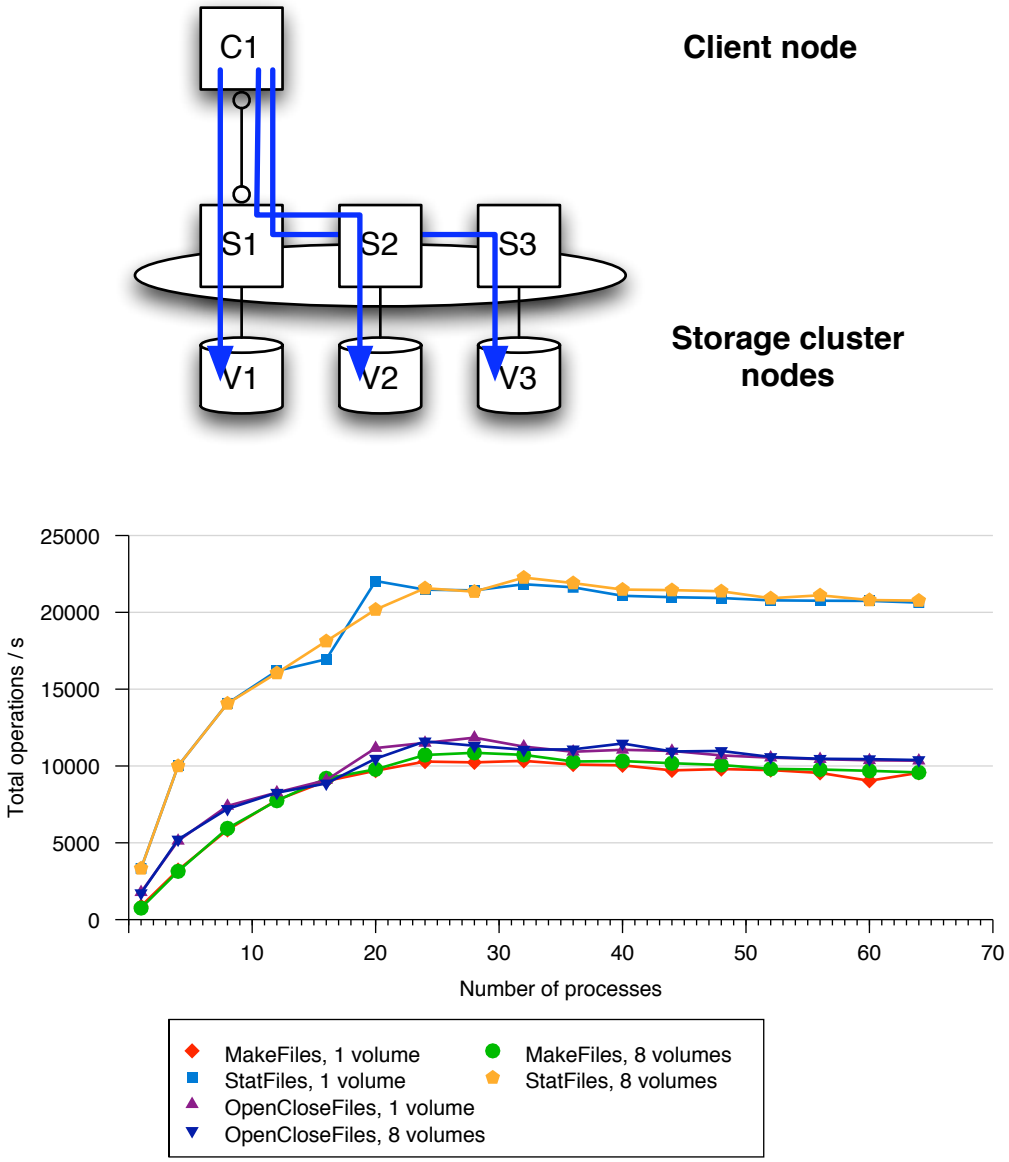

Figure 4.30: Single vs. multiple volumes in Ontap GX

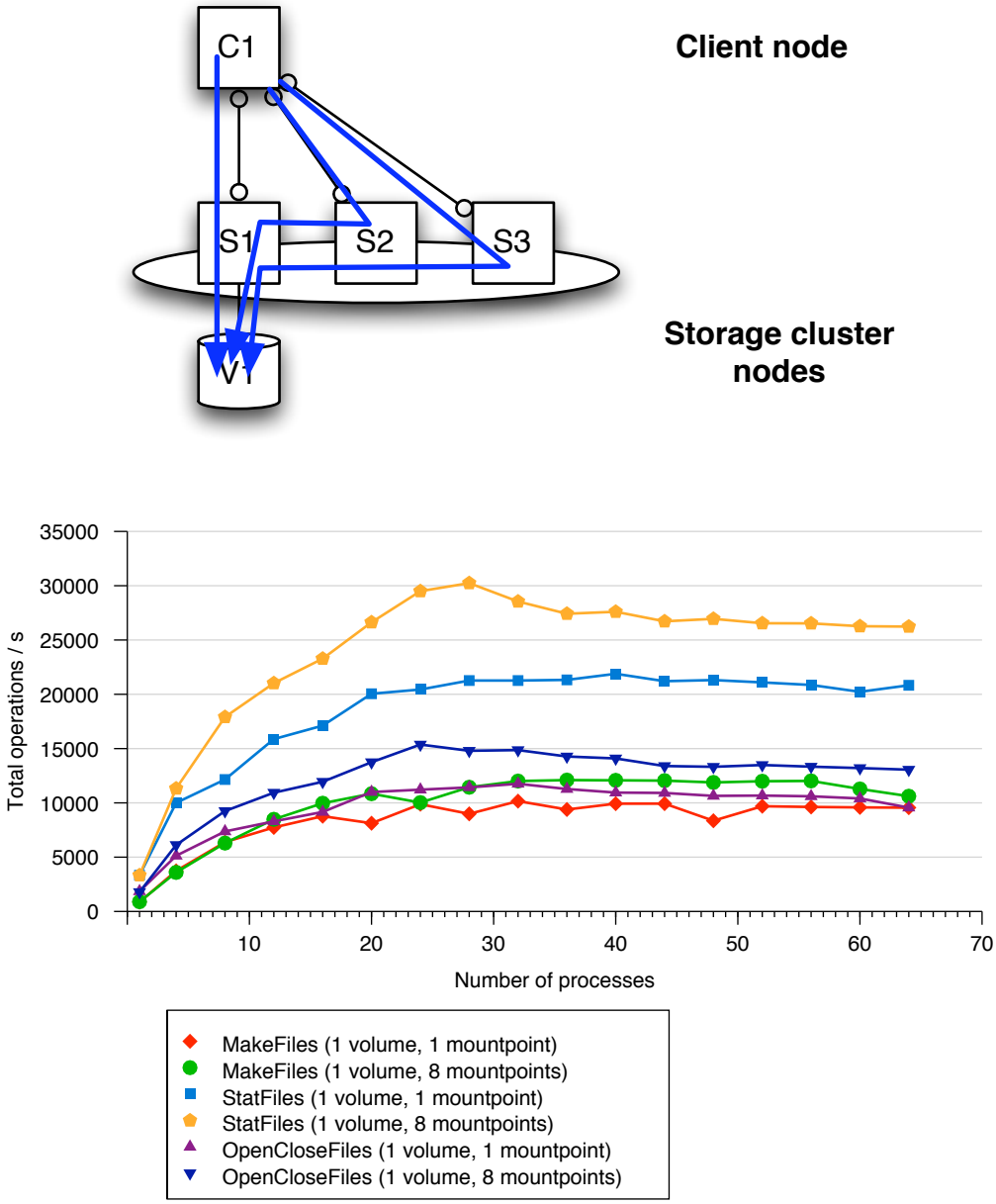

Figure 4.31: A single volume accessed through multiple mountpoints and storage nodes

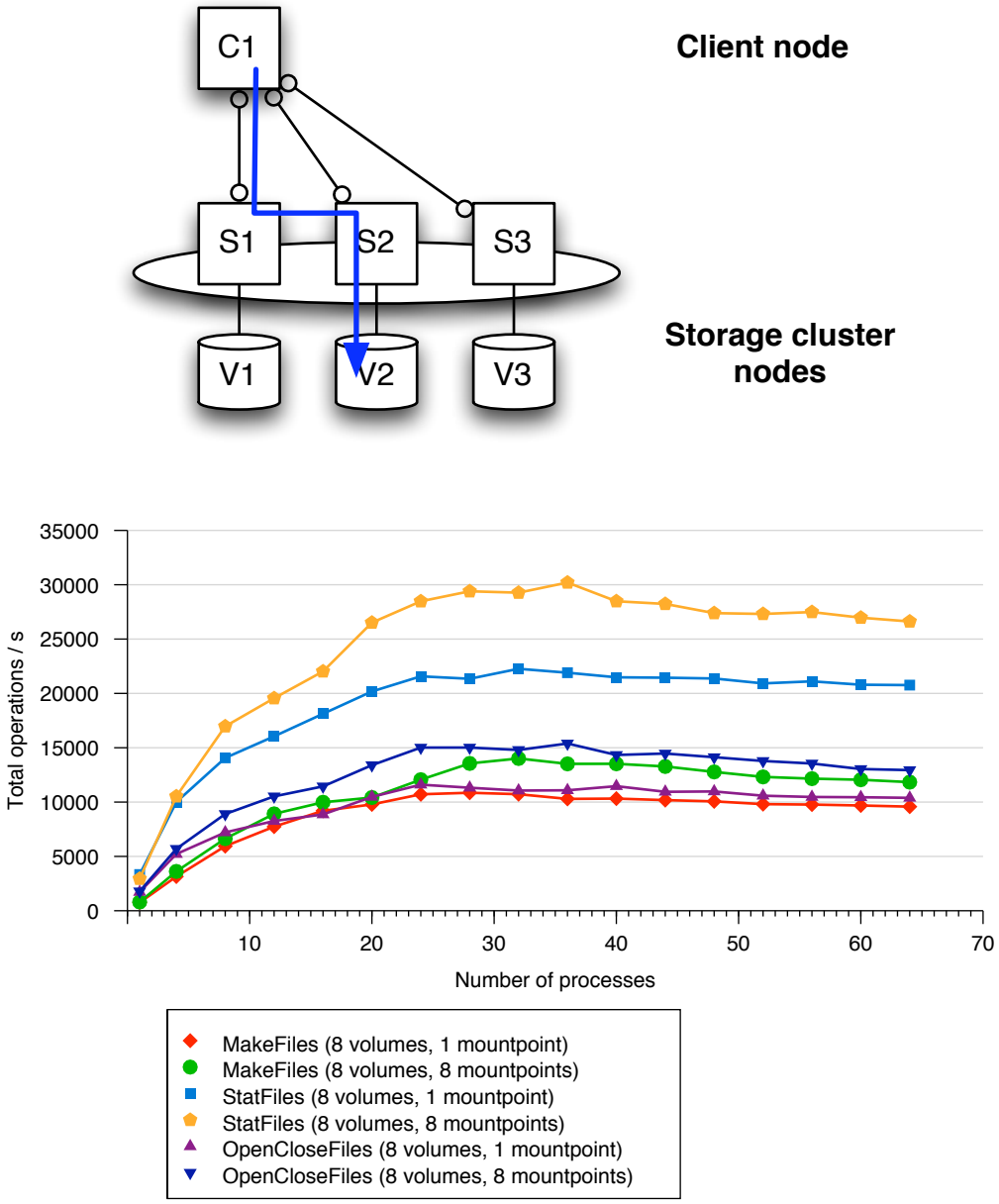

Figure 4.32: Single client accessing eight volumes through one or eight mountpoints . Because the destination volume is always on a storage node other than the mountpoint, all requests are forwarded.

and server is the dominating factor. This also means that the scalability here is probably limited by the client and not the storage system.

4.7.2 Multi-node operations on Ontap GX

An assesment of multi-node scalability of Ontap GX was performed on the HLRB2. As the HLRB2 is a production system, it was not possible to exactly align the compute nodes (partitions) and the storage nodes. Up to 10 partitions of the HLRB2 were used for the benchmark using two basic configurations with either one or six processes per node. Six processes were chosen because this was the number that had achieved the highest performance *per partition* in preparation measurements. For every configuration, the four DMetabench tests MakeFiles, MakeFiles65byte, OpenClose and StatFiles were performed and one, two, four, six and eight volumes, placed on different storage nodes, were used.

No special attention was given to which nodes mount which filers and where the volumes were placed; thus, this was a very realistic scenario wherein a user application is assigned some nodes by a batch system and has no choice regarding how to access the data. This means that a large proportion of the requests had to be forwarded by Ontap GX. Additionally, because there were 16 data interfaces on the eight filers but only 10 nodes were used, the load distribution was again not perfect. The results are shown in Figures 4.33 to 4.36 (please note the different axis scales on the top and bottom charts).

Clearly, advantages in metadata performance are only observed when the total load on the file system is sufficient. On the HLRB2, using only one process per node/partition is not sufficient, and therefore provides hardly any advantage.

With multiple processes per node *and* multiple nodes the situation changes dramatically: For example, in the MakeFiles benchmark (see Fig. 4.33, bottom chart), performance for one volume reaches a plateau with four nodes. With two volumes, the plateau is reached with 8 nodes and at double performance. Using 8 volumes scalability is almost linear for up to 10 nodes. Measurements for eight and 10 nodes in the top chart had interferences with other applications on the storage system (a verification was performed using time chart).

The situation is very similar for very small files (MakeFiles65byte), opening and closing (OpenClose) and StatFiles. In absolute numbers, almost 40,000 file creations per second or 45,000 open/close cycles with only 10 client nodes can be observed. As every additional volume seems to add 12,000 file creations or almost 20,000 open/close cycles per second, the Ontap GX storage subsystem of the HLRB2 was still not saturated in this test.

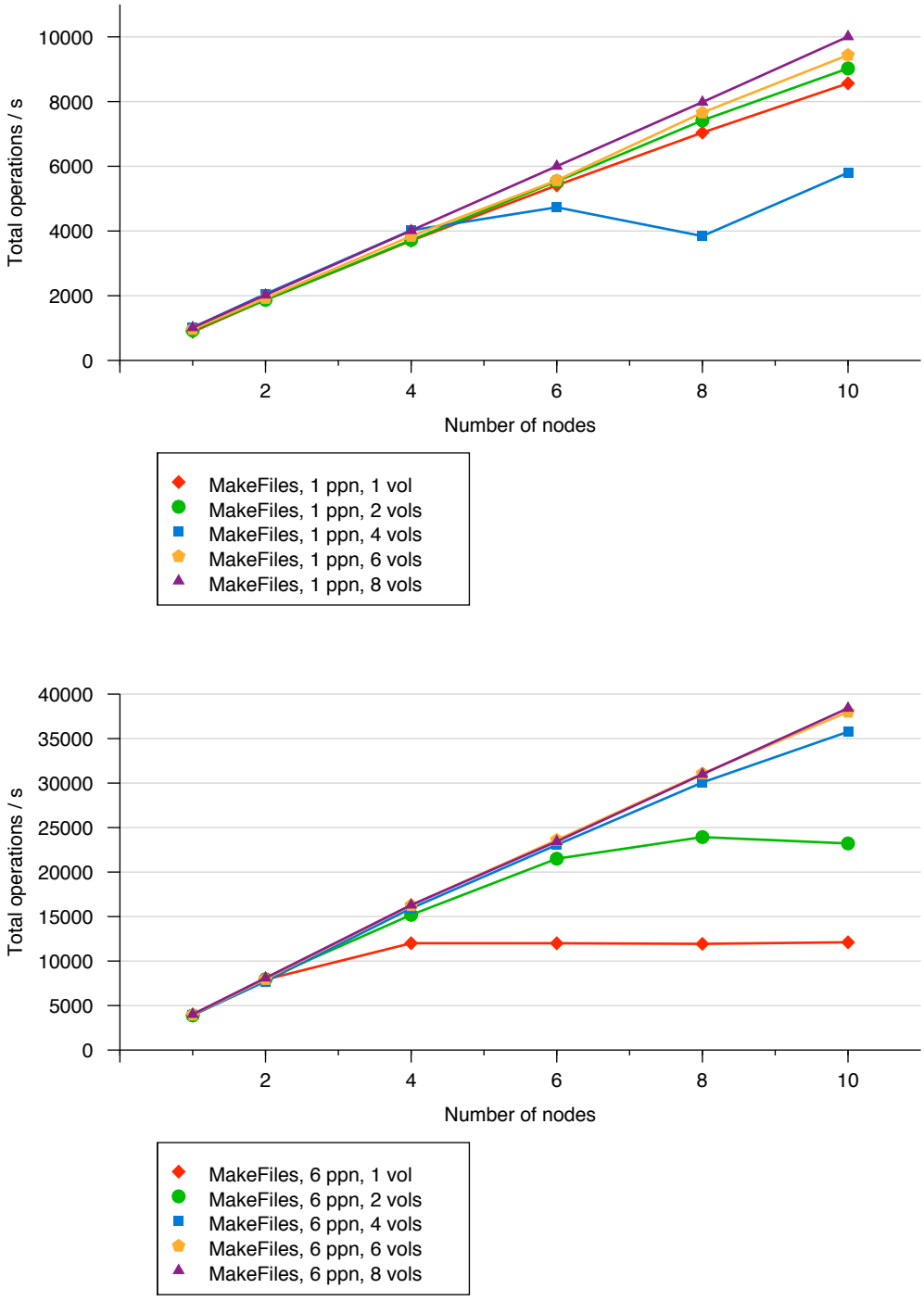

Figure 4.33: Scalability of MakeFiles on the HLRB2 using Ontap GX and a variable number of file system volumes placed on different filer nodes. The top chart shows one process per node and the bottom one six processes per node.

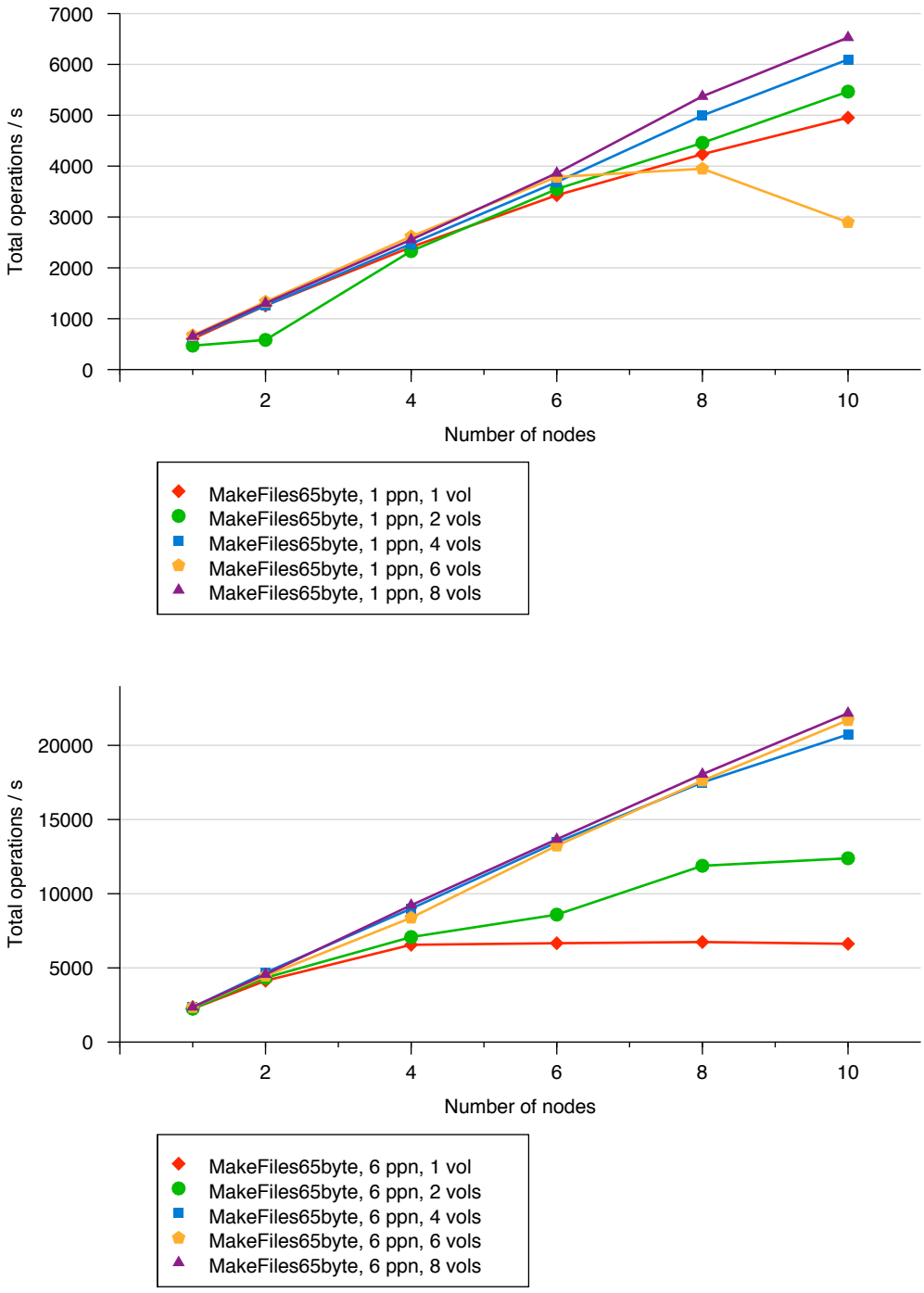

Figure 4.34: Scalability of MakeFiles65byte on the HLRB2 using Ontap GX in a setup similar to that in Fig. 4.33

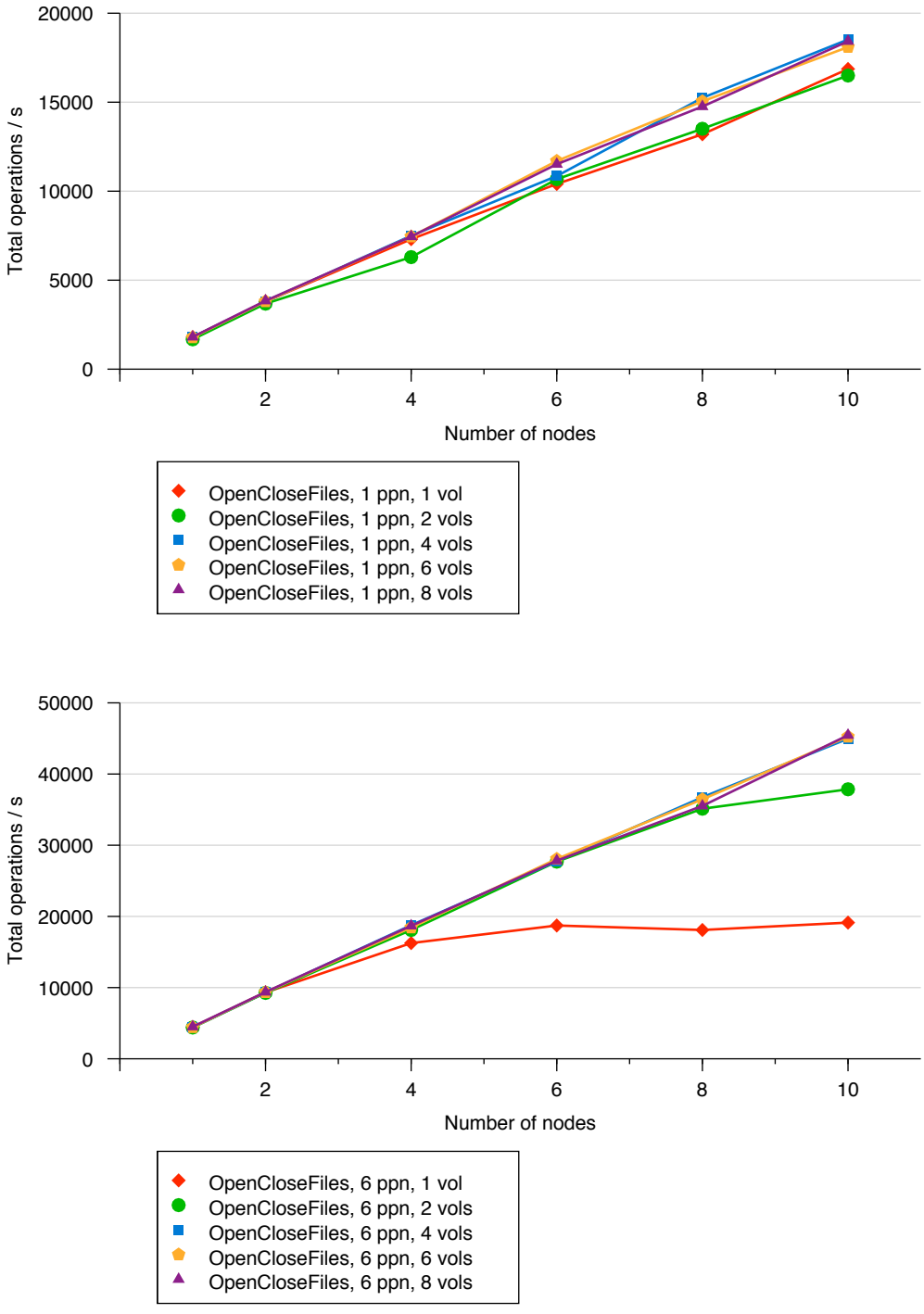

Figure 4.35: Scalability of OpenClose on the HLRB2 using Ontap GX in a setup similar to that in Fig. 4.33

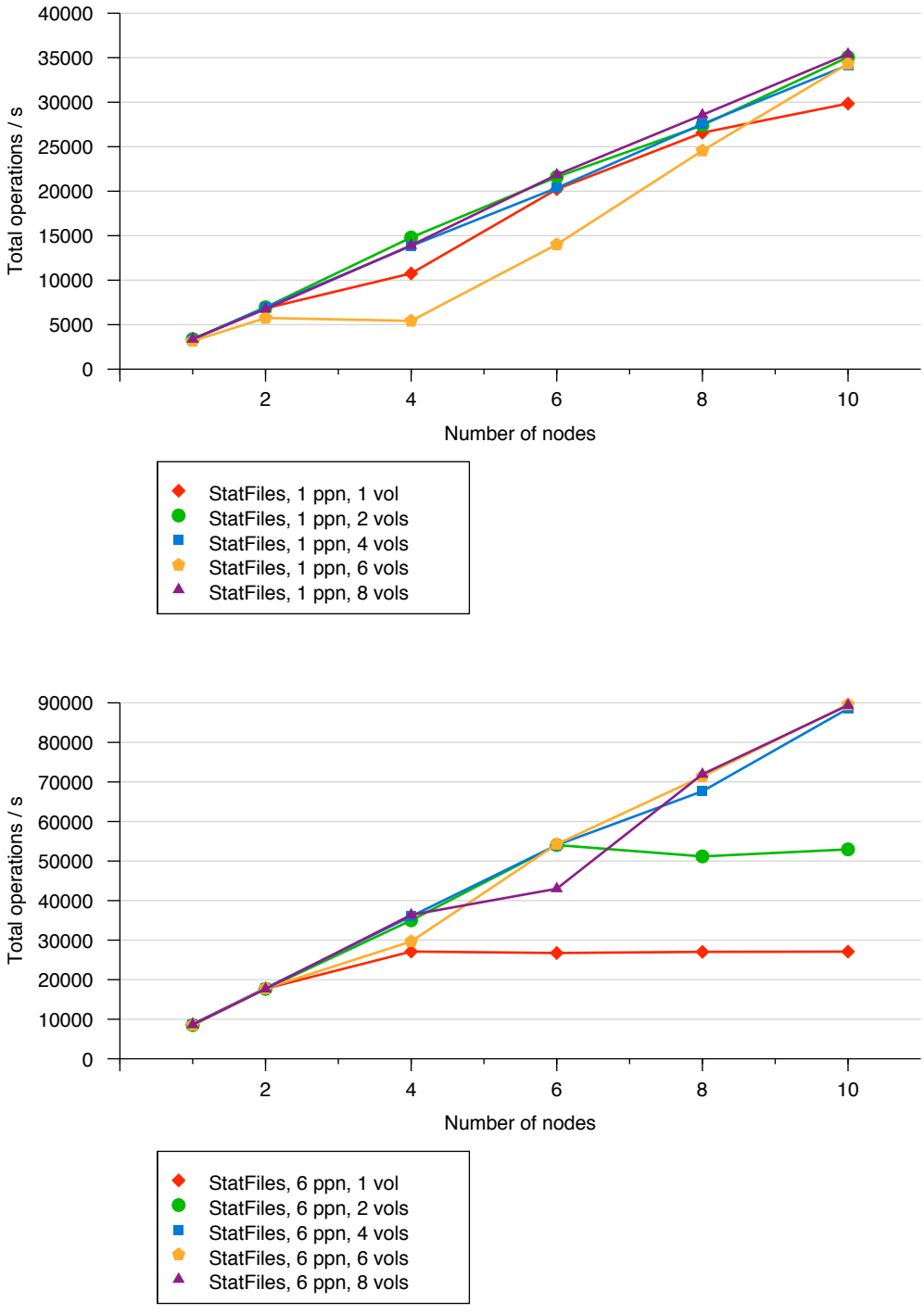

Figure 4.36: Scalability of StatFile on the HLRB2 using Ontap GX in a setup similar to that in Fig. 4.33

4.7.3 Measurements on AFS

Although the Andrew File System (AFS) also delivers a single namespace,[†††] there is a fundamental difference to Ontap GX: With AFS the client determines the volume location and contacts the appropriate servers (see Fig. 4.37). In AFS there is no forwarding of requests because the client communicates directly with the server hosting the volume. This also means that there is always only one network path from a client to a particular volume.

The following measurements were made with an early version of DMetabench while AFS was still being used on the LRZ Linux cluster.

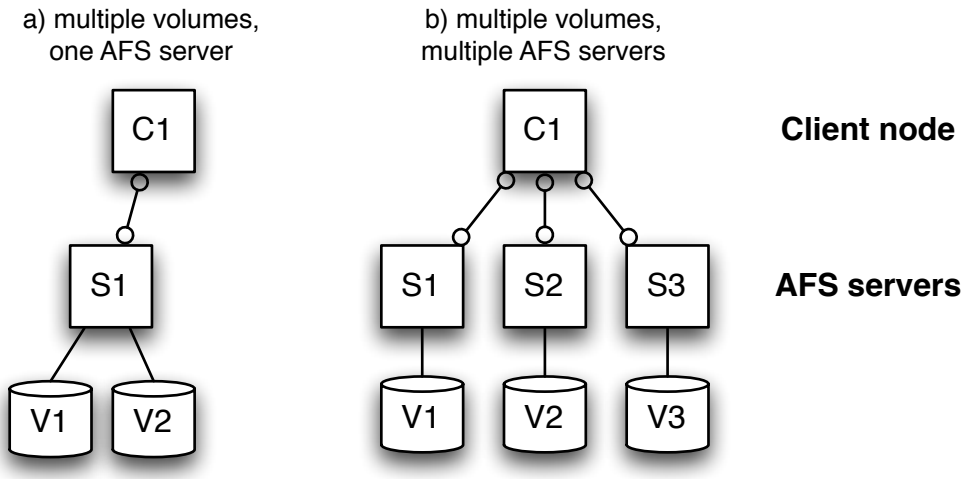

Figure 4.37: The AFS client manages connections to different AFS servers by itself. The number of simultaneous requests to single server is hard-coded to four in the *RX protocol* used by AFS.

Single server, multiple volumes

Figure 4.38 shows MakeFiles results from one to four AFS volumes residing on the same AFS file server. Unfortunately, the client machine hung with more than 28 simultaneous processes and thus the process count has been limited to 20. It can be seen that performance does not significantly increase with more than four processes, probably because there is a hard-coded limit of four simultaneous operations to a single AFS server (see section 4.6). Using additional volumes also does not increase performance if all the volumes are located on the same AFS server.

[†††] The AFS namespace is even *globally unique*.

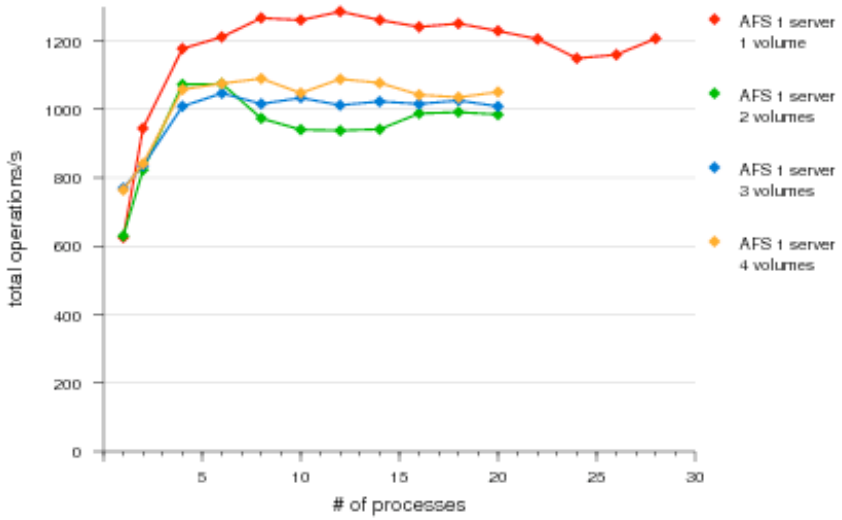

Figure 4.38: Single-client MakeFiles performance with multiple AFS volumes on the same AFS server

Multiple servers, multiple volumes

If multiple volumes are located on different servers, the AFS client starts and maintains a separate connection to every AFS server. In theory, performance should scale linearly with the number of AFS servers involved because for every additional server four more request slots are available to be utilized simultaneously. In fact, Figure 4.39 demonstrates a significant improvement.

Comparison of AFS and Ontap GX

Both AFS and Ontap GX can distribute parts of the namespace to multiple servers. Their most interesting difference is how performance scales on a single node with multiple processes: If AFS volumes belong to different servers, each volume adds significant metadata performance. In contrast, the NFS client still communicates with only one server in Ontap GX, so using more volumes does not automatically improve performance. On the other hand, the absolute numbers for AFS are quite low (”bad performance scales well”), but the concept of using multiple communication paths appears useful. Unfortunately for NFS v3, multiple mountpoints to a single-namespace filesystem change the semantics for multiple processes on the same node to the multiple-node semantics and open-close coherency, because the operating system is not aware that the mountpoints represent the same file system.

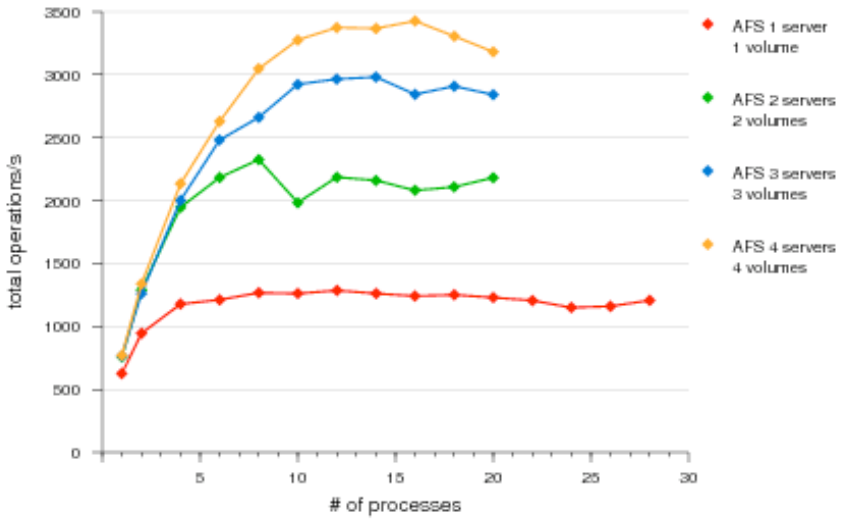

Figure 4.39: Single-client MakeFiles performance with multiple AFS volumes on multiple AFS servers

4.8 Write-back caching of metadata

Up to this point, all file systems presented (NFS, AFS, Lustre and CXFS) triggered communication with file system servers when a file was created using an `open()/close()` pair. A different approach that is possible when a write-back metadata cache is used has been discussed in the file system community for many years.

With a write-back metadata cache, the communication with the file system is delayed so that, for example, multiple file creations can be transmitted at the same time. Naturally, the semantic changes dramatically because locking becomes necessary to guarantee the uniqueness of file names. The challenges are quite similar to those of disconnected operations in file systems (e.g., in Coda [Sat06]) but it can be assumed that the network connection is working (even if one tries to avoid using it). This section focuses on the potential performance improvements offered by write-back metadata caches.

A very recent, but functional filesystem prototype for Linux implementing write-back data and metadata caching is POHMELFS written by Evgeniy Polyakov [Pol08]. The first public version of POHMELFS was tested using DMetabench and the MakeFiles benchmark.

As there were different problems with stability, only a very simple two-node setup was possible: one node ran the POHEMELFS server and the other was the client. The version tested flushes the cache only after an explicit `sync()`, so MakeFiles has been modified to issue a `sync()` every 5,000 files; otherwise the entire benchmark would run in the client's cache memory. As a performance baseline, the Linux NFS server was used on exactly the same server machines using both sync and async modes. The result is shown in Figure 4.40 .

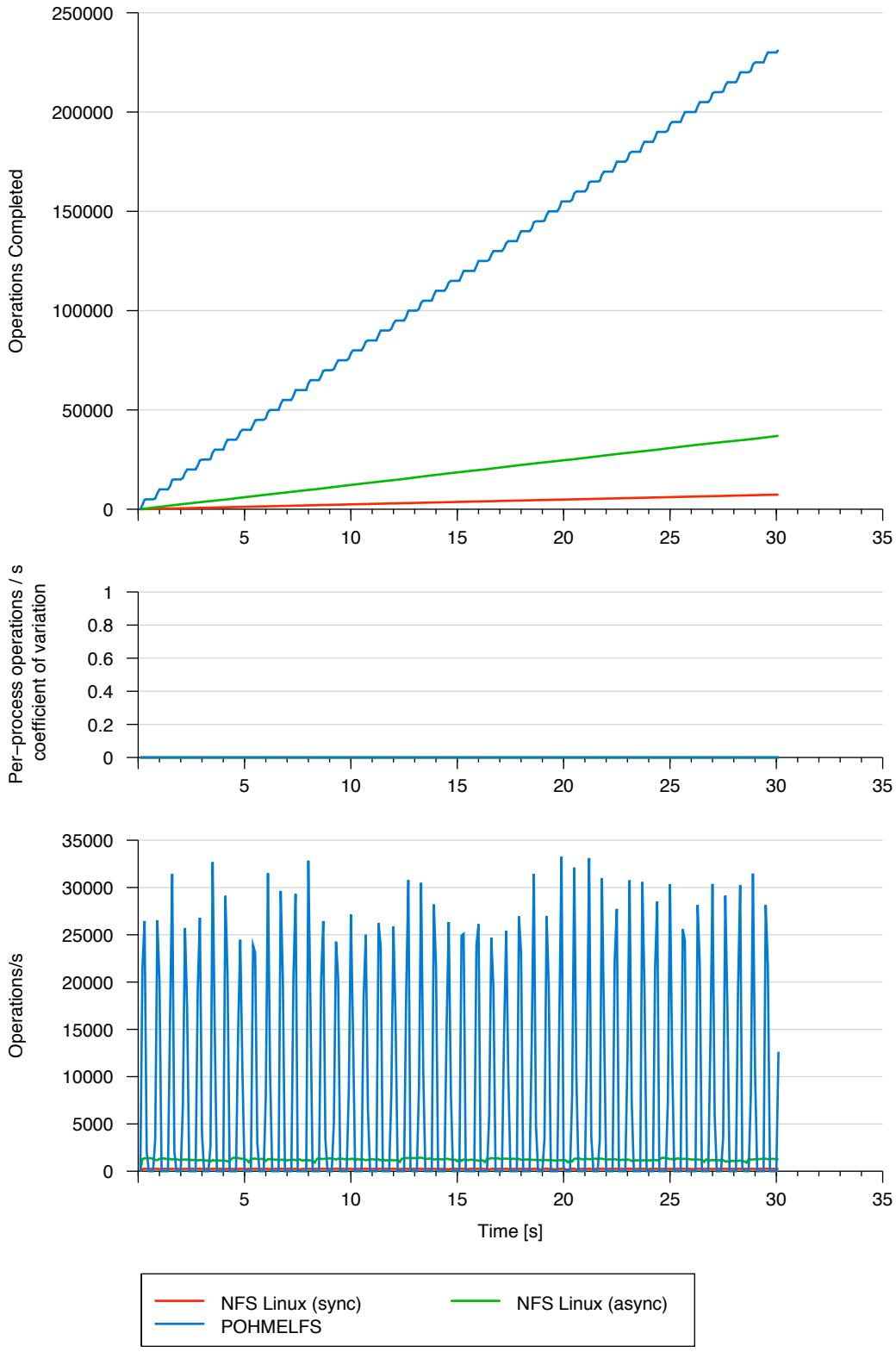

Figure 4.40: POHEMELFS with a metadata write-back cache compared to NFS

Considering that only one process on a single client is used and an artificial `sync()` command was added, the performance gains compared to NFS are significant - more than 7,500 file creations per second were achieved. The number of client-server roundtrips is significantly reduced by the metadata-capable write-back cache, and an efficient batching of operations becomes possible. The write-back behaviour of this version of POHMELFS is, however, very different even compared to that of local file systems, as there neither any time-based flushout nor provisions for write-back error handling. Practically, this means that POHMELFS could happily create 800,000 files in memory and then completely fail because the number of inodes (available files) on the server had been exceeded.

With proper locking and resource allocation mechanisms, write-back caching might become a significant method of improving write performance in distributed filesytems because it addresses one of the major problems: latency. Unfortunately, only after these necessary functions are implemented will it become possible to test the file system using multiple clients. Finally, the interesting fact that a distributed filesystem does not necessarily implicate multiple client nodes comes to mind.

4.9 Conclusions

A system-level test has shown how DMetabench can successfully aid in identifying disturbances during complex measurements. Multiple benchmarks on different, real production file systems at the LRZ confirmed the portability and file system independence of DMetabench. The number of simultaneously issued RPC requests was identified as a common limitation of NFS, Lustre and AFS. Increasing the limits proved necessary to raise performance on large SMP systems or in high-latency environments.

An important result is that metadata performance cannot be taken for granted even in so-called ”high performance” file systems. In absolute numbers, both Lustre and CXFS can deliver a few thousands of file creations per second. The performance limitations of a single metadata server can be resolved using namespace aggregation, which was demonstrated using both Ontap GX and its predecessor, AFS. Alternatively, for single-process performance write-back metadata caches promise great improvements and the removal of latency effects.

Chapter 5

Conclusions and future research

DMetabench, the novel parallel benchmark framework presented in Chapter 4, allows to conduct an experimental assessment of detailed metadata performance properties of real-world storage configurations. This section recapitulates the important findings and discusses the new research questions that have emerged from the results.

5.1 Using DMetabench for measuring metadata performance

Unlike existing benchmarks, DMetabench is a metadata micro-benchmark that focuses upon parallel, multi-process execution on one or multiple nodes. Its novel time-logging capability permits the recording of performance for every single process within regular time intervals during each measurement.

In the benchmarks presented, up to 500 parallel processes on up to 20 nodes were used in the measurements. By itself, this fact does not guarantee scaling in even larger environments, but as DMetabench belongs to the class of ”embarrassingly parallel” applications, there is no inter-process communication involved during the benchmark run and no architectural limitations. In practice, all tested distributed file systems could be easily saturated.

Workloads

As a benchmark framework, DMetabench can execute user-defined and customized metadata operations. This thesis concentrated on basic metadata use cases like file creation or getting file attributes, rather than complex or even application-like workloads. This allowed to precisely stress particular properties of the filesystems tested. Although DMetabench is very useful for optimizing and tuning distributed file system environments, it is less useful as a basis for a standard metadata benchmark that could be used to ”officially” compare filesystems. This is because the microbenchmark property would oversimplify the problem, which might push developers into optimizing corner-cases of the benchmark. While this is

true for any microbenchmark (also in CPU or system benchmarking) microbenchmarks are still very useful to precisely observe particular operations.

Time-based logging

Using the novel approach of logging performance within regular intervals during the benchmark permitted observation of time-related effects in file systems, such as flushing write-back caches or writing consistency points, as well as identification of external disturbances during of a benchmark run. This approach aids in obtaining useful data even if measurements are taken in production environments where it cannot be ruled out that other applications are interfering with the benchmark.

Graphical analysis

The data that results from performing benchmarks using DMetabench is inherently multi-dimensional: It includes measurements for different combinations of nodes and processes and time-based data is available for every combination and operation. Three basic graphical representations (timeline, per-node and per-process) can be used to compare different measurements so that the benchmarking and analysis process can be greatly accelerated.

This work was primarily based on qualitative observations and comparisons. While quantitative data is also available from DMetabench, the inevitable progress of hardware development makes absolute numbers obsolete within several months while the architectural properties and limitations remain constant.

Benchmarking best-practice

Several best-practice techniques developed from practical experience can be helpful when benchmarking metadata.

The first practice is that creating files (*MakeFiles* and its variations in DMetabench) as probably the most useful micro-benchmark for metadata. Doing so specifically stresses the uniqueness of file names in a directory, the single semantic guarantee common to all file systems, forces the file system to apply its mechanisms for consistency control and tests distributed system latencies. If the runtime is sufficiently long (longer than write-back cache flush intervals), storage systems will make the newly created files persistent and allow even more interesting observations.

A second technique is to push every file system to its performance limits and beyond. Many effects cannot be observed when only a light load is applied. Incidentally, this relates to another strong point of DMetabench, as DMetabench permits the use of inter-node and intra-node parallelism to generate loads *impossible to achieve* with sequential benchmarks.

Comparing filesystems can be very difficult if test clients are a smaller subsets selected from a larger population. In such a situation, it is reasonable to run tests for all file systems in question directly after each other using multiple `mpirun` calls from the same batch job.

Adding artificial latency

Latency purposefully inserted into a distributed file system can help to identify communications mechanisms and architectural limitations. The fact that DMetabench can push file systems to their limits by using parallel operations was discussed earlier in the thesis. Using additional latency, the behaviour of a single client can be isolated while avoiding the side effects of high loads on the file servers or storage systems. Thus, artificial latency is useful in assessing file system communication protocols.

5.2 Status of metadata performance

Chapter 4, which described several measurements of large, production distributed file systems, related an astonishing result: A small number of multi-core nodes easily saturate the most advanced commercially available file system servers when metadata operations are involved, several even becoming overloaded up to the point of temporary failure.

When *data* throughput is considered, the situation becomes very different, as much research has already been conducted on improving and parallelizing data movements and data operations. Data operations on different files are, from a semantical point of view, embarrassingly parallel. Similar work in the area of metadata performance scaling is just beginning. The filesystems tested used a single metadata server for the entire file system.

Differences between dedicated HPC file systems and normal user file systems continue to exist. The former, including Lustre and CXFS, are designed for data throughput on large files. A point worth discussing is whether the academic trend towards parallel file I/O (multiple processes accessing a single file) was, at least to some extent, caused by the poor metadata performance of early parallel file systems. If many parallel processes use a single common file, the number of metadata operations is greatly reduced. Users at the LRZ have been very slow in the adoption of new I/O models on the Linux cluster and the HLRB2. First, LRZ tuning efforts are focused on computational performance. Second, existing commercial HPC applications often depend upon standard POSIX-like I/O and user applications tend to stick to the one-file-per-process or – even worse – master-I/O-process models. Thus, high metadata performance remains crucial, as a major shift towards parallel file I/O is currently unobservable.

End-user file system protocols, such as NFS, are driven by different needs. Historically, they have been focused on interactive usage from end-user workstations and a more heterogenous system environment that included more operating systems and versions than has

the typical HPC cluster. Additionally, single clients often cannot be trusted, which complicates the implementation of elaborate client-server interactions (e.g., aggressive client caching). Moreover, workstation environments include the possibility of uncontrolled client reboots that lead to simpler, "per-node" failure recovery models.* Thus, the isolation of clients is a major design target for this kind of system.

5.2.1 Options for improving application metadata performance

For a single-threaded application, the communication latency between the client and the metadata server largely determines performance because access coordination occurs on the server. Thus a client must contact it before every operation.

Technical improvements

Progress in CPU performance and faster network interconnects (e.g., Infiniband) help reduce latency in LAN environments without requiring changes to file system protocols. Unfortunately, filesystems in WAN environments will always experience a certain amount of latency determined by the length of the communication paths.

Local metadata caching

Two techniques have been developed to bring network filesystems to the performance level of local filesystems for single-threaded applications. This first one, which is used by GPFS, is to delegate the metadata server responsibilities for a file to the client that uses it. The second technique is to use a write-back metadata cache. As the benchmarks with POHMELFS have shown, there is great potential in this type of solution. However, the persistence and failure semantics of a file system with local metadata caching are significantly different than those defined by POSIX: Caching the data in volatile memory creates more opportunities for data loss. From an application perspective, one might assume that all applications that rely on persistently storing files for transaction purposes will issue the necessary calls, such as `sync()`, to ensure that data is written to the servers and becomes persistent. It would be interesting to compile the assumptions about local file systems made in popular client and server applications.

Other problems with a write-back metadata cache are the complex recovery procedures required when clients crash and the theoretical possibility of the propagation of corrupted metadata from clients to the server. Practical research into the impact of recovery procedures within different scenarios (e.g., workstation environments and HPC cluster environments)

*In contrast cluster file systems such as CXFS regard the failure of a client as a major event because its locks must be released.

and the development of techniques for avoiding the propagation of corrupted metadata would be very useful.

Increasing parallelism for metadata operations

An alternate way to accelerate metadata-intensive workloads is to replace single-threaded execution by a parallelized execution. Doing so requires modifications to the application but can improve performance in current file systems by an order of magnitude, even for a single node.

As previously discussed, in contrast to distributed file systems, local file systems do not experience large improvements with an increasing number of parallel processes. This is partly due to latency and the fact that large storage systems used in distributed file systems are optimized for throughput using a higher degree of operation parallelism (e.g., using multiple disks and striping to enable distributed I/O). This optimization creates a conflict because local file systems do not truly benefit from parallel operations, whereas it might be the only performance-enhancing option available for existing distributed file systems.

5.3 Parallel metadata operations in distributed file systems

The possibility of parallelizing metadata operations in an application depends upon the amount of a priori knowledge about the files that is available. For data management applications (e.g., copying, backing up, archiving) it is possible to obtain information about the data set and distribute it among multiple processes. If data creation is triggered by external events (e.g., on mailservers) or calculations, dependencies may result in a re-serialization.

5.3.1 Distribution of metadata operations

The administrative operations of file systems, such as backing up or finding files, are often trivially parallelizable by dividing workloads among multiple processes running on one or multiple nodes. Unfortunately, these administrative tasks are often performed by very simplistic scripts or legacy utilities such as `find`. These programs are not susceptible to the complexity of parallelism.

Google's MapReduce sparked new interest in a functional approach to parallelization [DG04, DG08]. The same idea could also be implemented for metadata access. For example, traditional iterations over all files in a directory could use a pattern like the meta-program in Listing 5.1.[†] This iteration is very simple to understand, but effectively makes parallelizing very difficult by requiring the programmer to decide when to start and stop parallel processes and how many to use.

[†] This simple technique is also used in DMetabench.

Listing 5.1: Simple sequential directory iteration

```
dir = opendir ("/a/directory")
while dir_entry = dir.getNext() {
        do_operation_on_file( dir_entry )
}
```

Listing 5.2: Functional-style `map` operation on a directory

```
dir_map ( do_operation_on_file, "/a/directory" )
```

A parallelizable functional approach uses the higher-order `map` function, which is available in many functional programming languages (e.g., Lisp, Erlang but also Python). `Map` takes two parameters: a function and a list, and then applies the function to every element of the list, returning a list of results. A similar function called `dir_map` that applies the function to every file in a directory could be written. Using `map`, the explicit iteration process can be avoided (see Listing 5.2).

In this case, which is typical for many administrative metadata-intensive workloads, it is trivial to replace the iterative style with a functional style. The major advantage in doing so is that the programmer only has to specify the operation itself and does not have to consider all the details of distributing work to multiple processes. This functionality can be encapsulated inside `dir_map`.

There are several options for implementing `dir_map`. First, a simple sequential function that mirrors the behaviour of the iterative approach could be used. Alternately, the list of files in a directory can be divided and fed to a fixed or variable number of processes on one or multiple nodes that then operate in parallel. At this point, the relation to metadata benchmarking becomes apparent: The `dir_map` implementation could decide *at runtime* how to effectively divide work by using the (known) scaling properties of the particular file system obtained with DMetabench.

Several examples of the planning criteria are shown in Table 5.1. Performance charateristics of a file system obtained with a benchmark such as DMetabench could deliver data helpful in choosing an efficient manner of parallelization. The main point is that the complexity remains largely hidden from the viewpoint of the programmer, who only cares about the task itself. If the design of a particular file system is known, there are many heuristics that can be applied by the runtime system. In contrast to the program itself, the runtime system could automatically obtain information otherwise not available to user programs via a standard API, such as the hierarchy of volumes in a single-namespace file system.

Table 5.1: Sample criteria for parallelization planning

Operation
Recursive
Modifying/non-modifying
Acceptable age of data
Directory
Length of the directory path
Size and number of files
Existence of subdirectories
Type of subdirectories (mountpoints, global namespace volumes, ...)
Filesystem
Type and performance characteristics of the file system
Connectivity to the file system
Availability of additional client nodes
Data placement
Size and capacity of volumes
Existence and location of replicas
Node
Number of available processors
Current load

5.3.2 Inherently parallel metadata operations

The parallalization method described above does not change the way in which the operating system interacts with a distributed file system. This means that, according to the POSIX standards, every file is handled on its own and there is no way to work on multiple files or directories using a single operation.

An operation model involving multiple files at the same time could also be considered in analogy to *single instruction multiple data (SIMD)* processing on vector processors. The `READDIRPLUS` operation in NFS Version 3 is an existing example of this idea. The general sematics and the programming model for file access in POSIX is, however, unintentionally designed around the idea of independence among metadata operations, and cannot be easily mapped to compound operations. For example, if we imagine an `unlink()` function that deletes a list of files, semantics became complex. It would be difficult to determine the status of the operation if one of the files could not be deleted or whether it would be possible to undo the entire operation using some kind of transactional semantics. Efficient employment of compound operations requires an adequate mapping at the API level to make the semantics transparent.

A working group at Carnegie Mellon University is researching the means by which to make the extension of the POSIX I/O API "more friendly towards HPC, clustering, paral-

lelism, and high concurrency applications" [POS08]. The group has proposed "bulk metadata operations" or alternative semantics for accessing multi-tiered HSM filesystems. There has already been work in this area, which is reflected in current file systems (e.g., the accelerated backup used in GPFS [CMM+03]), but as of the present, there is no efficient standard bulk interface to file system metadata.

5.4 Load control and quality of service

Due to the increased degree of parallelism in metadata operations in a typical multi-user usage scenario, the file system must respond to high load gracefully and in a manner, that does not lead to reduced performance. Another area of concern is access fairness: How are different operations scheduled and what kind of criteria is used? It is clearly undesirable that a single user or group of users monopolizes the entire metadata processing capability of a file system. Filesystems with Quality-of-service (QoS) technology, for example XFS with its *guaranteed rate I/O*, have already been developed but their guarantees are only functional for *data* transfers. As previously discussed, in large parallel filesystems metadata performance could be a much more scarce resource than data throughput.

Finding suitable criteria and metrics for metadata QoS in a file system is an interesting research area. The example described in Chapter 4 has shown that existing priority schedulers are not truly effective for metadata-intensive operations. The spread of service-level agreements (SLAs) based on best-practice IT management methodologies, such as ITIL, creates a need for additional criteria and granularity, such as per-user or per-file, to be able to guarantee SLAs for storage access.

While setting relative performance levels such as "high" or "low" is useful, using absolute numbers would better help virtualize file system resources. For example, in an automatic book scanning system with multiple scanners, that generate an image file for every page, a "metadata performance quota" could be allocated to guarantee that, for example, 2,000 files could be created per minute during operations. Currently, separate hardware systems are often used to guarantee performance levels for storage, but over-provisioning and waste of resources are undesirable in power-constrained data centers.

In this context, a separation of administrative and user access also appears necessary. While an administrator might be interested in using parallelized tools for data management (e.g., performing backups), the impact of this process on user operations should be adjustable and enforced by the distributed file system.

In this context, the DMetabench framework could be a useful tool in assessing the manner in which a QoS system controls metadata performance.

5.5 Future research at LRZ

LRZs next supercomputing environment scheduled for 2011, the HLRB3, is currently being planned. With a performance of several Petaflops, the system will be more than two orders of magnitude faster than the HLRB2 and will present a multitude of new challenges regarding its I/O subsystems. In contrast to the current machine, the future system will probably integrate few large SMP machines with thousands of smaller SMP nodes to match varying requirements of different types of applications. Central file systems must then cope with intra-node and inter-node parallel access and large application checkpoints from both types of systems. It is expected that European GRID infrastructure projects will contribute a larger share of applications to the workload mix. This requires increased focus on pan-european data exchange and suitable file system access. The DMetabench framework will help to evaluate metadata performance of candidate file systems for the HLRB3 under the conditions described above. Additionally, special plugins for testing the performance of HSM subsystems may become necessary.

5.6 Summary

The thesis contributed to the scientific knowledge base on distributed file systems. It began by providing a structured overview of the current state of metadata storage and its protocols and semantics. It then presented existing metadata benchmarks before proposing significant improvements to them, including controlled intra- and inter-node parallelization and time-based logging. The result of these proposals was DMetabench, a distributed benchmark framework that implements these improvements. The thesis then sampled the current state of metadata performance on the basis of measurements taken in multiple large production file system environments. Finally, it summarized the results before presenting both promising performance improvement techniques for current file systems and proposing future areas of research.

Bibliography

[ABDL07] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the 5th Conference on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[ABFPB64] G. M. Amdahl, G. A. Blaauw, and Jr. F. P. Brooks. Architecture of the IBM System/360. *IBM Journal of research and development*, 8, 01/02 1964.

[And02] Darell Anderson. Fstress: A Flexible Network File Service Benchmark. Technical report, Department of Computer Science, Duke University, 2002.

[Ber00] D. J. Bernstein. Using maildir format. `http://cr.yp.to/proto/maildir.html`, 2000.

[Bha06] Akshay Bhargava. FlexShare Design and Implementation Guide. Technical Report 3459, Network Appliance, Inc., 2006.

[Cam98] Richard Campbell. *Managing AFS: the Andrew File System*. Prentice-Hall, 1998.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of OSDI'06: Seventh Symposium on Operating System Design and Implementation*, 2006.

[CFS07] CFS. *Lustre 1.6 Operations Manual*. Cluster File Systems, Inc., 2007.

[CIRT00] Philip H. Carns, Walter B. Ligon III, Robert B. Ross Rajeev, and Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, 2000.

[Cla66] W. A. Clark. The functional structure of OS/360, Part III Data management. *IBM Systems Journal*, 5(1):30–51, 1966.

[Cla03] Tom Clark. *Designing Storage Area Networks: A Practical Reference for Implementing Fibre Channel and IP SANs*. Addison-Wesley Professional, 2003.

[CMM+03] Robert J. Curran, Daniel L. McNabb, Demetrios K. Michalaros, Wayne A. Sawdon, Frank B. Schmuck, and James C. Wyllie. Parallel high speed backup for a storage area network (SAN) file system. US Patent 7092976, June 2003.

[CPS95] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. `http://www.ietf.org/rfc/rfc1813.txt`, June 1995.

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. *Proceedings of the 2004 USENIX Annual Technical Conference*, 2004.

[CTP+05] M. Cao, T. Ts'o, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Thomas. State of the Art: Where we are with the Ext3 filesystem. In *Proceedings of the 2005 Ottawa Linux Symposium*, 2005.

[CTT04] R. Card, T. Ts'o, and S. Tweedie. Design and Implementation of the Second Extended File System. In *Proceedings of the First Dutch International Symposium on Linux*, number ISBN 90-367-0385-9, 2004.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceesing of OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *COMMUNICATIONS OF THE ACM*, 51(1):107, 2008.

[DM02] I. Dowse and D. Malone. Recent Filesystem Optimisations on FreeBSD, 2002.

[DMJB98] Hitz D., Malcolm M., Lau J., and Rakizis B. Method for maintaining consistent states of a file system and for creating user-accessible read-only copies of a file system, June 1998.

[DN65] R.C. Daley and P.G. Neumann. A General-Purpose File System For Secondary Storage. In *AFIPS Conference Proceedings*, volume 27, pages 213–229, 1965.

[ECK+07] Michael Eisler, Peter Corbett, Michael Kazar, Daniel S. Nydick, and J. Christopher Wagner. Data ONTAP GX: A Scalable Storage Cluster. In *Proceedings of FAST' 07*, 2007.

[For03] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, Message Passing Interface Forum, 2003.

[Fri02] Aeleen Frisch. *Essential System Administration*. O'Reilly, 3rd edition, 2002.

[Fuh07] Patrick Fuhrmann. dCache, the Overview. Technical report, Deutsches Elektronen Synchrotron, 2007.

[GK97] Gregory R. Ganger and M. Frans Kaashoek. Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files. In *USENIX Annual Technical Conference*, pages 1–17, 1997.

[GNU08] GNUPlot. `http://www.gnuplot.info/`, 2008.

[Gra81] Jim Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of Seventh International Conference on Very Large Databases*, 1981.

[Gro04] The Open Group. The Single UNIX Specification, Version 3. Technical report, The Open Group, 2004.

[Gro06] Ed Grochowski. Hard Disk Drive Organization Announces a New Sector Length Standard. Press Release, March 2006.

[Grü03] Andreas Grünbacher. POSIX Access Control Lists on Linux. In *Proceesdings of the USENIX Annual Technical Conference, San Antonio, Texas*, 2003.

[HKM+88] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6:51–81, 1988.

[HLM02] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. Technical report, Network Appliance, TR 3002.

[Huf02] John L. Hufferd. *iSCSI: The Universal Storage Connection*. Addison-Wesley Professional, 2002.

[Kat97] J. Katcher. PostMark: A new file system benchmark. Technical report 3022, Network Appliance, 1997.

[Kaz88] Michael Kazar. Synchronization and Caching Issues in the Andrew File System. Technical report, Carnegie-Mellon University, ITC, 1988.

[KL07] Kunkel and Ludwig. Performance Evaluation of the PVFS2 Architecture. *pdp*, 00:509–516, 2007.

[Kle86] Steve R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *USENIX Summer*, pages 238–247, 1986.

[Kow78] T. Kowalski. FSCK - The UNIX System Check Program. Technical report, Bell Laboratory, Murray Hill, NJ 07974, 1978.

[KPCE06] Andy C. Kahn, Kayuri Patel, Raymond C. Chen, and John K. Edwards. File folding technique. Us patent, US Patent Office, 2006.

[Krz03] Martin Krzywinski. Clusterpunch - a distributed mini-benchmark system for clusters, 2003.

[LR07a] Leibniz-Rechenzentrum. Höchstleistungsrechner in Bayern. `http://www.lrz-muenchen.de/services/compute/hlrb/`, 2007.

[LR07b] Leibniz-Rechenzentrum. Statistik zu TSM. http://www.lrz-muenchen.de/services/datenhaltung/adsm/statistik/, 2007.

[LRT05] Rob Latham, Rob Ross, and Rajeev Thakur. The Impact of File Systems on MPI-IO Scalability. Technical report, Argonne National Laboratory, 2005.

[MB04] Lars Marowski-Brée. A new Cluster Resource Manager for heartbeat. Technical report, UK's Unix and Open Systems User Group, 2004.

[MG99] Marshall Kirk McKusick and Gregory R. Ganger. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*. The USENIX Association, 1999.

[Mic05] Microsoft. Overview of the Distributed File System Solution in Microsoft Windows Server 2003R2. Technical report, Microsoft Corporation, 2005.

[Mic06] Microsoft. Single Instance Storage in Microsoft Windows Storage Server 2003 R2. Technical white paper, Microsoft, 2006.

[Mit00] Roman Mitz. The Rx protocol. `http://rmitz.org/rx/Rx.pdf`, 2000.

[MJLF84] M. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. Technical report, Computer Systems Research Group, University of California, Berkeley, 1984.

[MM06] Richard McDougall and Jim Mauro. Filebench tutorial. `http://www.solarisinternals.com/si/tools/filebench`, 2006.

[MS96] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. Technical report, Silicon Graphics, Inc., 1996.

[NC06] William D. Norcott and Don Capps. Iozone Filesystem Benchmark. `http://www.iozone.org/`, 2006.

[Nie07] Ole Nielsen. Pypar - parallel programming with Python. `http://sourceforge.net/projects/pypar`, July 2007.

[Ora06] Oracle. Oracle Cluster File System (OCFS2) User's Guide. `http://oss.oracle.com/projects/ocfs2/dist/documentation/ocfs2\_users\_guide.pdf`, 2006.

[Ora07a] Oracle. *Oracle Database 11g Direct NFS Client*, July 2007.

[Ora07b] Oracle. *SecureFiles and Large Objects Developer's Guide 11g Release 1 (11.1)*, b28393-02 edition, October 2007.

[Pan07] Panasas. *Object Storage Architecture Whitepaper*. Panasas, Inc., 2007.

[Pat06] Mikulas Patocka. *Design and Implementation of the Spad Filesystem*. PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Software Engineering, 2006.

[Phi01] Daniel Phillips. A directory index for ext2. In *5th Annual Linux Showcase and Conference*, pages 173–182, 2001.

[Pil90] Mark Pilgrim. *Dive Into Python*. APress, 1990.

[Plo08] Ploticus. `http://ploticus.sourceforge.net/`, 2008.

[Pol08] Evgeniy Polyakov. POHMELFS - Parallel Optimized Host Message Exchange Layered File System. `http://tservice.net.ru/~s0mbre/`, April 2008.

[POS08] POSIX HPC Extensions / CMU. POSIX HPC Extensions Goals. `http://www.pdl.cmu.edu/posix/docs/POSIX-extensions-goals.pdf`, 2008.

[RD03] Guido Rossum and Fred L. Jr. Drake. *An Introduction to Python*. Network Theory Ltd., 2003.

[Reu07] Hartmut Reuter. Langzeitarchivierung von 1988 bis jetzt am RZG. Vortrag beim babs-workshop, bsb-münchen, Rechenzentrum Garching, Max-Planck-Gesellschaft, 2007.

[RHR+94] Peter L. Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving File Conflicts in the Ficus File System. In *USENIX Summer*, pages 183–195, 1994.

[RKPH01] Rolf Rabenseifner, Alice E. Koniges, Jean-Pierre Prost, and Richard Hedges. The Parallel Effective I/O Bandwidth Benchmark: b_eff_io. Technical report, High-Performance Computing Center (HLRS), 2001.

[RO92] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[RT78] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *The Bell System Technical Journal*, 57(6 (part 2)):1905+, 1978.

[SA05] Gil Sever and Noah Amit. Method for transparent real time compression of a file system data over file systems protocols with optional independent and transparent real time encryption. Us provisional patent application, US Patent Office, 2005.

[Sat06] Mahadev Satyanarayanan. The Coda File System. `http://www.coda.cs.cmu.edu/`, 2006.

[SDH+96] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Technical Conference*, pages 1–14, San Diego, CA, USA, 22–26 1996.

[SE06] Laure Shepard and Eric Eppe. SGI InfiniteStorage Shares Filesystem CXFS: A High-Performance, Multi-OS Filesystem from SGI. Technical report, Silicon Graphics, 2006.

[SFHV99] Douglas J. Santry, Michael J. Feeley, Norman C. Hutchinson, and Alistair C. Veitch. Elephant: The File System That Never Forgets. In *Workshop on Hot Topics in Operating Systems*, pages 2–7, 1999.

[SGI07a] SGI. FAM Overview. `http://oss.sgi.com/projects/fam/`, 2007.

[SGI07b] SGI. XFS overview and internals - 06 allocators. `http://oss.sgi.com/projects/xfs/training/xfs\_slides\_06\_allocators.pdf`, 2007.

[SGM+00] Margo Seltzer, Greg Ganger, M. Kirk McKusick, Keith Smith, Craig Soules, and Christopher Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *USENIX Annual Technical Conference*, pages 18–23, June 2000.

[SGSG03] C. Soules, G. Goodson, J. Strunk, and G. Ganger. Metadata efficiency in versioning file systems, 2003.

[SH02] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, January 2002.

[Sil06] Silicon Graphics, Inc. SGI Altix 4700 Servers and Supercomputers Data Sheet. `http://www.sgi.com/pdfs/3867.pdf`, 2006.

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of VLDB '07*, 2007.

[SNI02] SNIA. Common Internet File System Technical Reference. Technical report, Storage Networking Industry Association, 2002.

[SPE08] SPEC. SPECsfs 2008 User's Guide. Technical Report Version 1.0, Standard Performance Evaluation Corporation (SPEC), 2008.

[Str04] Oliver Strutynski. Design and Implementation of a benchmark for parallel file systems. Technical report, Technische Universität München, 2004.

[Str07] Stress. `http://weather.ou.edu/~apw/projects/stress/`, 2007.

[Sun05] Sun. ZFS On-Disk Specification. Technical report, Sun Microsystems, 2005.

[SvIG06] Russell Sears, Catharine van Ingen, and Jim Gray. To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem? Technical report, Microsoft Research, 2006.

[Tan95] Diane Tang. Benchmarking Filesystems. Technical Report TR-19-95, Harvard University, Cambridge, MA, USA, October 1995.

[TEFK95] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications. In *Proceedings of the The 19th ACM International Conference on Supercomputing*, 1995.

[TT02] Theodore Ts'o and Stephen Tweedie. Planned Extensions to the Linux Ext2/Ext3 Filesystem. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, 2002.

[Tur06] Jeff Turner. File Creation Latency (Solaris 10 OS). `http://www.middleworld.com/docs/latency1.pdf`, 2006.

[WdW03] Parkson Wong and Rob F. Van der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical report, NASA Advanced Supercomputing (NAS) Division, 2003.

[WK93] M. Wittle and Bruce E. Keith. LADDIS: The Next Generation in NFS Server Benchmarking. In *USENIX Summer*, pages 111–128, 1993.

[ZCCE03] N. Zhu, J. Chen, T. Chiueh, and D. Ellard. An NFS Trace Player for File System Evaluation, 2003.

List of Figures

List of Tables