Simulation-Aided Performance Evaluation of Input/Output Optimizations for Distributed Systems

— Masterarbeit —

Parallele und Verteilte Systeme
Institut für Informatik
Ruprecht-Karls-Universität Heidelberg

Bearbeiter:	Michael Kuhn
E-Mail-Adresse:	michael.kuhn@stud.uni-heidelberg.de
Matrikelnummer:	2405219
Aufgabensteller:	Prof. Dr. Thomas Ludwig
Betreuer:	Julian Kunkel
Abgabedatum:	30. September 2009

Ich versichere, dass ich diese Masterarbeit selbstständig verfasst, nur die angegebenen Quellen und Hilfsmittel verwendet und die Grundsätze und Empfehlungen „Verantwortung in der Wissenschaft“ der Universität Heidelberg beachtet habe.

30. September 2009, ...

Abstract

The performance of parallel cluster file systems suffers from many clients executing a large number of operations in parallel, because the I/O subsystem can be easily overwhelmed by the sheer amount of incoming I/O operations. This, in turn, can slow down the whole distributed system.

Many optimizations exist that try to alleviate this problem. Client-side optimizations do preprocessing to minimize the amount of work the file servers have to do. Server-side optimizations use server-internal knowledge to improve performance.

The PIOsimHD framework contains components to simulate, trace and visualize applications. It is used as a testbed to implement optimizations that could later be implemented in real-life projects.

The main focus of this thesis lies on comparing existing client-side optimizations and newly implemented server-side optimizations like Server-Directed I/O, which provides server-side optimizations for both read and write operations. It chooses the order of I/O operations and tries to aggregate as many operations as possible to decrease the load on the I/O subsystem and improve overall performance. The Interleaved Two-Phase protocol is a modification of ROMIO's Two-Phase protocol, which only accesses contiguous file regions.

HDSunshot is used to visualize and analyze some of the results. It is also used to evaluate different optimization techniques by analyzing the resulting traces.

The results show that client-side optimizations do not necessarily beat server-side optimizations in terms of performance, but suggest that even simple server-side optimizations are good enough for many use cases. Integrating such optimizations into parallel cluster file systems could alleviate the need for sophisticated client-side optimizations. Due to their additional knowledge of internal workflows server-side optimizations may be better suited to provide high performance in general.

Acknowledgements

I thank Thomas Ludwig for his guidance and support on many occasions. Special thanks go to Julian Kunkel for providing me with reference material, promptly fixing bugs in PIOsimHD and — most importantly — proofreading the thesis. I also want to thank Manuela Ernst and Michael Neumann for some last-minute corrections.

Contents

Chapter 1

Introduction

In this chapter definitions of parallel cluster file systems and non-contiguous I/O[1] *are given. A special focus lies on the problems these file systems are facing when accessed by large numbers of clients, because the goal of this thesis is to examine possible solutions for these problems. Additionally, storage devices providing the underlying storage for parallel cluster file systems and the simulator software PIOsimHD are introduced.*

1.1 Parallel Cluster File Systems

Cluster file systems are built in such a way that several machines connected by some network make up one file system. Usually every machine is either a data server or a metadata server, that is, it only stores objects of the respective type. It is also possible but unusual that each machine acts as a combined data and metadata server.

Every machine has only direct access to its own storage, so no machine has a total overview of the complete file system. Consequently, either the servers have to communicate with each other or clients must contact multiple servers. In any event data has to be sent across the network, thus introducing an additional delay between the request and response. For Ethernet networks this latency usually lies between 50 µs and 500 µs.

If a client wants to read a file, it first has to figure out on which metadata servers the metadata about this file can be found. Then it must request the data from each data server that holds data of this particular file. In addition to the disk access times of the servers the client has to connect via the network to all those servers. This is expensive — especially for small requests — due to network latency and bandwidth restrictions.

Cluster file systems can be accessed by multiple remote clients simultaneously. Parallel cluster file systems additionally allow clients to access the same file

[1] Input/Output

concurrently. To do this, some form of concurrency control must be in place. This, however, must not be confused with ordinary locking, because it is done above the file system layer. Locking can be used to protect a file against concurrent access, but not the file system itself. Therefore, a mechanism must be implemented at the file system level to keep the clients from corrupting the file system by modifying the same part of it at the same time.

Parallel cluster file systems are designed to handle a large number of clients and deliver high performance. However, due to their distributed design most operations are expensive to perform, which in turn can seriously impact overall performance. For this reason new algorithms to efficiently perform I/O on these file system have emerged and a multitude of optimizations are available to improve performance even further.

Figure 1.1 shows four clients accessing different blocks of data in a file. All 12 blocks of the file are read, each client reading three blocks of data. However, one client's blocks are spread across the file. Depending on the order of the clients' requests, the file server may need to perform a lot of seeks within the file. This overhead increases with a growing number of clients, to the point where the file server may be busy doing more seeks than actual I/O. This is one of the main problems found in large-scale systems with a lot of clients accessing small portions of a file.

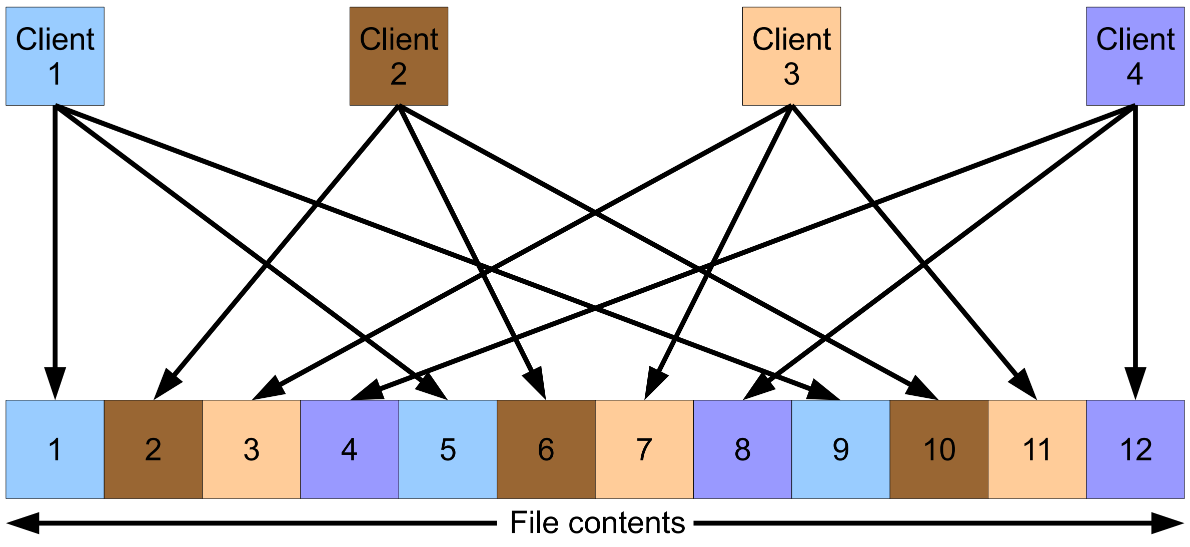


Figure 1.1: Multiple Clients Accessing the Same File

For a more in-depth analysis of problems in parallel cluster file systems and possible solutions, see [Kuh07].

1.2 Storage Devices

Storage devices relevant for parallel cluster file systems can be roughly separated into two categories — mechanical ones and non-mechanical ones. Other differences like connection interfaces (SCSI[2], ATA[3], SATA[4], etc.) and built-in optimizations (Caches, NCQ[5], etc.) are not important in this regard. HDDs[6] represent the mechanical storage devices, while SSDs[7] represent the non-mechanical ones.

Some file system also use SANs[8] as the underlying storage. However, these SANs also use HDDs or SSDs to actually store data.

The influence of their unique characteristics on the optimizations examined in this thesis are described in more detail in section 2.2 (page 16).

1.2.1 Hard Disk Drives

HDDs consist of multiple rotating platters with magnetic surfaces and read-write heads. To access data on the disk the read-write heads have to move and wait for the platters to rotate to the appropriate position. Since moving the read-write heads is a mechanical operation it is relatively slow to perform. In addition, the platters rotate at a constant speed — usually 7.200 or 10.000 RPM[9] —, limiting the minimum latency and maximum throughput. Access times for drive accesses usually lie between 5 ms and 15 ms, depending on the area of the disk that should be accessed. This can cause performance to degrade significantly for random accesses.

Modern HDDs include a random access cache, and operating systems usually have dedicated file system caches to speed up disk operations. Without these caches HDDs would be unusably slow. However, this does not solve the problem where many random read accesses are performed for the first time, that is, where the data is not cached already.

1.2.2 Solid State Drives

In contrast to HDDs, SSDs contain no mechanical parts, but use solid state memory — usually flash memory — in order to store the data. Access times for drive accesses usually lie between 20 µs and 200 µs, depending on the type of operation. This

[2]Small Computer System Interface
[3]Advanced Technology Attachment
[4]Serial Advanced Technology Attachment
[5]Native Command Queuing
[6]Hard Disk Drives
[7]Solid State Drives
[8]Storage Area Networks
[9]Revolutions Per Minute

makes them orders of magnitude faster than HDDs, especially when many random accesses are involved.

SSDs can only write blocks in the form of so-called erase blocks, that is, the old data has to be read, modified in memory, erased and then written back to the solid state memory. Modern SSDs use erase blocks of 512 KiB[10] and more, which can severely impact performance when performing many small write operations. Newer SSDs and OSs[11] support the `TRIM`[12] command to mitigate this problem, but this is not supported everywhere. In contrast, reads can be performed with a granularity of 4 KiB. This causes the read and write performances to differ greatly most of the time. For example, current Intel SSDs can read up to 250 MiB[13]/s, but only write 75 MiB/s.

Due to their very recent appearance in the mass market SSDs are still significantly more expensive and offer less capacity when compared to traditional HDDs. Therefore, SSDs are usually not yet considered when storing large amounts of data, except for workloads involving large amounts of random I/O operations. For example, this includes large database applications.

1.3 Non-Contiguous I/O

Traditionally, data is accessed in contiguous regions, identified by an offset and a size. Non-contiguous I/O enables applications to read several contiguous regions that may be separated by holes of arbitrary size without actually requesting each of the regions separately. This significantly reduces the complexity needed to handle complex data types in a program. For example, the user can construct a data type for a matrix diagonal and use it like any other data type for I/O. Using non-contiguous I/O to perform such operations may give the underlying file system — or another layer between the application and the file system — the opportunity to optimize these kinds of operations by exploiting the fact that several seemingly independent operations are in fact part of one non-contiguous operation. On the other hand, this can increase the complexity needed to do I/O and may require additional optimizations to handle such I/O efficiently. Some simple optimizations are presented in section 2.2 (page 16).

Figure 1.2 (page 12) shows a file consisting of 12 blocks and one client accessing the file. The client uses one non-contiguous I/O operation to read eight blocks in three contiguous regions (1–3, 6–7 and 10–12).

[10]Kibi Byte
[11]Operating Systems
[12]For more information see `http://en.wikipedia.org/wiki/TRIM_(SSD_command)`.
[13]Mebi Byte

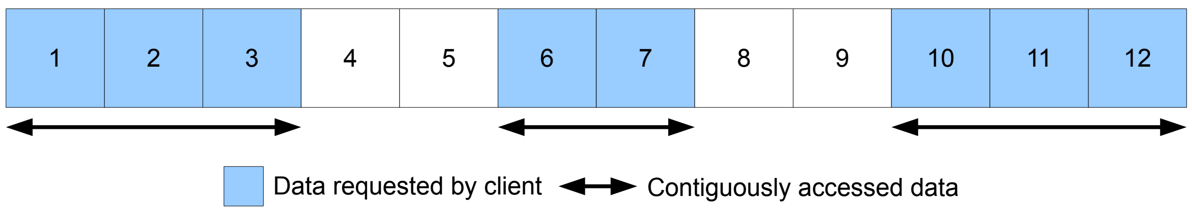


Figure 1.2: Non-Contiguous I/O

1.4 PIOsimHD

PIOsimHD[14] is a framework containing all components necessary to simulate, trace and visualize applications. One of its goals is to allow easy and fast prototyping of new algorithms for I/O optimization. Therefore, all optimizations presented in this thesis are implemented in PIOsimHD-Simulator to evaluate their theoretical benefits.

These optimizations are not dependent on any specific project environment and can serve as a starting point for adoption of promising optimizations into real-life projects. This helps avoiding the overhead of implementing several complex optimization variants in even more complex real-life projects when only the most promising one is really needed.

A more detailed description of PIOsimHD can be found in section 3.1 (page 21).

1.5 Goals of the Thesis

The main goal of this thesis is to determine whether server-side optimizations are sufficient to provide acceptable I/O performance for common workloads. Traditionally, client-side I/O libraries provide very sophisticated optimizations to improve performance in distributed systems. However, it is not clear whether this complexity is really needed or if simpler server-side optimizations can provide similar results for many cases.

1.6 Outlook

Chapter 2 (page 14) introduces related work and basic concepts that form the basis of the optimizations done in this thesis. Chapter 3 (page 21) presents the software used to implement and evaluate these optimizations. This chapter also gives an overview of the internal structure of the simulator software PIOsimHD. The

[14]Parallel I/O Simulator

basic design of the optimizations is elaborated in chapter 4 (page 30). Chapter 5 (page 37) focuses on the actual implementation of these optimizations. Finally, the benefits of these changes are examined in detail. In chapter 6 (page 61) the actual impact on performance in terms of throughput is evaluated. Chapter 7 (page 88) focuses on the visualization of these results. Chapter 8 (page 95) summarizes and concludes the thesis. Additionally, ideas for future work are proposed.

Summary

The performance of parallel cluster file systems suffers from many clients executing a large number of operations in parallel. Many optimizations exist that try to alleviate this problem. These optimizations should be suitable for both HDDs and SSDs, because HDDs are still used for large-scale data storage, but SSDs play an increasingly important role. Non-contiguous I/O can be used as a basis for a multitude of optimizations by providing the file system with additional information. PIOsimHD is used as a testbed to implement optimizations that could later be implemented in real-life projects. The main focus lies on comparing existing client-side optimizations and newly implemented server-side optimizations.

Chapter 2

Related Work and State of the Art

In this chapter related optimization techniques are presented and the difference between client-side and server-side optimizations is explained. Additionally, simpler optimizations like data sieving, collective I/O and the Two-Phase protocol are explained in more detail. Lastly, the unique characteristics of the storage devices already mentioned in chapter 1 (page 8) are considered with regard to the presented optimizations.

2.1 Related Work

A multitude of approaches exist to optimize I/O in distributed systems. They can be basically classified into two categories. Some approaches focus heavily on the client, trying to minimize the work the servers have to do. For example, the clients' RAM[1] can be used to cache and batch I/O operations to reduce the load on the servers. Other approaches do not do preprocessing on the clients and let the servers handle all the work themselves. The servers can then employ their own optimizations.

Simpler client-side optimizations like data sieving and collective I/O are presented in [TGL99], which focuses on their implementation in ROMIO. These optimizations are described in more detail in section 2.2 (page 16).

In [SIC$^+$07, SIPC09] a refined and extended version of the Two-Phase protocol called Multiple-Phase Collective I/O is presented. The communication phase is split up in several steps, in which pairs of clients communicate with each other in parallel. These multiple steps are used to progressively increase the locality of the data. This increases the performance of the subsequent I/O phase and therefore of the whole application.

File accesses in distributed applications are often non-contiguous. In [KRVP07] a method to optimize non-contiguous write operations using log-based storage[2] is

[1]Random Access Memory

[2]See `http://en.wikipedia.org/wiki/Log-structured_file_system`.

presented. This log-based storage always appends new data at the end of the log and thus converts non-contiguous write operations into contiguous ones. Special care must be taken when reading the file, because the log has to be replayed to create a traditional file for reading. Therefore, this optimization is especially useful for write-once or write-mostly cases like log files and snapshots.

Another popular client-side optimization is to use a distributed file cache as presented in [PTH$^+$01, LCCW05, LCC$^+$07, TLC$^+$99]. While [TLC$^+$99] uses a dedicated cache that is accessed via a so-called Distributed-Parallel Storage System master, [LCCW05, LCC$^+$07] describes a cache built into the MPI-IO[3] library which uses the main memory of all participating clients as a large two-staged write-behind cache. The method in [PTH$^+$01] is geared towards the IBM's GPFS[4] and also uses a cache managed by the participating clients. To avoid concurrent access, each GPFS file block is managed by exactly one client and all access to this block is done via this client. Double buffering is used to improve performance by overlapping communication and I/O.

Parallel cluster file systems usually only offer one consistency policy. In [VNS05] a new file system is presented that allows users to specify their applications' consistency needs. For example, temporary files can be stored locally or locking can be disabled when parallel access is ruled out. This allows users to tune the file system itself to deliver maximum performance for their specific workloads.

A technique called super-scalar compression is presented in [Zuk05]. The main idea is that the CPU[5] only uses part of its processing power when waiting for I/O to complete. Spare CPU power can be used to compress and decompress the file data on demand. When using ultra lightweight compression, this can result in a significant speedup.

In [Kot97] an alternative to client-side optimizations — in particular, the Two-Phase protocol — is presented. Disk-directed I/O is used to optimize the data flow on the file server itself, that is, optimizations are no longer done by the clients. This technique has the benefit that the file servers can use information about their physical disk layout to optimize accesses and also avoids the communication and computation overhead of the clients caused by the Two-Phase protocol.

The above optimizations only focus on the file data itself. However, metadata performance also plays an important role in parallel cluster file systems. In [DW07, WPBM04, BMLX03, KKL08, KKL09] approaches to speed up metadata-heavy operations or for efficient management of file metadata are presented. Even though the focus of this thesis is pure data performance, this aspect of overall performance should be noted for completeness.

[3]Message Passing Interface Input/Output
[4]General Parallel File System
[5]Central Processing Unit

In [BDK97] the authors present several components of a simulator framework that is capable of parallel simulation of MPI-IO applications and a parallel file system. In contrast to this design, PIOsimHD does not support parallel simulation, but allows modeling and simulation of all involved components of the distributed system.

2.2 State of the Art

Some optimizations are of special interest and are therefore presented in detail.

2.2.1 Data Sieving

Data sieving is used to optimize non-contiguous I/O. Instead of reading multiple small blocks of data, several such blocks are grouped together and then read as a larger block. On the one hand this avoids unnecessary seeking and can improve performance, while on the other hand this may cause unneeded data to be read and subsequently discarded. Obviously, this is only beneficial if the amount of unneeded data is relatively small.

Figure 2.1 shows one client accessing three contiguous blocks of data (1–3, 6–7 and 10–12) in a file. The data sieving algorithm may now read just one contiguous block of data (1–12), discard the unnecessary parts (4–5 and 8–9) and return the requested blocks to the application.

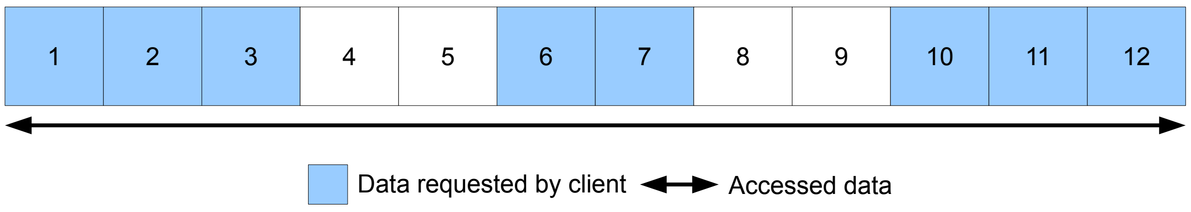


Figure 2.1: Data Sieving

This optimization also relies heavily on the fact that contiguous regions of a file are stored contiguously on disk. If the file is fragmented, this may cause performance degradation. It is also possible to use this technique for write operations. However, this may require read-modify-write operations to be performed, which further reduces the potential benefits. For example, this can happen when the clients do not write the whole contiguous region covered by the data sieving algorithm.

2.2.2 Collective I/O

Collective I/O can be used to explicitly relate I/O operations performed by multiple clients with each other. For example, when using individual I/O the operations performed by one client may have already started or even finished by the time the second client's operations are received by the file system. When using collective I/O all operations are available at the same time, allowing the file system to perform optimizations that are impossible to do with individual I/O.

For example, the additional information provided by collective I/O can be used to optimize non-contiguous I/O very well. Multiple non-contiguous accesses can be combined to improve performance, as demonstrated below.

Figure 2.2 shows two clients each accessing three contiguous blocks (1, 3–4, and 7–9 and 2, 5–6, and 10–12 respectively) of data in a file. When performing individual I/O the file system may now first serve client 1, followed by client 2. This may cause inefficient execution of both requests — for example, by unnecessarily seeking within the file or by accessing the same set of data twice when using data sieving. When using collective I/O the file system can recognize that both clients' requests access the whole file and use that knowledge to optimize the operation by merging all requests into one larger request.

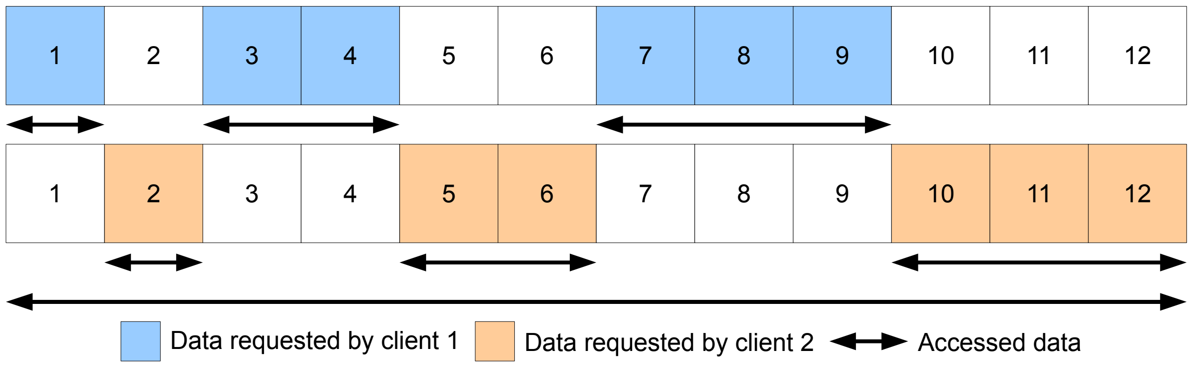


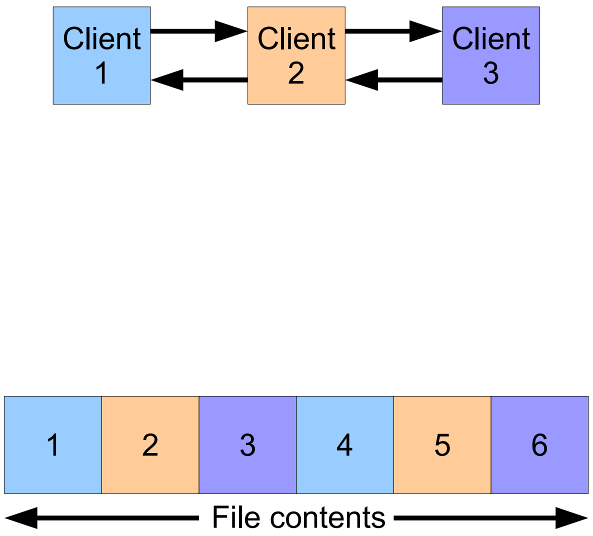
Figure 2.2: Collective I/O

2.2.3 Two-Phase

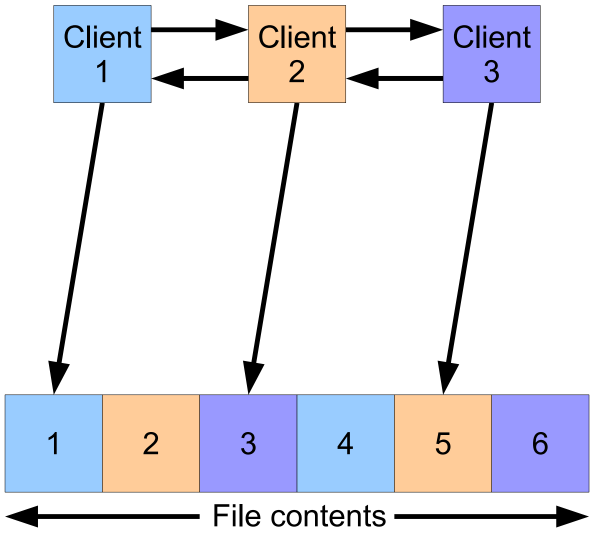
The Two-Phase protocol is an optimization for collective I/O implemented in ROMIO. It uses separate communication and I/O phases to optimize I/O accesses of multiple clients, but introduces additional communication overhead. The basic idea is to let the clients do the actual work of optimizing the accesses to ease the load on the file system. While the communication phase is used to share information necessary for meaningful optimizations among the clients, the I/O phase is used to perform the actual I/O as determined in the communication phase. The Two-Phase protocol implemented in ROMIO mainly focuses on making the accesses contiguous.

Figures 2.3a (page 18), 2.3b (page 18) and 2.3c (page 18) show a typical execution of the Two-Phase protocol. Each client wants to access two data blocks spread across the file. The color of the data blocks corresponds to the color of the accessing client. For example, client 1 wants to access data blocks 1 and 3, client 2 wants to access data blocks 2 and 5, and client 3 wants to access data blocks 3 and 6.

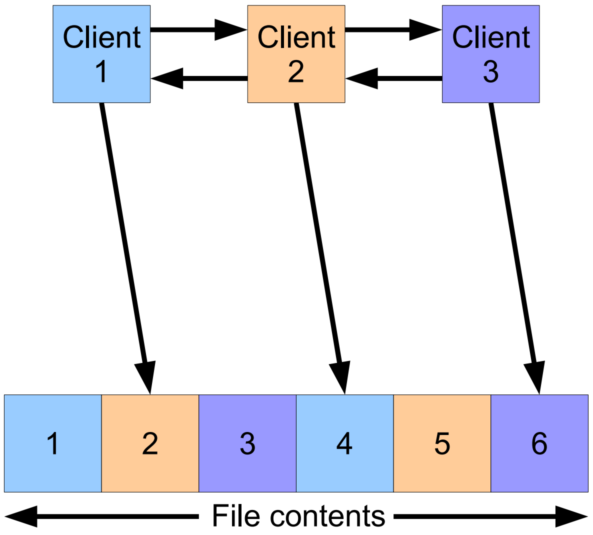
It is important to note that each client communicates with all other clients. For simplicity, these implicit arrows are omitted in the figures.



(a) Phase 1



(b) Phase 2 — Iteration 1



(c) Phase 2 — Iteration 2

Figure 2.3: Two-Phase

Figure 2.3a shows the communication phase of the Two-Phase protocol. All clients negotiate with each other which file areas have to be accessed. According to this information the clients decide whether they should continue to perform the Two-Phase protocol or simply access their respective data blocks individually. For example, ROMIO only uses the Two-Phase protocol when the clients' accesses are overlapping. When performing the Two-Phase protocol they split up the file equally into so-called file domains. Each client is then responsible for one file domain. For example, client 1 is responsible for the file domain consisting of the data blocks 1 and 2, client 2 is responsible for the one consisting of the data blocks 3 and 4, and so on.

Figure 2.3b (page 18) and 2.3c (page 18) show the I/O phase of the Two-Phase protocol. All clients only access data blocks within their respective file domains. When reading, the clients forward the read data to the clients which requested the data in the communication phase. When writing, the clients receive the data from the clients which requested to write the data in the communication phase and write it to the file system. As each client only uses a relatively small amount of main memory as a buffer for the Two-Phase protocol these two steps — communicating with the other clients and accessing the file — can be performed multiple times when needed. For example, figure 2.3b (page 18) shows the first iteration of the I/O phase, while figure 2.3c (page 18) shows the second iteration.

It is easy to see that splitting the file into file domains and distributing these file domains among the clients leads to non-contiguous accesses in each iteration of the I/O phase. For example, figures 2.3b (page 18) and 2.3c (page 18) show that the Two-Phase protocol causes the data blocks 1, 3 and 5 to be accessed in iteration 1, while the data blocks 2, 4 and 6 are accessed in iteration 2. This may have a negative effect on performance, depending on the size of the accesses used internally by the Two-Phase protocol. This problem should be negligible with the Two-Phase protocol's default settings. However, due to striping schemes commonly used in parallel cluster file systems this optimization may be ineffective on these file systems, because accesses must be split up according to the stripe size anyway.

2.3 Storage Devices

All of these optimizations were developed with HDDs as storage devices in mind. Therefore, they may or may not be suited for SSDs. The following tries to evaluate their usefulness for both HDDs and SSDs.

2.3.1 Hard Disk Drives

Due to the relatively slow random access times of HDDs, it may be highly beneficial to merge multiple non-contiguous operations into contiguous ones, even if this involves reading or writing more data than needed. This is due to the fact that the rotating platters are better suited for accessing large contiguous regions of data. Obviously, this applies to both read and write operations.

However, clients can only use the file's logical layout to perform optimizations and logically contiguous accesses are not necessarily contiguous anymore when they are mapped to the file's physical layout on disk. This is due to the fact that most file systems do not — and usually can not — store files as one contiguous region on disk, because free space fragments over time. This may even cause performance degradation, depending on the file system's fragmentation rate. Server-side optimizations do not suffer from this problem, because they have access to the file's physical layout and can use it to make more appropriate decisions.

2.3.2 Solid State Drives

SSDs do not suffer from the problem of slow random accesses. Therefore, it is usually not needed to combine non-contiguous operations when this involves reading or writing additional data.

However, it can be beneficial to combine several operations to use the block sizes used internally by the SSD. For example, performing write operations with the erase block size may save unnecessary read-modify-write overhead. Again, this is more useful when performed on the server, because the file's logical layout usually differs from its physical one and the erase block size may not be available on the clients.

Summary

Many I/O optimizations for distributed systems exist. Client-side optimizations do preprocessing to minimize the amount of work the file servers have to do. Server-side optimizations use the servers' internal knowledge to improve performance. Data sieving can be used to turn non-contiguous I/O into contiguous I/O, while collective I/O provides the file servers with additional information to enable better optimizations. The Two-Phase protocol is an example of a client-side optimization implemented in ROMIO. I/O optimizations should work with both HDDs and SSDs.

Chapter 3

Software Environment

In this chapter an overview of PIOsimHD is given and its components are explained in detail. Its internal architecture is explained with a simple example of a simulated cluster environment. The features of the currently implemented cache layers are explained, because they play an important role in this thesis. Additionally, third-party software used by PIOsimHD is presented briefly.

3.1 PIOsimHD

PIOsimHD is a framework containing all components necessary to simulate, trace and visualize applications. The PIOsimHD-Model and PIOsimHD-Simulator parts allow simulating arbitrary network topologies, servers and client applications. Simple applications can be implemented directly within PIOsimHD-Simulator for fast testing of new algorithms. PIOsimHD also contains modified versions of MPICH2 and PVFS[1] to allow tracing of real MPI programs, which can then also be simulated. HDSunshot can display the resulting traces, allowing a detailed visual performance analysis.

Some of its stated goals are:

- Fast and easy prototyping of new algorithms for I/O and MPI[2] optimization
- Reveal bottlenecks in applications
- Allow better understanding of observed system behavior
- Test applications on different cluster hardware — for example, on grids
- Simple yet extensible models which can be selected on demand
- Explicit abstraction from real world complexity
- Usage of common tools to analyze simulation results

[1]Parallel Virtual File System
[2]Message Passing Interface

3.1.1 Architecture

PIOsimHD-Model and PIOsimHD-Simulator provide the possibility to define arbitrary models. For example, they can be used to model and simulate computer clusters, networks between computers and applications running on them.

Figure 3.1[3] shows the components supported by PIOsimHD-Model and PIOsimHD-Simulator in the form of an exemplary cluster model. The model contains two switches, connecting a total of three nodes. Each switch contains ports which establish connections with the NIC[4] within the nodes. Additionally, switch #1 is connected to switch #2 via an upstream port. Node #1 acts as a server, which in turn contains a cache layer and an I/O subsystem. The cache layer can be used to simulate caches of the operating system and storage devices. The I/O subsystem on the other hand can be used to simulate the actual storage devices. For example, the I/O subsystem could be a HDD or a SSD. Nodes #2 and #3 act as clients, which can host multiple client processes. For example, two client processes of the Jacobi application run on nodes #2 and #3. These two client processes are part of

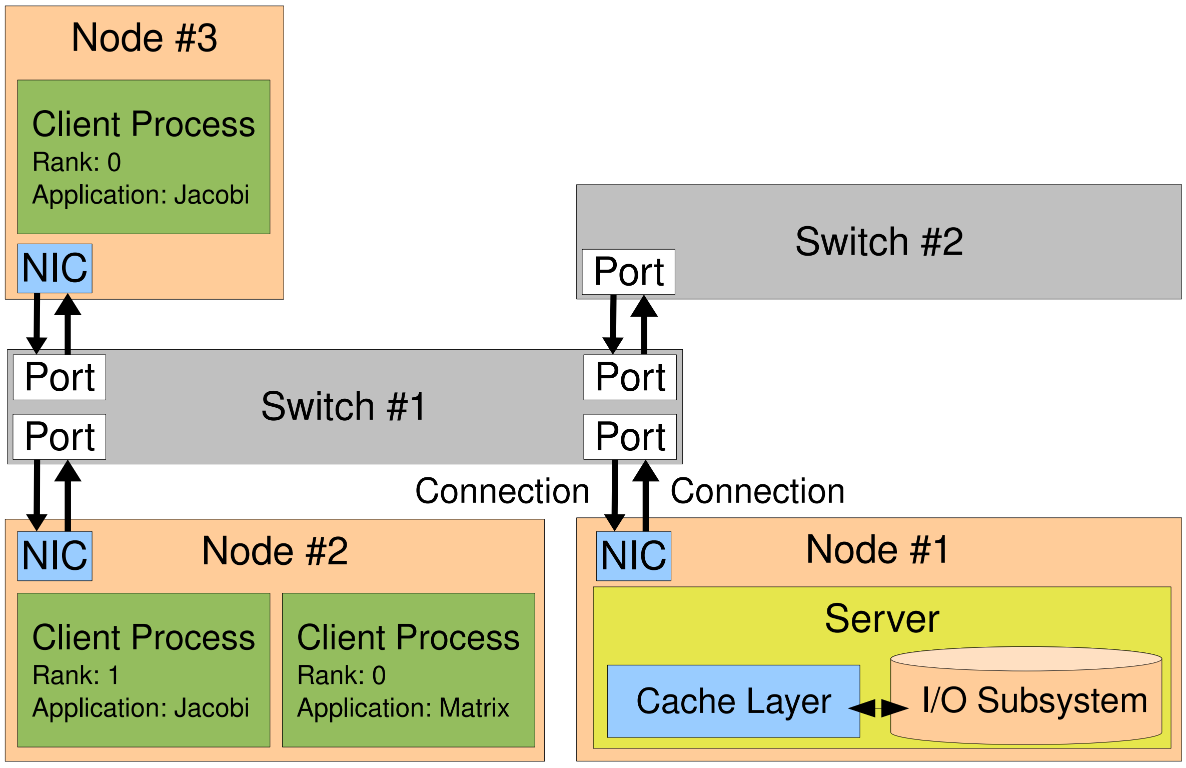


Figure 3.1: Simulator Components

[3]The figure is taken from [Kun] and modified with the author’s permission.
[4]Network Interface Card

the same MPI application and can easily communicate with each other. All clients processes can use node #1's server to store and retrieve data.

The model also allows more complex setups. For example, nodes can be configured to host both client processes and servers. In addition to that, each component can be configured individually. For example, some servers can be configured to use hard disk drives without caches while other servers use solid state drives with elaborate caching mechanisms. Obviously, the same is also true for nodes with client processes.

3.2 PIOsimHD-Model

PIOsimHD-Model allows modelling of arbitrary components involved in the simulation. For example, PIOsimHD-Model contains models for hardware components — like HDDs and NICs — and software components — like individual MPI commands.

Each command and component model can have multiple implementations, which are actually contained in PIOsimHD-Simulator. For example, an MPI command can be implemented in a non-optimized and in an optimized way. This allows users to easily measure the benefits of certain optimizations.

3.3 PIOsimHD-Simulator

PIOsimHD-Simulator is the framework's core component, which performs the actual simulation. It also contains implementations for the models defined in PIOsimHD-Model.

The following three important aspects of PIOsimHD-Simulator are elaborated: The cache layers, how network packets are transported within the network and the internal processing of I/O requests. Most of this information can also be found in [Kun].

3.3.1 Network Packet Transport

The network packet transport uses the following design criteria:

- **Efficiency:** It is mandatory that network components can be fully utilized if there are packets to transport.
- **Realism:** The data flow must be somehow realistic. If there is a bottleneck it is not acceptable that all packets queue on this component. In reality,

switched networks use buffers to allow limited queueing of packets. Retransmits occur on packet loss and timeouts force retransmission of lost packets.

- **Simplicity:** Observed behavior should be understandable to some extent.
- **Performance:** The implementation should not heavily impact simulation performance, because usually a large number of events are simulated.

The current design models flow control as follows:

- A network component transmits a number of packets to another one until enough data is sent to compensate the connection latency. Then the component stalls submission and queues further packets to this particular destination.
- Processing of a packet sends a receipt confirmation to the sender and allows transferring packets worth the connection latency of this packet. Therefore, processing of a packet might wake up a cascade of blocked components upstream.
- Once a component finishes processing a packet it is ready to accept new packets. Pending packets should be delivered immediately to avoid starvation of downstream components.
- Each component queues packets using per-target queues, that is, packets for different destinations are managed in different queues. Whenever one destination is stalled another destination might still be ready to receive packets. For example, assuming a complex network topology with network components interconnected at varying speeds, a slow target might get stalled in the intermediate switches. However, other network components with fast interconnects might still be able to receive data. Therefore, the switches must relay their packets to these faster network components.
- NICs must be allowed to block receiving further packets. For example, this is necessary when dealing with I/O operations. Writing data to a file system involves transferring data to servers. If the servers' I/O subsystems are slower than their NICs, data could queue up infinitely. Since this is not realistic the NICs must block transmission when there is not enough RAM available.
- Packet size is limited by a so-called network granularity[5] to ensure interleaved processing of packets. Messages are partitioned into the network granularity on the fly. There are two aspects concerning the network granularity's size.

[5]The default network granularity is 100 KiB.

Firstly, smaller packets result in more events in the simulation and consequently increase simulation time, but allow better interleaving of packets from multiple sources on one network component. Bigger packets decrease the simulation effort, but reduce interleaving.

3.3.2 I/O Request Processing

From the clients' point of view an I/O request is issued with a list of offset-size pairs — this is called `ListIO` internally. These lists of non-contiguous operations can be generated automatically by file views.

Each I/O operation issued by a client consists of several discrete steps, which are processed sequentially. In addition to the client an I/O operation may involve multiple servers, including their cache layers, I/O subsystems, NICs and the network. Data flow through the network is detailed above and consequently spared here. The actual processing of an I/O request depends on the request's type, that is, whether a read or write operation is requested.

Read Request

1. The client submits a read request to the NIC.
2. The network transmits requests to all servers involved in the I/O operation.
3. Each server accepts the request and announces the whole request to its cache layer.
4. The cache layer selects an I/O operation to perform next and forwards it to the I/O subsystem. It is important to note that the size of a single contiguous I/O operation is limited by the so-called I/O granularity[6].
5. Once the I/O subsystem finishes an operation the cache layer is notified and can then issue new operations. Data received from the cache layer is immediately added to the network flow, which might result in new network packets.
6. The client's read operation is implicitly finished once all data is received.

Write Request

1. The client submits a write request to the NIC.
2. The network transmits requests to all servers involved in the I/O operation.

[6]The default I/O granularity is 10 MiB.

3. Each server accepts the request and responds with an acknowledgement.
4. The client waits for all acknowledgements and starts sending the actual I/O data to all servers as soon as they are received.
5. Once the data is received by the server it notifies its cache layer about the request immediately. The cache layer puts the received data into the server's I/O cache and can then submit it to the I/O subsystem. If there is not enough free cache memory remaining the server will block further receives of data. Once the I/O subsystem finishes the job the cache is freed, allowing further requests to get queued. The cache layer can also choose which write operation to perform next, analogous to the read case.
6. Once a request is completed the server sends a write completion message to the client.
7. The client's write operation is finished once all write completion messages are received.

3.3.3 Cache Layers

The following cache layers are currently implemented in PIOsimHD-Simulator.

It is important to note that even though the I/O granularity defaults to 10 MiB, I/O operations sent across the network are split up into smaller operations to accommodate the network granularity, which defaults to 100 KiB.

`NoCache`

The `NoCache` cache layer does no caching at all. All incoming I/O operations are forwarded directly to the I/O subsystem. Among other things, this means that write operations take as long as the I/O subsystem needs to actually write out the data to the underlying storage devices. Additionally, read operations are preferred by the `NoCache` cache layer — and, in fact, by all other cache layers — when possible.

Since the `NoCache` cache layer does not combine I/O operations the maximum operation size is 100 KiB due to the network granularity.

`SimpleWriteBehindCache`

The `SimpleWriteBehindCache` cache layer does rudimentary caching. Incoming write operations are queued in the server's RAM to be written out in a background thread. This technique is called write-behind. As opposed to the `NoCache` cache layer this means that write operations do not block the calling client and return immediately once the data is received by the servers. The actual operation then finishes in the background.

Since the `SimpleWriteBehindCache` cache layer does not combine I/O operations the maximum operation size is 100 KiB due to the network granularity.

AggregationCache

The `AggregationCache` cache layer performs simple write optimizations as well as write-behind. It tries to combine the next I/O operation with as many other queued I/O operations as possible. This is done by performing forward and backward combination by respectively appending and prepending other I/O operations to the current I/O operation. Two I/O operations can be combined when they cover a contiguous region when merged — that is, $\text{offset}_1 + \text{size}_1 = \text{offset}_2$ or vice versa. However, this can — and should — be changed to also combine overlapping accesses in the future. This minor detail is irrelevant for the benchmarks used in this thesis, because they do not perform any overlapping accesses.[7]

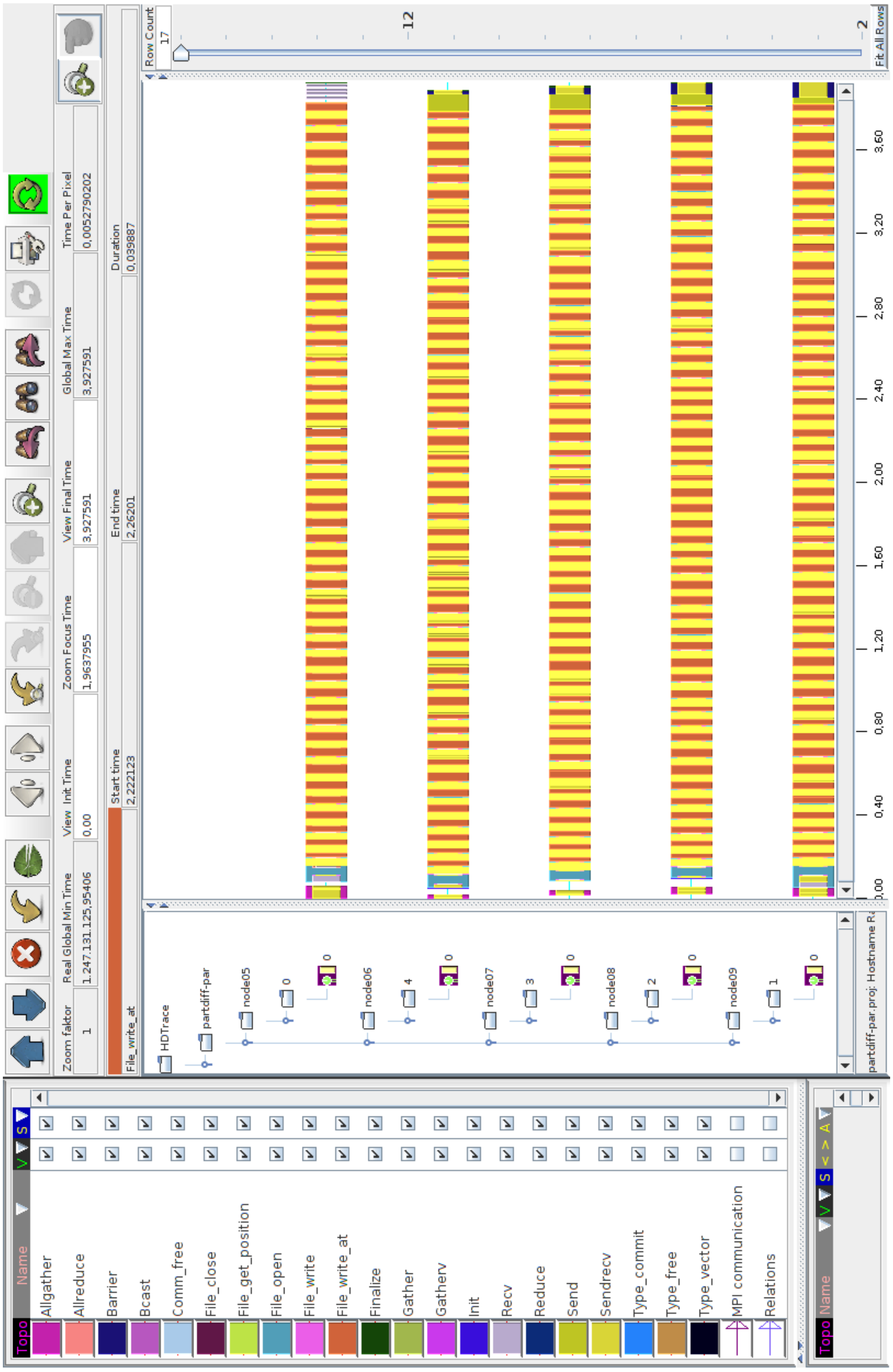
Since the `AggregationCache` cache layer combines I/O operations the operation size is only limited by the I/O granularity, that is, the maximum operation size is 10 MiB.

3.4 HDSunshot

Figure 3.2 (page 28) shows HDSunshot, PIOsimHD's trace visualization tool. It is based on Jumpshot[8], but adapted to support the HDTraceFormat and featuring many UI[9] enhancements. It supports post-mortem performance analysis of traced applications.

HDSunshot consists of the main timeline window and the legend on the left side. The legend allows the user to specify exactly which traced operations should be displayed in the timeline window. The timeline window displays a trace hierarchy on the left side. Applications can be found on the first level, nodes on the second level, ranks on the third level and threads on the fourth level. To the right of the trace hierarchy, the lower part contains a timeline with the traced operations themselves above it. Each traced operation is drawn in a different color to allow for easy differentiation. The bar at the top — below the icons — contains detailed information about the currently selected operation. For example, for read and write operations the amount of accessed data is displayed. Additionally, each operation's duration is displayed there.

[7]The random benchmarks can perform overlapping accesses, but this is very unlikely.
[8]See `http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/index.htm`.
[9]User Interface



Zoom faktor: 1
Real Global Min Time: 1.247.131.125,95406
View Init Time: 0,00
Zoom Focus Time: 1,9637955
View Final Time: 3,927591
Global Max Time: 3,927591
Time Per Pixel: 0,0052790202

File_write_at
Start time: 2,222123
End time: 2,26201
Duration: 0,039887

Row Count: 17

Figure 3.2: HDSunshot

The traces can be generated either by real MPI applications or by simulated ones. This is useful for comparing simulated applications with their real counterparts, which allows easy verification of the simulated results.

3.5 PVFS

PVFS[10] is a parallel cluster file system, which supports multiple data and metadata servers. To distribute the load, file data is striped across all available data servers with a default stripe size of 64 KiB. The first data server is chosen randomly, then a round-robin scheme is used. However, file metadata is not distributed. The metadata for any file is managed by exactly one metadata server. Which metadata server is responsible is determined by a hashing algorithm.

PVFS offers several features that make it attractive for research:

- Free software, released under the GPL[11]/LGPL[12]
- The code base has a manageable size of about 250 000 SLOC[13]
- Supports MPICH2
- Easy to install and configure

3.6 MPICH2

MPICH2[14] is a high-performance and widely portable implementation of the MPI standard. It also includes ROMIO to support the MPI-IO part of the MPI-2 standard. ROMIO supports various I/O optimizations, such as data sieving and the Two-Phase protocol.

Summary

The PIOsimHD framework contains components to simulate, trace and visualize applications. PIOsimHD-Model and PIOsimHD-Simulator allow simulating arbitrary network topologies, servers and client applications. PIOsimHD-Simulator contains several different cache layers which employ varying degrees of caching. MPICH2 and PVFS can be used to trace real MPI applications including MPI-IO. HDSunshot is used to visualize the resulting traces.

[10]PVFS is available at `http://www.pvfs.org/`.
[11]GNU General Public License
[12]GNU Lesser General Public License
[13]Source Lines of Code
[14]MPICH2 is available at `http://www.mcs.anl.gov/mpi/mpich/`.

Chapter 4

Design

In this chapter the design of the optimizations implemented in this thesis — that is, Server-Directed I/O and the Two-Phase protocol — is elaborated. Additionally, an alternative version of the Two-Phase protocol called Interleaved Two-Phase is presented.

4.1 Aggregation Cache

The `AggregationCache` cache layer currently only performs very basic optimizations for write operations. When the next operation is processed, all pending operations are examined to find out whether a pending operation can be appended or prepended to the next operation. Apart from this, the `AggregationCache` always processes operations in the order in which they are received from the clients.

These optimizations are extended to also work for read operations. This is done by simply performing the same examination and merging of operations in the read case.

4.2 Server-Directed I/O

Libraries for distributed I/O such as ROMIO often use client-side optimizations — such as the Two-Phase protocol — to improve the overall I/O performance. In contrast to this, Server-Directed I/O describes optimization techniques employed on the file server itself. One of the main goals of this thesis is to determine whether server-side optimizations are sufficient to provide acceptable I/O performance. These server-side optimizations are implemented as the Server-Directed I/O cache layer in PIOsimHD.

As mentioned in section 2.3 (page 19), the main problem with client-side optimizations is the fact that these optimizations usually do not have enough information to efficiently optimize operations. When using server-side optimizations like Server-Directed I/O the servers themselves decide which operations they should perform next and in which order they should be performed. Since the servers obviously have all information about the underlying storage devices more intelligent decisions can be made.

The main point of Server-Directed I/O is the fact that the servers determine when and how to perform I/O operations, as opposed to the usual approach of servers being simple data stores and the clients handling all I/O decisions. The advantage of this approach is the fact that the servers have access to all pending I/O requests and can use this knowledge, as opposed to the clients which only have access to their current I/O requests. Collective I/O can be used to alleviate this problem to some extent by granting the clients access to the I/O requests of all participating clients. However, this is also different from other server-side optimizations like the existing `AggregationCache`, because they simply process requests in FIFO[1] order and do not perform any reordering.

The actual optimizations are presented below with examples of the read and write optimizations.

4.2.1 Read

A large number of clients performing many small read operations can easily saturate the I/O system. The optimizations implemented in the Server-Directed I/O cache layer try to satisfy as many clients requests with as little actual I/O requests as possible and to determine the best way of handling these I/O requests.

The main method to do this is to merge multiple client requests into larger contiguous read operations. To be able to perform the merge operations as efficiently as possible all client requests are stored in per-file queues and sorted by the request's offset within the file. The Server-Directed I/O cache layer has access to all pending read requests and can change their order to improve performance.

Figure 4.1 (page 32) provides an example of the read optimizations performing three client requests as one larger read operation. For example, these three requests could be part of one collective non-contiguous I/O operation or originate from separate clients. The following steps can be seen:

(a) A new read request (#1) is inserted at offset 128 with size 64.
The request is inserted at position 0, because the queue is empty.

(b) A new read request (#2) is inserted at offset 64 with size 64.
The request is inserted at position 0, because its offset is smaller than the offset of request #1.

[1] First In, First Out

(c) A new read request (#3) is inserted at offset 128 with size 128.
The request is inserted at position 2, because its offset is not smaller than the offset of request #2.

(d) The read requests #1, #2 and #3 are merged together.
All three requests can be satisfied by reading the contiguous file region at offset 64 with size 192.

The actual implementation can also merge read requests which are separated by holes of a given size. For simplicity, this case is omitted in this example.

Since there is only one read requests remaining, it is clear which one needs to be processed next.

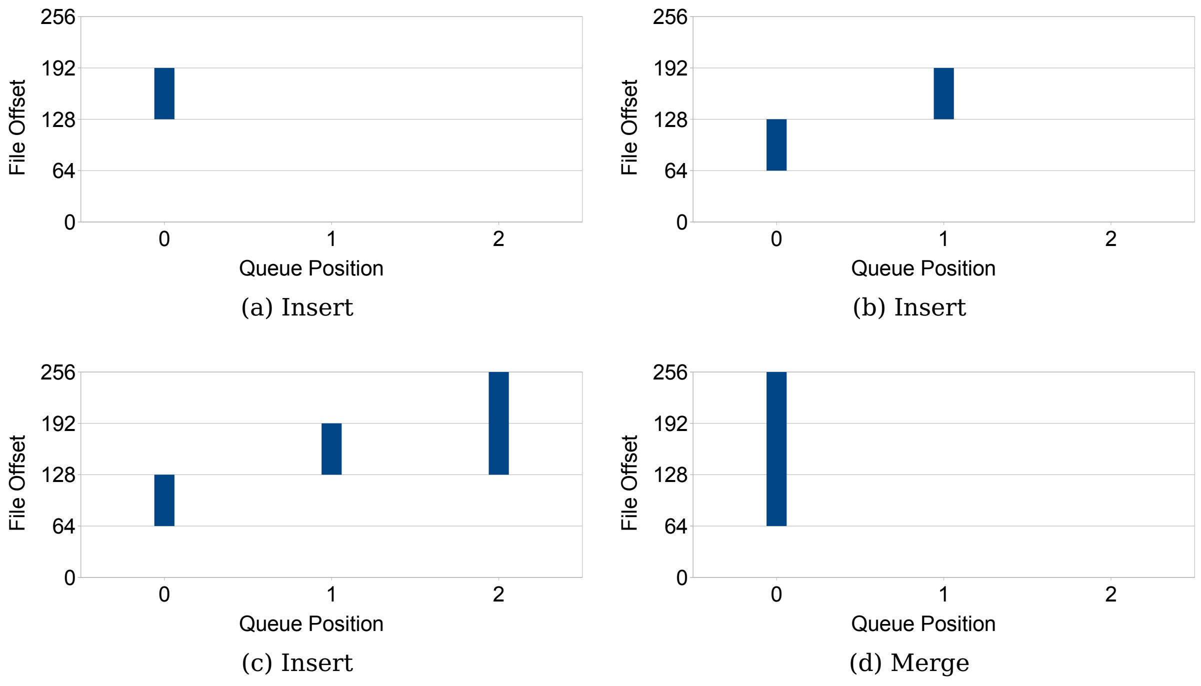


Figure 4.1: Server-Directed I/O — Read Optimizations

4.2.2 Write

The write optimizations implemented in the Server-Directed I/O cache layer also try to satisfy as many clients requests with as little actual I/O requests as possible and to determine the best way of handling these I/O requests.

The main method to do this is to discard unnecessary write requests early and merge multiple client requests into larger contiguous write operations. The client requests are stored in the same way as in the read case to allow efficient merge operations. The Server-Directed I/O cache layer can also change the order of pending write requests and delay the actual writing of the data to improve performance.

Figure 4.2 (page 34) provides an example of the write optimizations performing six client requests as two larger write operations. For example, these six requests could be part of one collective non-contiguous I/O operation or originate from separate clients. The following steps can be seen:

(a) A new write request (#1) is inserted at offset 128 with size 64.
The request is inserted at position 0, because the queue is empty.

(b) A new write request (#2) is inserted at offset 64 with size 64.
The request is inserted at position 0, because its offset is smaller than the offset of request #1.

(c) A new write request (#3) is inserted at offset 128 with size 128. The request is inserted at position 2, because its offset is not smaller than the offset of request #2.

(d) The write request #3 overwrites the write request #1.
The request would overwrite all the data, therefore the old request can be discarded.

(e) A new write request (#4) is inserted at offset 256 with size 64. The request is inserted at position 2, because its offset is not smaller than the offset of request #3.

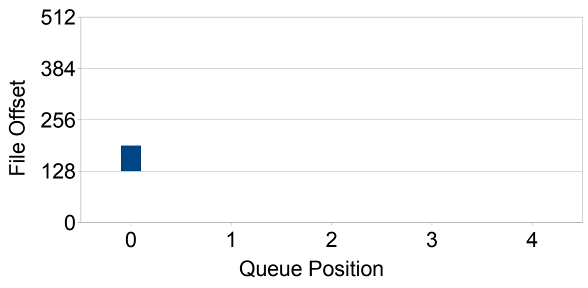
(f) A new write request (#5) is inserted at offset 384 with size 64. The request is inserted at position 2, because its offset is not smaller than the offset of request #4.

(g) A new write request (#6) is inserted at offset 448 with size 64. The request is inserted at position 2, because its offset is not smaller than the offset of request #5.

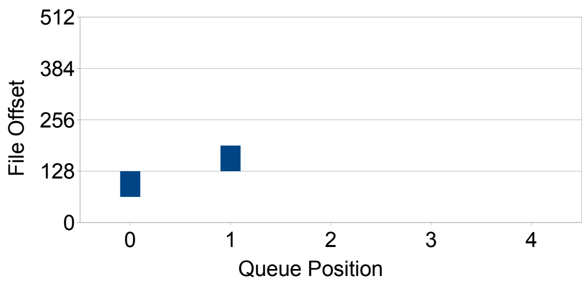
(h) The write requests #2, #3 and #4 are merged together. The write requests #5 and #6 are merged together.
All remaining five requests can be satisfied by writing two contiguous file regions at offset 64 with size 256 and offset 384 with size 128.

The situation presented in step (d) is a special case of a new write request overwriting data of an older write request. In the general case, the old request needs to be truncated or split up if the new request only overwrites part of the old request's data.

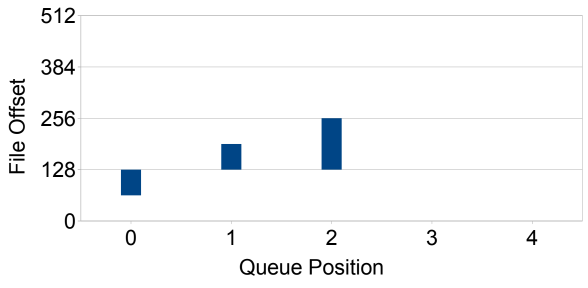
Since there are now two write requests remaining, the server can decide which one to process next. Selecting the best request to process can depend on a number of factors, including the offset and size of the request.



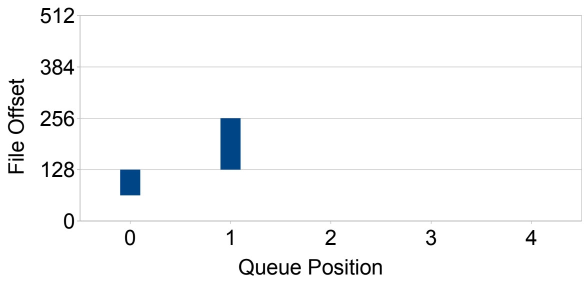
(a) Insert



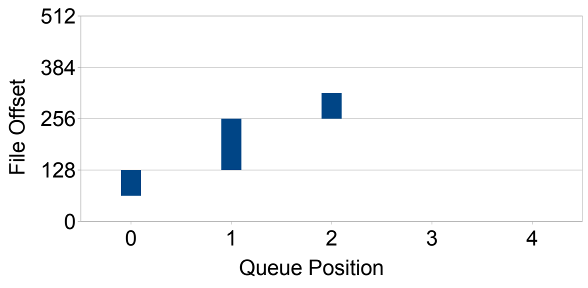
(b) Insert



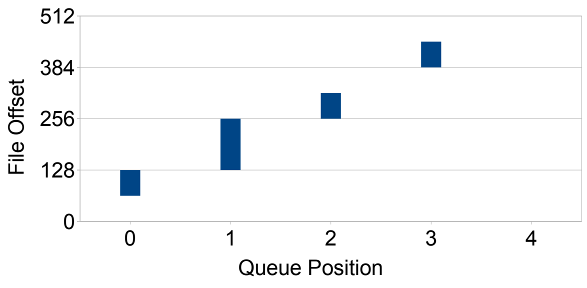
(c) Insert



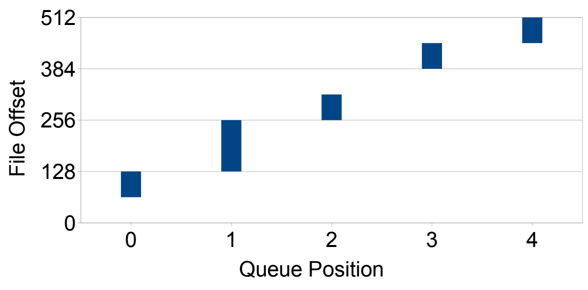
(d) Overwrite



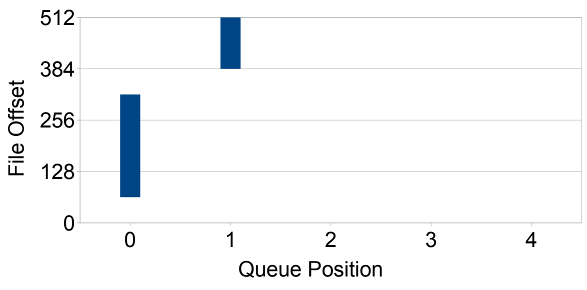
(e) Insert



(f) Insert



(g) Insert



(h) Merge

Figure 4.2: Server-Directed I/O — Write Optimizations

For a more detailed explanation and more information on additional optimizations, see section 5.2 (page 43).

4.3 Collective I/O

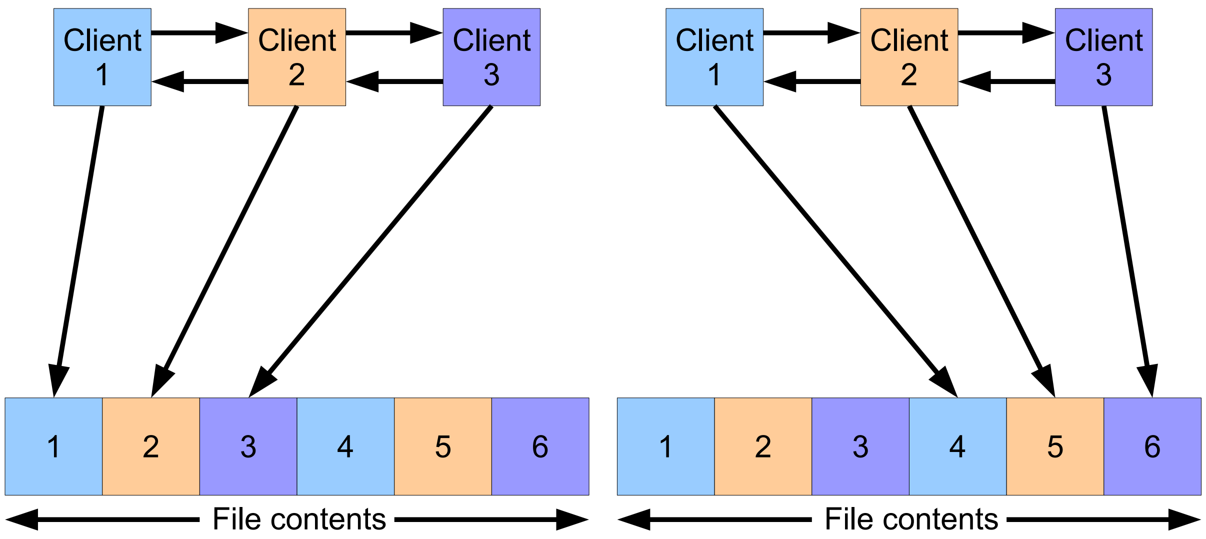
Collective I/O provides the basis for many client-side optimizations. One of these optimizations is the Two-Phase protocol used by ROMIO.

The simulator does not provide collective read and write operations, therefore they have to be implemented first. This is needed to be able to compare client-side optimizations — such as the Two-Phase protocol — with server-side optimizations like Server-Directed I/O.

4.3.1 Interleaved Two-Phase

The internal workings of the Two-Phase protocol are detailed in subsection 2.2.3 (page 17). It is easy to see that splitting the file into file domains and distributing these file domains among the clients leads to non-contiguous accesses in each iteration of the I/O phase. This can cause performance degradation, because these access patterns can not be optimized easily by the underlying cache layer.

Figure 4.3a and 4.3b show the I/O phase of the Interleaved Two-Phase protocol. As can be seen, the clients access a contiguous region of data in each iteration. This grants the I/O subsystem more room for optimizations.



(a) Phase 2 — Iteration 1

(b) Phase 2 — Iteration 2

Figure 4.3: Interleaved Two-Phase

This modification works exactly like the original Two-Phase protocol, except that accesses are performed in a contiguous fashion to minimize seek operations. This is achieved by not distributing file domains among the clients, but rather dividing the file into chunks of the size of the Two-Phase buffer. The first chunk is then accessed by the first client, the second chunk by the second client, and so on. Consequently, each client accesses one such chunk per iteration. This process is then repeated as necessary until all data is read or written. Since the clients' accesses are interleaved this modification is called the Interleaved Two-Phase protocol.

Summary

Server-Directed I/O provides optimizations for both read and write operations. It tries to aggregate as many operations as possible to decrease the load on the I/O subsystem and improve overall performance. Additionally, it has a full overview of all pending operations and can use this information to select the next operation based on several factors to maximize performance. The Interleaved Two-Phase protocol is a modification of ROMIO's Two-Phase protocol, which only accesses contiguous file regions.

Chapter 5

Implementation

In this chapter an overview of the internal structure of PIOsimHD-Model and PIOsimHD-Simulator is given. This includes the implementation of some example components and commands. Additionally, the actual implementations of the optimizations already presented in chapter 4 (page 30) — that is, the Server-Directed I/O component and the Two-Phase commands — are explained in detail.

5.1 Internals

PIOsimHD supports arbitrary component and command models and implementations. Example for both are given to explain the workflow needed to extend PIOsimHD with new components and commands, which is needed to integrate Server-Directed I/O and collective I/O into PIOsimHD.

5.1.1 Component Implementations

The simulator supports implementation of arbitrary components for simulating hardware and software. For example, it is possible to model abstract concepts like servers and physical components like cache layers within the servers.

The following server component model possesses a size attribute. For example, this size could represent the size of the server's main memory or the size of a cache.

Listing 5.1 shows the `Example` server component model. This model should be put into the file `PIOsimHD-Model/src/de/hd/pvs/piosim/model/components/Example/Example.java`. The component model is used to store attributes for this specific component and provides the basis for derived server component models for this server component.

Listing 5.1: Server Component Model Example

```
abstract public class Example
extends BasicComponent<Server> {
  @Attribute
  @NotNegativeOrZero
  long size = 0;

  final public void setSize(long size) {
    this.size = size;
  }

  final public int getSize() {
    return size;
  }

  @Override
  final public String getComponentType() {
    return Example.class.getSimpleName();
  }
}
```

Listing 5.2 shows the `SimpleExample` server component model. This model should be put into the file `PIOsimHD-Model/src/de/hd/pvs/piosim/model/components/-Example/SimpleExample.java`. This server component model simply inherits all attributes of the server component model shown in listing 5.1 (page 37).

Listing 5.2: Simple Server Component Model Example

```
public class SimpleExample
extends Example {
}
```

Listing 5.3 shows the interface used for this server component. This interface should be put into the file `PIOsimHD-Simulator/src/de/hd/pvs/piosim/simula-tor/components/Example/IGExample.java`. Interfaces can be used to force all server component implementations to provide a common interface. This makes it possible to provide multiple models for a single component.

Listing 5.3: Server Component Interface Example

```
public interface IGExample<Type extends SPassiveComponent>
extends IGComponent<Type>
{
  public void example();
}
```

Listing 5.4 (page 39) shows the `SimpleExample` server component implementation. This implementation should be put into the file `PIOsimHD-Simulator/src/de/hd/-pvs/piosim/simulator/components/Example/GSimpleExample.java`. This provides the server component implementation for the server component model shown in listing 5.2.

Listing 5.4: Simple Server Component Implementation Example

```java
public class GSimpleExample
extends SPassiveComponent<SimpleExample>
implements IGExample<SPassiveComponent<SimpleExample>>
{
  @Override
  public void example() {
    ...
  }
}
```

The simulator only uses component implementations specified in the file `Model-ToSimulatorMapper.txt`. This file can be used to specify multiple component implementations for a single component. For more details on this, see subsection 5.1.2 (page 40).

Listing 5.5 shows the file `ModelToSimulationMapper.txt`. Components themselves are prefixed with `+`, followed by their respective component implementations. In this case the `Example` component is implemented by `GSimpleExample`.

Listing 5.5: Component Mapping

```
# This file describes the existing model component of a given type i
   ↪ .e. "Node" the available Model implementations and mapping to
   ↪ the simulation implementation.
+ClientProcess
de.hd.pvs.piosim.model.components.ClientProcess.ClientProcess = de.
   ↪ hd.pvs.piosim.simulator.components.ClientProcess.
   ↪ GClientProcess

...

+Example
de.hd.pvs.piosim.model.components.Example.SimpleExample = de.hd.pvs.
   ↪ piosim.simulator.components.Example.GSimpleExample
```

5.1.2 Command Implementations

The simulator supports implementation of arbitrary commands for simulating applications. For example, it is possible to implement simple commands like the POSIX[1] `read()` and `write()` functions, and more complex commands like the MPI-IO functions.

Listing 5.6 shows the `Example` command model. This model should be put into the file `PIOsimHD-Model/src/de/hd/pvs/piosim/model/program/commands/Example.java`. The command model provides a basis for the command implementations, that is, all command implementations get passed an instance of the command model. Models can also have so-called attributes. These attributes can be annotated, which allows them to be automatically populated with values from traces and checked for validity. For example, the `Example` command model has one attribute called `communicator`. The `NotNull` annotation specifies that this attribute must be set, while the `Attribute` annotation specifies the attribute's name to be `cid` in traces.

Listing 5.6: Command Model Example

```java
public class Example
extends Command {
  @NotNull
  @Attribute(xmlName="cid")
  protected Communicator communicator;

  public Example() {
    ...
  }

  public helloWorld() {
    ...
  }
}
```

Listing 5.7 (page 41) shows the `SimpleExample` command implementation. This implementation should be put into the file `PIOsimHD-Simulator/src/de/hd/pvs/piosim/simulator/program/Example/SimpleExample.java`. The simulator calls the command's `process()` method, which can be separated into so-called steps. The first step is always `CommandProcessing.STEP_START`, while `CommandProcessing.STEP_COMPLETED` signifies that the command is finished. Additional steps can be defined at will, for more details see listing 5.8 (page 41).

[1]Portable Operating System Interface (for Unix)

Listing 5.7: Simple Command Example

```java
public class SimpleExample
extends CommandImplementation<Example> {
  @Override
  public void process(Example cmd,
                      CommandProcessing OUTresults,
                      GClientProcess client,
                      int step,
                      NetworkJobs compNetJobs) {
    switch (step) {
    case (CommandProcessing.STEP_START): {
      // Do some work ...
      cmd.helloWorld();
      ...

      OUTresults.setNextStep(CommandProcessing.STEP_COMPLETED);

      return;
    }
    }

    return;
  }
}
```

The `SimpleExample` implementation starts with `CommandProcessing.STEP_START`, calls the `helloWorld()` method of the command model `cmd` and then sets the next step to `CommandProcessing.STEP_COMPLETED` with `OUTresults.setNextStep()`, ending the command's processing.

Listing 5.8 shows the `SophisticatedExample` command implementation. This implementation should be put into the file `PIOsimHD-Simulator/src/de/hd/pvs/-piosim/simulator/program/Example/SophisticatedExample.java`.

Listing 5.8: Sophisticated Command Example

```java
public class SophisticatedExample
extends CommandImplementation<Example> {
  @Override
  public void process(Example cmd,
                      CommandProcessing OUTresults,
                      GClientProcess client,
                      int step,
                      NetworkJobs compNetJobs) {

    final int ANOTHER_STEP = 2;

    switch (step) {
    case (CommandProcessing.STEP_START): {
      int rank = client.getModelComponent().getRank();

      // Do some more work ...
      ...

      OUTresults.setNextStep(ANOTHER_STEP);

      return;
    }
    case (ANOTHER_STEP): {
      // Do even more work ...
      ...

      // Call the SimpleExample implementation
      OUTresults.invokeChildOperation(cmd, CommandProcessing.
          ↪ STEP_START, de.hd.pvs.piosim.simulator.program.Example.
          ↪ SimpleExample.class);

      OUTresults.setNextStep(CommandProcessing.STEP_COMPLETED);

      return;
    }
    }

    return;
  }
}
```

The `SophisticatedExample` implementation starts with `CommandProcessing.-STEP_START`, performs some additional work and then sets the next step to `ANOTHER_STEP`, which is a step specific to the `SophisticatedExample` command implementation. In the `ANOTHER_STEP` step, the implementation then uses `OUTresults.invokeChildOperation()` to start a nested operation of the `SimpleExample` command implementation. This in turn executes the `SimpleExample`'s `process()` method with `CommandProcessing.STEP_START` and returns when the processing would normally end — that is, when CommandProcessing.STEP_COMPLETED is set. Afterwards, it sets the next step to CommandProcessing.STEP_COMPLETED, ending the command's processing.

The simulator only uses command implementations specified in the file CommandToSimulationMapper.txt. This file can be used to specify multiple command implementations for a single command. For example, the SimpleExample and SophisticatedExample command implementations can be used to provide two different command implementations for the Example command.

Listing 5.9 shows the file CommandToSimulationMapper.txt. Commands themselves are prefixed with +, followed by their respective command implementations. In this case the Example command is implemented by Global.NoOperation, Example.SimpleExample and Example.SophisticatedExample. The last specified command implementation is used by default, but this can be changed at will.

Listing 5.9: Command Mapping

```
# define the command groups and the contained implementations and
    ↪ the simulator implementations
# each row is either a set of commands which are implemented or the
    ↪ actual set of implementation classes for these commands.

# All undefined operations are mapped to NoOperation:
+NoOperation
de.hd.pvs.piosim.simulator.program.Global.NoOperation

# Automatically generated within the simulator.
+Compute
de.hd.pvs.piosim.simulator.program.Global.NoOperation
de.hd.pvs.piosim.simulator.program.Compute.Time

...

+de.hd.pvs.piosim.model.program.commands.Example
de.hd.pvs.piosim.simulator.program.Global.NoOperation
de.hd.pvs.piosim.simulator.program.Example.SimpleExample
de.hd.pvs.piosim.simulator.program.Example.SophisticatedExample
```

5.2 Server-Directed I/O

As described in subsection 5.1.1 (page 37), Server-Directed I/O is implemented as a new server component, namely a new cache layer called ServerDirectedIO. To integrate the Server-Directed I/O cache layer into the existing framework a new model and implementation have to be added. The model is put into the file PIOsimHD-Model/src/de/hd/pvs/piosim/model/components/-ServerCacheLayer/ServerDirectedIO.java and is omitted here, because it is a trivial extension of the existing AggregationCache model. This implementation is put into the file PIOsimHD-Simulator/src/de/hd/pvs/piosim/simulator/components/ServerCacheLayer/GServerDirectedIO.java and is now presented in detail. Lastly, the new implementation needs to be added to the file ModelToSimulationMapper.txt, allowing it to be used as a cache layer.

Listing 5.10 shows the Server-Directed I/O class. It is based on the aggregation cache layer, which is basically a simpler version of the Server-Directed I/O cache layer. The aggregation cache layer also tries to combine multiple operations into larger ones, but is less efficient and can not cope as well with multiple files. It also does not perform any optimizations regarding the order in which pending I/O operations are performed.

Listing 5.10: GServerDirectedIO.java — Main Class

```java
public class GServerDirectedIO extends GAggregationCache {
  ...
}
```

Listing 5.11 shows the data structures used by the Server-Directed I/O implementation. lastFile and lastOffset are used to store the last accessed file and offset respectively. This information is used to minimize costly seek operations. queuedReadJobs and queuedWriteJobs are used to store per-file queues of outstanding I/O operations in the form of IOJobs. The other cache layers use global queues for all operations. However, per-file queues allow easier and faster merging of I/O operations. Additionally, these queues are sorted by the IOJobs' file offsets.

Listing 5.11: GServerDirectedIO.java — Data Structures

```java
MPIFile lastFile = null;
long lastOffset = -1;

HashMap<MPIFile, LinkedList<IOJob>> queuedReadJobs = new HashMap<
   ↪ MPIFile, LinkedList<IOJob>>();
HashMap<MPIFile, LinkedList<IOJob>> queuedWriteJobs = new HashMap<
   ↪ MPIFile, LinkedList<IOJob>>();
```

Listing 5.12 (page 45) shows the IOJob interface. An IOJob represents a single I/O operation operating on file at offset with size Bytes of data. IOJobs can either be READ or WRITE operations.

Listing 5.12: `GServerDirectedIO.java` — `IOJob` Interface

```java
public class IOJob implements EventData {
  static public enum IOOperation {
    READ,
    WRITE
  }

  final private MPIFile file;
  final private long size;
  final private long offset;

  final private IOOperation type;

  public IOJob(MPIFile file, long size, long offset, IOOperation
     ↪ operation);
  public IOJob(IOJob oldJob);

  public long getOffset();
  public long getSize();
  public IOOperation getType();
  public MPIFile getFile();
}
```

Listing 5.13 shows the `getNextSchedulableJob` method. As the name suggests, this method returns the next `IOJob` that should be performed by the I/O subsystem. Currently, read operations are preferred. File systems like ZFS[2] also prioritize read operations to keep the system responsive, but make sure that no starvation of write operations occurs. This functionality is still lacking in the Server-Directed I/O implementation — and in fact all other cache layers —, because there is currently no priority or way to determine how long operations have been waiting in the queue.

Listing 5.13: `GServerDirectedIO.java` — `getNextSchedulableJob`

```java
@Override
protected IOJob getNextSchedulableJob() {
  IOJob io;

  io = getJob(IOOperation.READ);

  if (io == null) {
```

[2]Zettabyte File System

```
    io = getJob(IOOperation.WRITE);
  }

  return io;
}
```

Listing 5.14 shows the `getJob` method. This method does the actual work of selecting an `IOJob` that should be performed next. It uses different strategies to decide which `IOJob` is best suited for execution.

Listing 5.14: `GServerDirectedIO.java` — `getJob`

```
private IOJob getJob(IOOperation type) {
  ...

  LinkedList<IOJob> list = null;
  LinkedList<IOJob> lastFileList = null;

  IOJob io = null;
  IOJob ioClose = null;
  IOJob ioLarge = null;

  Iterator<IOJob> it = null;

  ...

  for (LinkedList<IOJob> l : queue.values()) {
    long size = 0;
    long tmp = 0;

    it = l.iterator();

    /* Determine the size of the job queue. */
    while (it.hasNext()) {
      tmp += it.next().getSize();
    }

    /* Prefer operating on the last file. */
    if (l == lastFileList) {
      tmp *= 10;
    }

    /* Use the largest job queue. */
    if (tmp > size) {
      size = tmp;
      list = l;
    }
  }

  ...

  if (type == IOOperation.READ) {
    list = mergeReadJobs(list);
  } else if (type == IOOperation.WRITE) {
    list = mergeWriteJobs(list);
  }

  ...

  while (it.hasNext()) {
    IOJob cur = it.next();

    if (ioClose == null || (lastOffset >= 0 && Math.abs(cur.
        ↪ getOffset() - lastOffset) < 5 * 1024 * 1024 && cur.
        ↪ getOffset() < ioClose.getOffset())) {
      /* Determine the IOJob closest to the last position. */
      ioClose = cur;
    } else if (ioLarge == null || cur.getSize() > ioLarge.getSize())
        ↪  {
      /* Determine the largest IOJob. */
      ioLarge = cur;
    }

  }

  ...
}
```

The first step is to determine which per-file queue to use. Firstly, `queue` is set to either `queuedReadJobs` or `queuedWriteJobs`, depending on `type`. Then it computes the total size of all operations in each per-file queue — that is, how much data should be read or written — and uses the largest one. However, the queue of the file that was last accessed is preferred by multiplying its size with a factor of `10`.

In the next step the `mergeReadJobs` or `mergeWriteJobs` method is used to merge as many `IOJobs` in the selected queue as possible. These methods try to combine multiple contiguous operations into larger operations. Additionally, the `mergeReadJobs` method supports data sieving, that is, reading more data than is necessary to combine even operations that are non-contiguous. However, data sieving is disabled by default, because it can cause performance degradation in some cases. Therefore, a cost-benefit analysis needs to be done in the future to determine in which cases the data sieving algorithm should be used. The algorithm then needs to be adapted accordingly.

In the last step the actual `IOJob` is selected. Currently, the closest — in relation to the last access — and the largest `IOJobs` are considered. In the special case that an `IOJob` continues the last access — that is, starts at the last accessed offset — it can be used immediately, because in this case the underlying storage device does not have to perform any seek operation. This is of course only true for HDDs, which are the default storage devices used in PIOsimHD-Simulator for now. Code for this functionality exists, but is disabled at the moment, because tests suggest that this can decrease performance in some cases. Therefore, this special case must be analyzed further.

5.2.1 Delayed Writes

Listing 5.15 shows the explicit write-behind caching in the `getJob()` method. Obviously, this only applies to write operations. In the first step the size of all operations in all per-file queues is determined. Then it is checked whether this size exceeds the `cacheSize` and when the last flush happened.

Listing 5.15: `GServerDirectedIO.java` — Delayed Writes

```
...

if (type == IOOperation.WRITE) {
  long tmp = 0;
  Epoch time = getSimulator().getVirtualTime();

  for (List<IOJob> l : queue.values()) {
    it = l.iterator();

    while (it.hasNext()) {
      tmp += it.next().getSize();
    }
  }

  if (tmp < cacheSize && lastFlush != null && (lastFlush.add(0.33).
     ↪ compareTo(time) > 0)) {
    return null;
  }
}

...
```

Pending operations are only executed whenever the pre-defined cache size is exceeded or the last flush happened more than ⅓ s ago. Otherwise, operations are cached and scheduled for later execution. This can significantly reduce the number of total I/O operations, because it allows more efficient merging.

However, to be able to fully utilize this explicit write-behind caching a new flush command must be introduced, because I/O operations can be lost otherwise. For example, I/O operations could still be cached when all clients finish, which causes the simulator to terminate. Pending I/O operations are then lost.

Listing 5.16 shows the `addFlush()` method, which is called whenever clients request a flush by sending a `RequestFlush`. This method first tries to merge all remaining pending I/O operations for this particular file. Afterwards, all merged I/O operations are executed. Flush requests are announced to all cache layers, but currently only the `ServerDirectedIO` one makes use of it.

Listing 5.16: `GServerDirectedIO.java` — Flushing

```
@Override
protected void addFlush(RequestFlush req) {
  if (queuedWriteJobs.get(req.getFile()) == null) {
    return;
  }

  List<IOJob> list;
  Iterator<IOJob> it;

  list = mergeWriteJobs(queuedWriteJobs.get(req.getFile()));
  it = list.iterator();

  while (it.hasNext()) {
    IOJob io = it.next();

    ioSubsystem.startNewIO(io);
    numberOfScheduledIOOperations++;
  }

  queuedWriteJobs.get(req.getFile()).clear();
}
```

When this method finishes no more pending write operations for the given file exist. File flushing is currently only triggered by closing a file with the `Fileclose` command.

5.2.2 Performance Considerations

Both the `AggregationCache` and `ServerDirectedIO` cache layers use special data structures and algorithms to enable merging of I/O operations. When implementing these optimizations in real-life file systems, the performance of these algorithms is very important, because the merging is performed each time an I/O operation is forwarded to the I/O subsystem. Therefore, an overview of their algorithmic complexity is given below.

Aggregation Cache

The aggregation cache uses a simple data structure to store all pending I/O operations. This queue is unsorted.

Insertion of a new operation is of order $O(1)$, because the new operation can be simply appended to the queue.

However, because the queue is unsorted merging of I/O operations is of order $O(n^2)$ where n is the total number of pending I/O operations.

Server-Directed I/O

The Server-Directed I/O cache layer uses per-file data structures to store pending I/O operations for each file individually. Each of the queues is sorted by the operations' offsets.

Insertion of a new operation is of order $O(m)$ where m is the number of pending I/O operations for the current file. This is achieved by using a linked list and walking the whole queue on each insertion.

Due to the sorted per-file queues merging of I/O operations is of order $O(m)$ where m is the number of pending I/O operations for the current file. This is achieved by walking the queue once and merging operations on the fly.

5.3 Collective I/O

As described in subsection 5.1.2 (page 40), collective I/O operations for reading and writing are implemented as new commands, namely the `Filereadall` and `Filewriteall` commands. To integrate the collective I/O commands into the existing framework new models and implementations have to be added. The models are put into the files `PIOsimHD-Model/src/de/hd/pvs/piosim/model/program/commands/Filereadall.java` and `PIOsimHD-Model/src/de/hd/pvs/piosim/model/program/commands/Filewriteall.java` respectively. The models are again omitted. These implementations are put into the directories `PIOsimHD-Simulator/src/de/hd/pvs/piosim/simulator/program/Filereadall/` and `PIOsimHD-Simulator/src/de/hd/pvs/piosim/simulator/program/Filewriteall/` respectively. Both directories contain implementations using individual I/O (`Direct.java`), the Two-Phase protocol (`TwoPhase.java`) and the Interleaved Two-Phase protocol (`InterleavedTwoPhase.java`). Lastly, the new implementations need to be added to the file `CommandToSimulationMapper.txt`, allowing them to be used as commands.

In order to properly implement collective operations it is necessary to be able to store some kind of meta information about the collective operation. For example, it is necessary to know which clients take part in this particular collective operation. It may also be necessary to store some kind of state information. Since no general solution is readily available, it has to be implemented for the collective operations considered here — that is, `Filereadall` and `Filewriteall`. The `Filewriteall` implementation in `TwoPhase.java` is used as an example.

Listing 5.17 shows the data structures used to hold the necessary meta information. `sync_blocked_clients` is used to synchronize all participating clients for the collective operation. `meta_info` is used to share meta information among all clients.

Listing 5.17: `Filewriteall/TwoPhase.java` — Data Structures

```
private static HashMap<FilewriteallWrapper, List<
    ↪ FilewriteallContainer>> sync_blocked_clients;
private static HashMap<Filewriteall, List<FilewriteallContainer>>
    ↪ meta_info;
```

Listing 5.18 (page 52) shows the `FilewriteallWrapper` class used to synchronize all participating clients of a collective operation. It is constructed in such a way that a `FilewriteallWrapper` instance created for a `Filewriteall` command is equal to another `FilewriteallWrapper` instance if and only if both used `Filewriteall` commands belong to the same collective operation. This is accomplished by overwriting the `equals()` and `hashCode()` methods.

Listing 5.18: `Filewriteall/TwoPhase.java` — `FilewriteallWrapper`

```java
final class FilewriteallWrapper {
  private Filewriteall command;

  public FilewriteallWrapper(Filewriteall cmd) {
    command = cmd;
  }

  @Override
  public boolean equals(Object obj) {
    if (obj.getClass() != getClass()) {
      return false;
    }

    FilewriteallWrapper compare = (FilewriteallWrapper) obj;

    return (compare.command.getCommunicator() == this.command.
       ↪ getCommunicator())
        && (compare.command.getProgram().getApplication() == this.
           ↪ command.getProgram().getApplication())
        && (compare.command.getClass() == this.command.getClass());
  }

  @Override
  public int hashCode() {
    return command.getCommunicator().hashCode() + command.getClass()
       ↪ .hashCode();
  }
}
```

By putting all matching commands into the `sync_blocked_clients` structure this class is used to find and collect all participating clients.

Listing 5.19 shows the `FilewriteallContainer` class used to provide meta information to the clients during the runtime of the collective operation.

Listing 5.19: `Filewriteall/TwoPhase.java` — `FilewriteallContainer`

```java
final class FilewriteallContainer {
  private Filewriteall command;
  private GClientProcess clientProcess;
  private CommandProcessing commandProcessing;

  ...

  public FilewriteallContainer(Filewriteall command, GClientProcess
     ↪ clientProcess, CommandProcessing commandProcessing) {
    this.command = command;
    this.clientProcess = clientProcess;
    this.commandProcessing = commandProcessing;

    ...
  }

  public Filewriteall getCommand() {
    return command;
  }

  public GClientProcess getClientProcess() {
    return clientProcess;
  }

  public CommandProcessing getCommandProcessing() {
    return commandProcessing;
  }

  public Integer getRank() {
    return clientProcess.getModelComponent().getRank();
  }

  ...
}
```

By putting the meta information of all participating clients into the `meta_info` structure, all clients have access to this meta information during the collective operation. It is also possible to access other clients' meta information when necessary.

Additional meta information can be added easily. For example, the Two-Phase operations use it to store the current iteration of the I/O phase.

Listing 5.20 (page 54) shows the `STEP_START` step used to fill the provided data structures. As can be seen, all clients first create `FilewriteallWrapper` and `FilewriteallContainer` instances based on the current command. All matching commands are added to the `sync_blocked_clients` structure. Completion is checked by comparing the size of the communicator — that is, the number of participating clients — and the number of collected commands. Afterwards, meta information is stored in the `meta_info` structure for each client. Finally, the `sync_blocked_clients` structure is cleaned up. The `meta_info` structure is cleaned up when the clients finish the operation — that is, in the last step of the `process()` method.

Listing 5.20: `Filewriteall/TwoPhase.java` — `STEP_START`

```java
@Override
public void process(Filereadall cmd, CommandProcessing OUTresults,
   ↪ GClientProcess client, int step, NetworkJobs compNetJobs) {
  ...

  switch (step) {
  case (CommandProcessing.STEP_START): {
    FilereadallWrapper wrapper = new FilereadallWrapper(cmd);
    List<FilereadallContainer> waitingClients = sync_blocked_clients
       ↪ .get(wrapper);
    FilereadallContainer container = new FilereadallContainer(cmd,
       ↪ client, OUTresults);

    ...

    waitingClients.add(container);

    if (waitingClients.size() < cmd.getCommunicator().getSize()) {
      ...
    } else {
      List<FilereadallContainer> cmds = new ArrayList<
         ↪ FilereadallContainer>();

      for (FilereadallContainer c : waitingClients) {
        cmds.add(c);
      }

      ...

      for (FilereadallContainer c : waitingClients) {
        meta_info.put(c.getCommand(), cmds);
      }

      ...

      sync_blocked_clients.remove(wrapper);
    }

    ...
  }

  ...
}
```

5.3.1 Two-Phase

The implementation of the Two-Phase protocol described here follows the implementation in ROMIO as closely as possible. However, some differences exist. For example, when using the Two-Phase protocol to perform a collective write operation ROMIO's implementation uses a read-modify-write operation to keep write accesses contiguous. This does not matter for the benchmarks used in chapter 6 (page 61), because their access patterns are designed in such a way as to result in contiguous accesses when aggregated by the Two-Phase protocol. However, it may degrade performance for non-optimizing cache layers.

The discrete steps in the `process()` method can be seen as states of a simple state machine[3]. This can be used to illustrate the inner workings of the Two-Phase implementations of the `Filereadall` and `Filewriteall` commands. The state machine would then starts in its start state called `STEP_START` and would end when it reaches its end state `STEP_COMPLETED`.

This is illustrated by state machine figures, which are structured as follows: Arrows show possible transitions between the individual states. The arrow without a previous state signifies the start state, which is called `STEP_START`. The end state is called `STEP_COMPLETED` and illustrated with a darker background color. The separated area called `Direct` signifies a nested state machine, that is, the existing `Direct` state machine is executed in the context of the Two-Phase state machine. This can be used to avoid duplicating existing functionality.

Read

Figure 5.1 (page 56) shows the corresponding state machine for the Two-Phase implementation of the `Filereadall` command, followed by a description of the state machine's individual states.

[3]For more information about state machines, see `http://en.wikipedia.org/wiki/Finite-state_machine`.

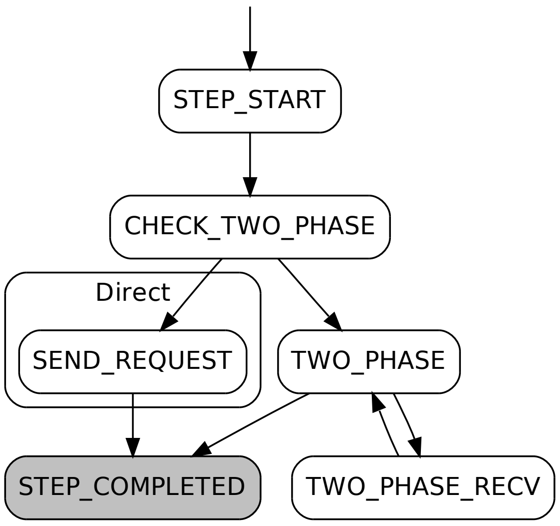


Figure 5.1: Two-Phase Read State Machine

STEP_START The purpose of this state is to synchronize all participating clients and set up all required data structures. For more information, see section 5.3 (page 51).

CHECK_TWO_PHASE This state is used to check whether the Two-Phase protocol should be used at all. The current way of doing this is to check whether the accesses of the participating clients overlap. Obviously, this requires communication between the clients to exchange their accesses. If they do not, the Two-Phase protocol is not used and a transition to `SEND_REQUEST` of the `Direct` state machine is performed, which simply uses individual I/O operations. Otherwise a transition to `TWO_PHASE` is done.

TWO_PHASE This state is used to determine the file region to read. This is done by using the minimum and maximum offsets and reading everything in between. The resulting file region is then distributed evenly across all clients — each client is then responsible for a so-called file domain. Each client then reads as much data as possible — this is limited by an internal Two-Phase buffer size. All clients also tell other clients which data they need from their respective file domains. If there is data remaining to be read a transition to `TWO_PHASE_RECV` is performed. Otherwise a transition to `STEP_COMPLETED` is done and the meta information for this operation is cleaned up.

TWO_PHASE_RECV In this state clients have received the data from their respective file domains. Requested data is then distributed to all other clients, which requested it in the previous state. A transition to TWO_PHASE is performed.

STEP_COMPLETED This state signals the end of the state machine. All data is read and the Filereadall command is completed.

In the Direct implementation of the Filereadall command each client internally uses individual I/O calls to read their requested data. This implementation is used when the CHECK_TWO_PHASE state determines that the Two-Phase protocol should not be used.

The states of the Direct state machine are as follows:

SEND_REQUEST In this state each client requests the needed data directly from the involved servers. A transition to STEP_COMPLETED is performed.

Write

Figure 5.2 shows the corresponding state machine for the Two-Phase implementation of the Filewriteall command, followed by a description of the state machine's individual states.

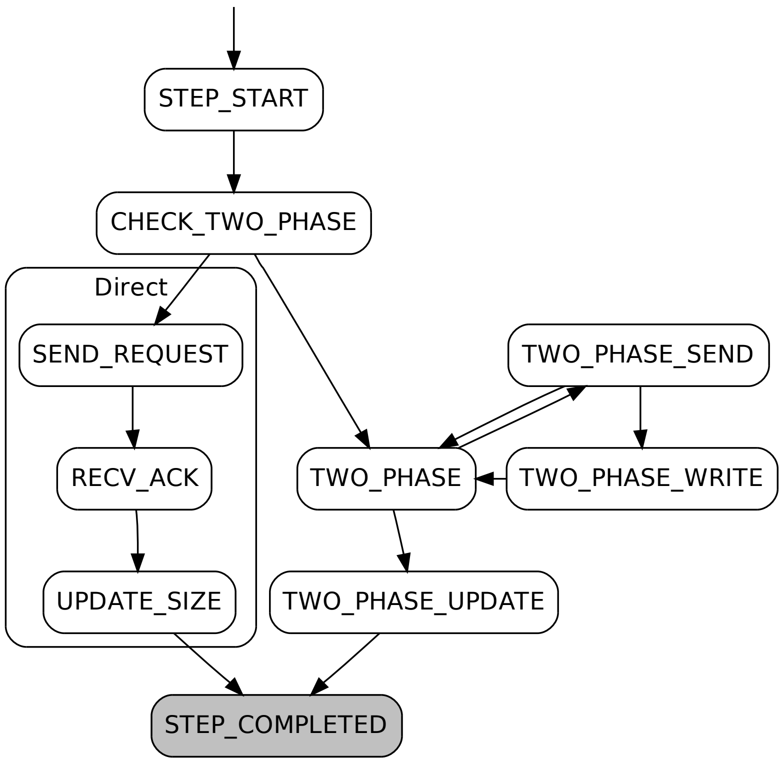


Figure 5.2: Two-Phase Write State Machine

STEP_START The purpose of this state is to synchronize all participating clients and set up all required data structures. For more information, see section 5.3 (page 51).

CHECK_TWO_PHASE This state is used to check whether the Two-Phase protocol should be used at all. The current way of doing this is to check whether the accesses of the participating clients overlap. Obviously, this requires communication between the clients to exchange their accesses. If they do not, the Two-Phase protocol is not used and a transition to `SEND_REQUEST` of the `Direct` state machine is performed, which simply uses individual I/O operations. Otherwise a transition to `TWO_PHASE` is done.

TWO_PHASE This state is used to determine the file region to write. This is done by using the minimum and maximum offsets. The resulting file region is then distributed evenly across all clients — each client is then responsible for a so-called file domain. Each client then sends as much of their data as possible to the owners of the respective file domains — this is limited by an internal Two-Phase buffer size. If there is data remaining to be written a transition to `TWO_PHASE_SEND` is performed. Otherwise a transition to `TWO_PHASE_UPDATE` is done.

TWO_PHASE_SEND In this state all file domain owners have received the data to write from the other clients and start write operations on the involved servers. If there is no data to be written a transition to `TWO_PHASE` is performed. Otherwise a transition to `TWO_PHASE_WRITE` is done.

TWO_PHASE_WRITE In this state the servers are ready to receive the file data. Therefore, the actual file data is transferred to them. A transition to `TWO_PHASE` is performed.

TWO_PHASE_UPDATE This state is used to update the file size. Additionally, the meta information for this operation is cleaned up. A transition to `STEP_COMPLETED` is performed.

STEP_COMPLETED This state signals the end of the state machine. All data is written and the `Filewriteall` command is completed.

In the `Direct` implementation of the `Filewriteall` command each client internally uses individual I/O calls to write their data. This implementation is used when the `CHECK_TWO_PHASE` state determines that the Two-Phase protocol should not be used.

The states of the `Direct` state machine are as follows:

SEND_REQUEST In this state each client starts a write operation directly on the involved servers. A transition to RECV_ACK is performed.

RECV_ACK In this state the servers are ready to receive the file data. Each client transfers the file data directly to them. A transition to UPDATE_SIZE is performed.

UPDATE_SIZE This state is used to update the file size. A transition to STEP_-COMPLETED is performed.

5.3.2 Interleaved Two-Phase

The implementation of the Interleaved Two-Phase protocol is a slight modification of the original Two-Phase protocol's implementation as described in subsection 4.3.1 (page 35). In fact, the steps performed in the process() method are the same and therefore figures 5.1 (page 56) and 5.2 (page 57) also apply for the Interleaved Two-Phase protocol. For simplicity's sake only the differences between the two implementations are described here.

Read

TWO_PHASE This state is used to determine the file region to read. This is done by using the minimum and maximum offsets and reading everything in between. The resulting file region is then partitioned into equally sized chunks — the size of the chunks is determined by the Two-Phase buffer size. Each client is then responsible for exactly one chunk and reads it. All clients also tell other clients which data they need from their respective chunks. If there are chunks remaining to be read a transition to TWO_PHASE_RECV is performed. Otherwise a transition to STEP_COMPLETED is done and the meta information for this operation is cleaned up.

TWO_PHASE_RECV In this state clients have received the data from their respective chunks. Requested data is then distributed to all other clients, which requested it in the previous state. A transition to TWO_PHASE is performed.

Write

TWO_PHASE This state is used to determine the file region to write. This is done by using the minimum and maximum offsets. The resulting file region is then partitioned into equally sized chunks — the size of the chunks is determined by the Two-Phase buffer size. Each client then sends as much of their data as possible to the owners of the respective chunks. If there are chunks remaining to be written a transition to TWO_PHASE_SEND is performed. Otherwise a transition to TWO_PHASE_UPDATE is done.

`TWO_PHASE_SEND` In this state all chunk owners have received the data to write from the other clients and start write operations on the involved servers. If there is no data to be written a transition to `TWO_PHASE` is performed. Otherwise a transition to `TWO_PHASE_WRITE` is done.

Summary

PIOsimHD-Model and PIOsimHD-Simulator allow the implementation of arbitrary components and commands. Components are used to model real-life hardware components, while commands are used to model MPI commands. It is possible to provide multiple variant implementations for both components and commands. Server-Directed I/O is implemented as a cache layer in PIOsimHD-Model and PIOsimHD-Simulator. It supports delayed writes to allow more extensive optimization of outstanding I/O operations. The Two-Phase protocol as well as the Interleaved Two-Phase protocol are implemented as commands in PIOsimHD-Model and PIOsimHD-Simulator. They contain a framework to manage meta information for collective I/O.

Chapter 6

Evaluation

This chapter gives an overview of the used benchmarks — both synthetic and real-life — and presents the results obtained with them. The main goal is to determine how client-side and server-side optimizations compare and interact with each other. The synthetic benchmarks consist of tests using random I/O, individual I/O and collective I/O. Additionally, a test designed to simulate database access patterns is used. A parallel partial differential equation solver called `partdiff-par` is used as a real-life benchmark.

6.1 Environment

It is important to note that all results mentioned in this chapter are measured in a simulated cluster environment within PIOsimHD-Simulator. The following describes how this cluster environment is configured.

6.1.1 Cluster Configuration

All synthetic benchmarks — that is, the random I/O test, the individual I/O tests and the collective I/O tests — used in this evaluation use a standardized cluster configuration to be able to compare the different tests. For an overview of the components below refer to subsection 3.1.1 (page 22).

The cluster is made up of twenty `Nodes`. Ten of these `Nodes` are used as clients, while the other ten are used as servers. Each `Node` has one CPU, 1 000 MiB of RAM and a 1 GBits/s Ethernet `NIC`. The CPUs can calculate 1 000 000 IPS[1] and have an internal data transfer speed of 1 000 MiB/s. The `NICs` can transfer up to 100 MiB/s and have a latency of 200 µs. All `Nodes` are connected to one `Switch` with a maximum bandwidth of 1 000 MiB/s. Additionally, each `Node`'s `I/O Subsystem` consists of one HDD with a transfer rate of 50 MiB/s and 7 200 RPM. The `HDDs` have an average seek time of 10 ms and a track-to-track seek time of 1 ms. Whenever

[1]Instructions per Second

two subsequent accesses are not more than 5 MiB apart the HDD has to perform a track-to-track seek. Otherwise an average seek has to be performed.

The clients' data is striped across the servers with a simple round-robin scheme that works like RAID[2]-0. The striping size is set to 64 KiB, which corresponds to PVFS's default striping size.

The `Cache Layer` is varied to compare the `NoCache`, `SimpleWriteBehindCache`, `AggregationCache` and `ServerDirectedIO` cache layers. No `Cache Layer` performs any read-ahead or data sieving optimizations.

6.1.2 File Layout

All evaluated synthetic benchmarks — that is, the random I/O test, the individual I/O tests and the collective I/O tests — use a standardized file layout to be able to compare the different tests.

Figure 6.1 shows the file layout used for this evaluation. The file itself has a size of 1 000 MiB and is read or written by ten clients in iterations, advancing through the file with each iteration. Each iteration consists of ten data blocks of the same size. The size of the data blocks can be varied — the tests use data blocks of size 5 KiB, 50 KiB and 512 KiB. A data block size of 512 Bytes causes the simulation to take too long and is therefore omitted. Each data block of an iteration is read or written by a different client, as denoted by the client number within the data block. For example, in each iteration block 1 is written by client 1, block 2 is written by client 2, and so on. The number of iterations is determined by the data block size. Each client reads or writes one data block per iteration.

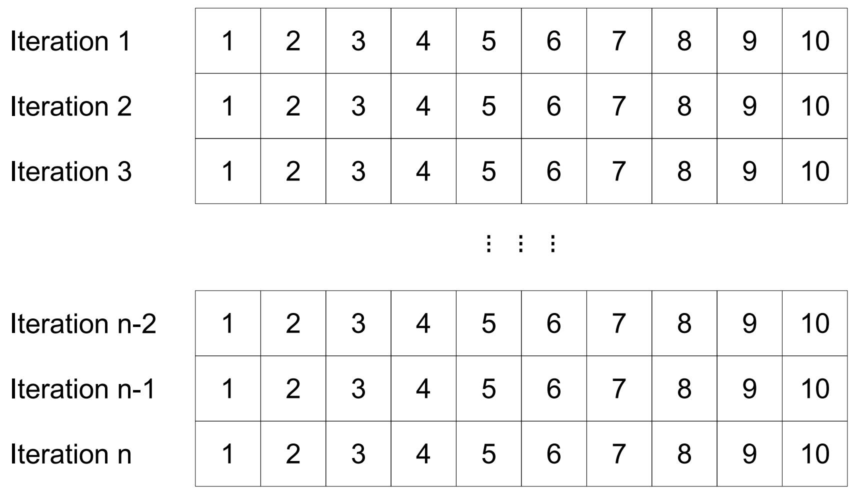


Figure 6.1: File Layout

[2] Redundant Array of Inexpensive Disks *or* Redundant Array of Independent Disks

6.2 Performance Considerations

The cluster consists of ten clients and ten servers, each capable of transferring 100 MiB/s. Since the switch has a throughput of 1 000 MiB/s the clients can communicate with the servers at full speed. However, the HDDs have a maximum speed of 50 MiB/s, which limits the I/O performance. Therefore, the clients can read or write at most 500 MiB/s from or to the file system served by the ten servers. The clients do not perform any computation, therefore this amount of throughput should be achievable. Additionally, all clients and servers have empty caches at the beginning of each test, that is, all data must be read from the underlying storage device. This means that read throughput can not exceed the above number.

The maximum throughput is lower when using the Two-Phase protocol, because it introduces additional communication overhead between the clients. This problem could be alleviated by using overlapping communication and I/O phases, but this is currently not implemented.

PIOsimHD-Simulator splits up write operations into blocks of 100 KiB, which can cause write performance to be worse than the read performance. This is due to the fact that this splitting up causes more — and smaller — operations to be performed when a non-optimizing cache layer is used. However, due to the fact that write operations can be cached for an arbitrary time and then processed in the background there is more room for optimization. Read operations need to be processed as fast as possible, because the clients need the requested data to continue operation. Therefore, when only performing a relatively small number of read operations in parallel read performance is usually much lower than write performance. When many read operations are batched together this gap between read and write performance should shrink.

For a more in-depth analysis of performance factors in distributed systems see [Kun07].

6.3 Random I/O Test

The random I/O test uses randomized individual I/O operations. Each client performs a sequence of contiguous read or write operations in a randomized order. To make the different runs of the tests comparable, the randomization is done with a pre-defined random seed — that is, each run yields the same result. The read and writes cases use the same random seeds.

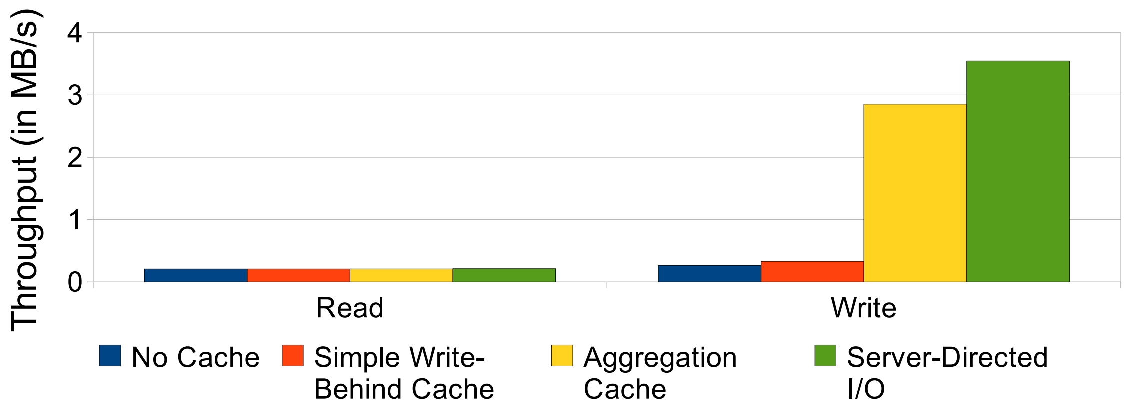
To obtain meaningful results ten runs with different random seeds are performed and the average values over these runs are used for comparison. The standard deviation is not shown in the figures, because it is too small.

Each figure shows per-client throughput, that is, the maximum throughput is 50 MiB/s.

Figure 6.2a shows the results of the random I/O test when run with a data block size of 5 KiB. In the read case, the `NoCache` and `SimpleWriteBehindCache` cache layers have the same performance, because they do not perform any read optimizations. The same is true for all the following tests. The `AggregationCache` cache layer also delivers the same performance, even though it performs read optimizations. The `ServerDirectedIO` cache layer is slightly faster. In the write case, the `SimpleWriteBehindCache` cache layer increases the throughput slightly. The `AggregationCache` and `ServerDirectedIO` cache layers increase throughput by factors of 11 and 14 respectively.

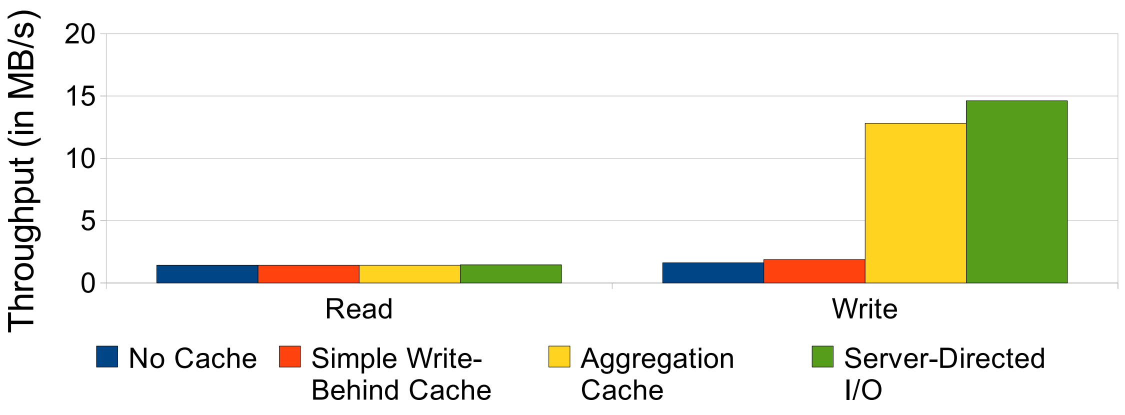
Figure 6.2b (page 65) shows the results of the random I/O test when run with a data block size of 50 KiB. In the read case, the `ServerDirectedIO` cache layer again increases performance slightly. In the write case, the `SimpleWriteBehind-Cache` cache layer increases throughput slightly when compared to the `NoCache` cache layer. The `AggregationCache` and `ServerDirectedIO` cache layers increase throughput by factors of 8 and 9 respectively.

Figure 6.2c (page 65) shows the results of the random I/O test when run with a data block size of 512 KiB. In the read case, the `ServerDirectedIO` cache layer again increases performance slightly. In the write case, the `SimpleWriteBehindCache` cache layer does not improve performance at all when compared to the `NoCache` cache layer. The `AggregationCache` cache layer increases throughput by a factor of 6 while the `ServerDirectedIO` cache layer increases it by a factor of 7.

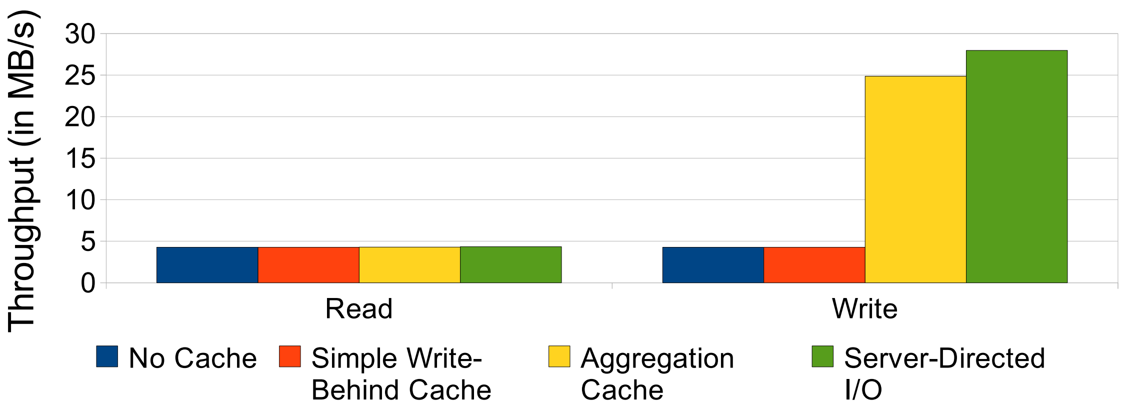


(a) 5 KiB Data Blocks

Figure 6.2: Random I/O Test



(b) 50 KiB Data Blocks



(c) 512 KiB Data Blocks

Figure 6.2: Random I/O Test

Summary

Overall, the benefits provided by the read optimizations performed by the `AggregationCache` and `ServerDirectedIO` cache layers are limited. This is due to the fact that at most ten operations can be combined per iteration. This happens very rarely, because the accesses are spread randomly across the file. However, the `ServerDirectedIO` cache layer provides slight performance improvements for all block sizes. Improving and enabling the cache layer's data sieving algorithm could provide significant improvements for random read operations.

In the write case, the `SimpleWriteBehindCache` cache layer provides a slight performance boost except for the largest data block size where it performs the same as the `NoCache` cache layer. Both the `AggregationCache` and `ServerDirectedIO` cache layers provide substantial performance improvements, regardless of the data block size. However, a better speedup is achieved for small block sizes. The

ServerDirectedIO cache layer's performance is always slightly better than the AggregationCache cache layer's performance.

The results look very much alike, regardless of the data block size. This is due to the randomness of the test, which allows almost no optimizations of read operations. Even though the write operations can be optimized better no test reaches the maximum throughput of 50 MiB/s.

6.4 Individual I/O Test

The individual I/O test uses individual I/O operations, that is, each client reads or writes its data blocks in an ascending order. A varying number of data blocks are written with one operation. The contiguous test reads or writes only one data block at a time, while the non-contiguous ones read or write multiple data blocks per operation.

6.4.1 1 Data Block per Iteration

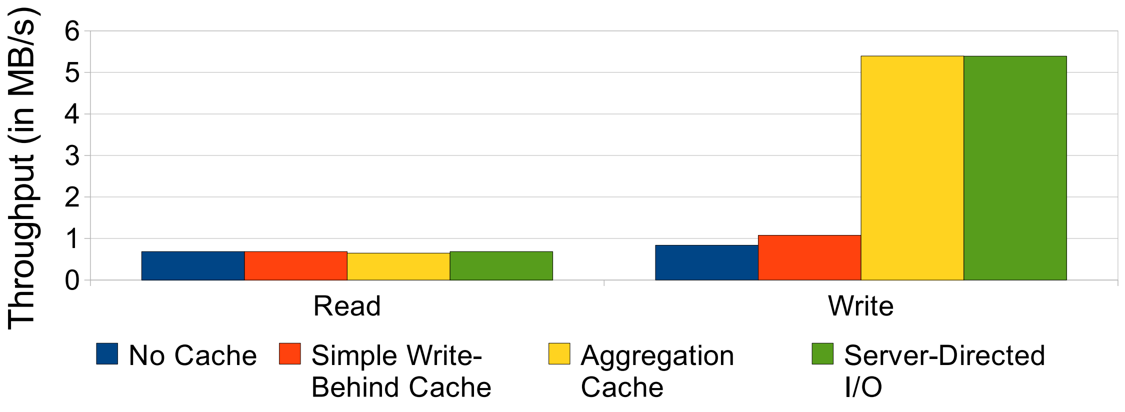
This test uses contiguous I/O, that is, only one data block is read or written per iteration.

Figure 6.3a (page 67) shows the results of the individual I/O test when run with a data block size of 5 KiB. In the read case, the AggregationCache cache layer decreases performance slightly, while the ServerDirectedIO cache layer delivers the same performance as the NoCache and SimpleWriteBehindCache cache layers. This is due to the fact that — because of the striping size — only one server is processing requests at any time. Additionally, the small data block size causes some operations to be finished before the other clients' operations arrive, which makes merging operations impossible. In the write case, the SimpleWriteBehindCache cache layer increases the throughput slightly. The AggregationCache and ServerDirectedIO cache layers increase throughput by a factor of 7.

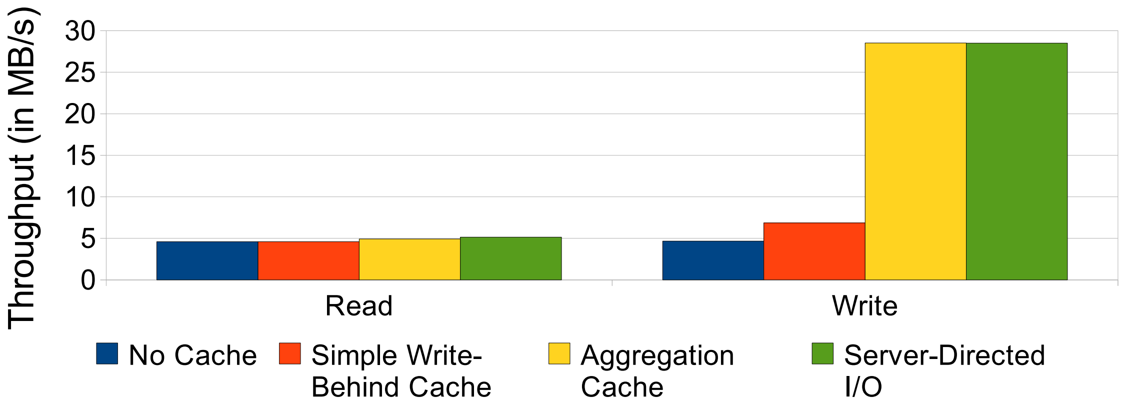
Figure 6.3b (page 67) shows the results of the individual I/O test when run with a data block size of 50 KiB. In the read case, both the AggregationCache and ServerDirectedIO cache layers increase performance slightly. This is due to the fact that only one operation is performed on each server, which can not be optimized significantly. In the write case, the SimpleWriteBehindCache cache layer increases throughput only slightly, while the AggregationCache and ServerDirectedIO cache layers increase it by a factor of 6.

Figure 6.3c (page 68) shows the results of the individual I/O test when run with a data block size of 512 KiB. In the read case, the AggregationCache and ServerDirectedIO cache layers increase performance tremendously. The Aggregation-

Cache cache layer increases throughput by factor of 2, while the ServerDirectedIO cache layer increases it by a factor of 5. This is due to the fact that the larger data block size causes more operations to be performed on each server, which can then be optimized. In the write case, the SimpleWriteBehindCache cache layer offers less improvement than in the previous tests. Both the AggregationCache and ServerDirectedIO cache layers increases throughput by a factor of 5.

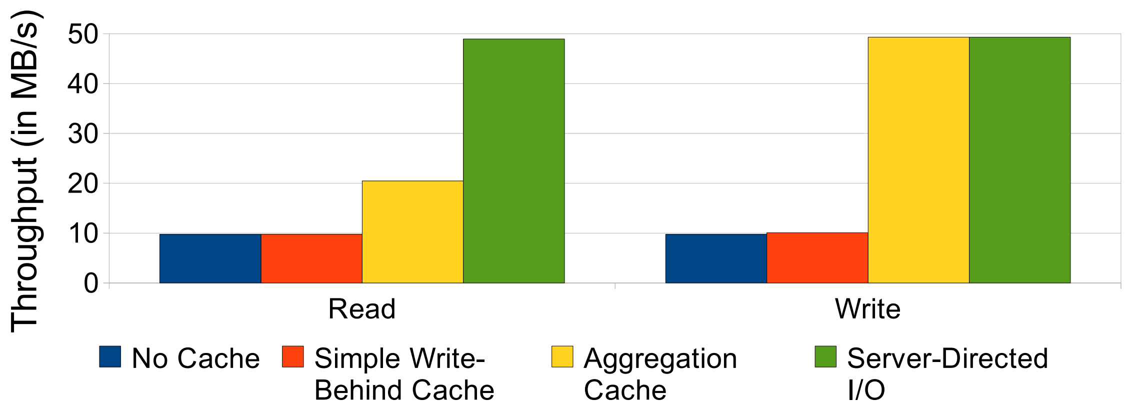


(a) 5 KiB Data Blocks



(b) 50 KiB Data Blocks

Figure 6.3: Individual I/O Test — 1 Data Block per Iteration



(c) 512 KiB Data Blocks

Figure 6.3: Individual I/O Test — 1 Data Block per Iteration

Summary

Overall, the benefits provided by the read optimizations performed by the `AggregationCache` and `ServerDirectedIO` cache layers are very dependent on the data block size. There are almost no gains for the smaller data block sizes of 5 and 50 KiB, while a data block size of 512 KiB makes possible tremendous speedups. Preliminary results show that more speedup is also possible for smaller data block sizes when using 100 clients. This is due to the fact that this also increases the amount of concurrent operations, which allows better optimizations to be performed.

In the write case, the `SimpleWriteBehindCache` cache layer provides only a slight performance boost. Both the `AggregationCache` and `ServerDirectedIO` cache layers provide substantial performance improvements, regardless of the data block size.

The `ServerDirectedIO` cache layer almost reaches the maximum throughput of 50 MiB/s for both read and write operations when using a data block size of 512 KiB.

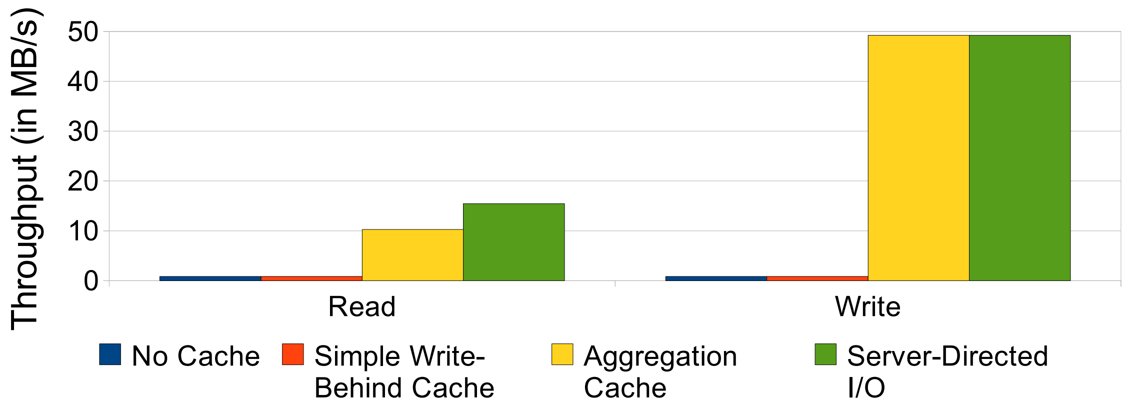
6.4.2 100 Data Blocks per Iteration

This test uses non-contiguous I/O, that is, 100 data blocks are read or written per iteration. This gives more I/O-related information to the cache layers than in the previous test and allows them to optimize accesses better.

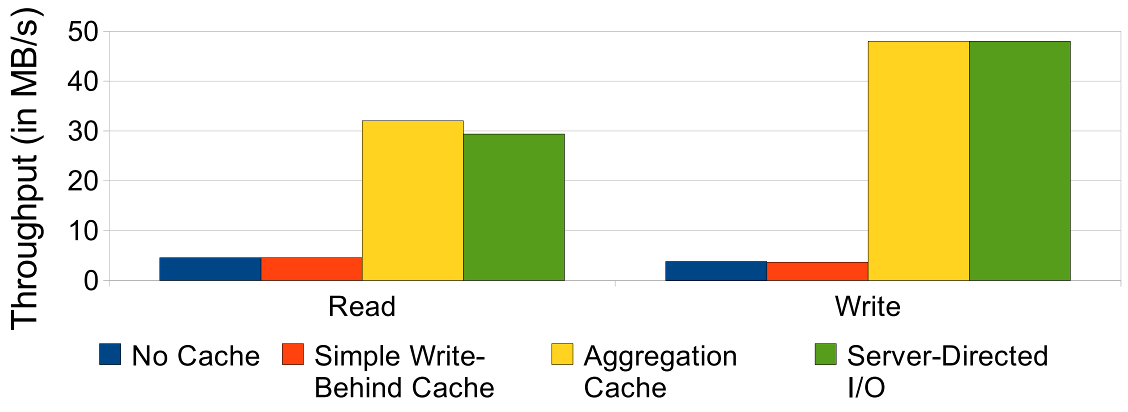
Figure 6.4a (page 69) shows the results of the individual I/O test when run with a data block size of 5 KiB. In the read case, the `AggregationCache` cache layer improves performance by a factor of 12, while the `ServerDirectedIO` cache layer improves it by a factor of 18. In the write case, both the AggregationCache and ServerDirectedIO cache layers increase throughput by a factor of 58.

Figure 6.4b shows the results of the individual I/O test when run with a data block size of 50 KiB. In the read case, the AggregationCache cache layer increases performance by a factor of 7, while the ServerDirectedIO cache layer performs slightly worse. In the write case, the AggregationCache and ServerDirectedIO cache layers increase throughput by a factor of 11.

Figure 6.4c (page 70) shows the results of the individual I/O test when run with a data block size of 512 KiB. In the read case, the AggregationCache cache layer increases performance by a factor of 5, while the ServerDirectedIO cache layer performs slightly worse. In the write case, the AggregationCache and ServerDirectedIO cache layers increase throughput by a factor of 7.5.

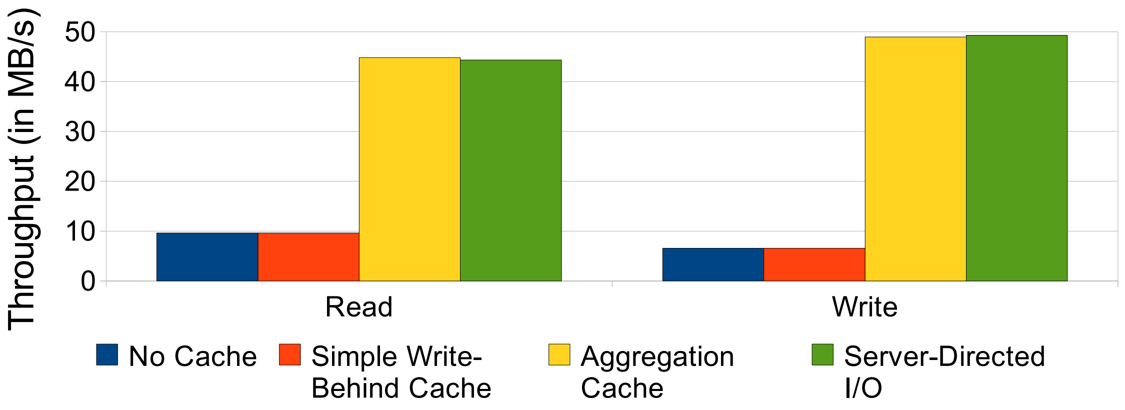


(a) 5 KiB Data Blocks



(b) 50 KiB Data Blocks

Figure 6.4: Individual I/O Test — 100 Data Blocks per Iteration



(c) 512 KiB Data Blocks

Figure 6.4: Individual I/O Test — 100 Data Blocks per Iteration

Summary

Overall, the benefits provided by the read optimizations performed by the `AggregationCache` and `ServerDirectedIO` cache layers are very dependent on the data block size. However, due to the batching of 100 operations the results are 5 to 15 times better than in the contiguous case. The only exception is the test with a data block size of 512 KiB/s.

In the write case, the `SimpleWriteBehindCache` cache layer provides no performance boost at all. Both the `AggregationCache` and `ServerDirectedIO` cache layers provide substantial performance improvements, regardless of the data block size.

Both the `AggregationCache` and `ServerDirectedIO` cache layers reach the maximum throughput of 50 MiB/s for write operations regardless of the data block size.

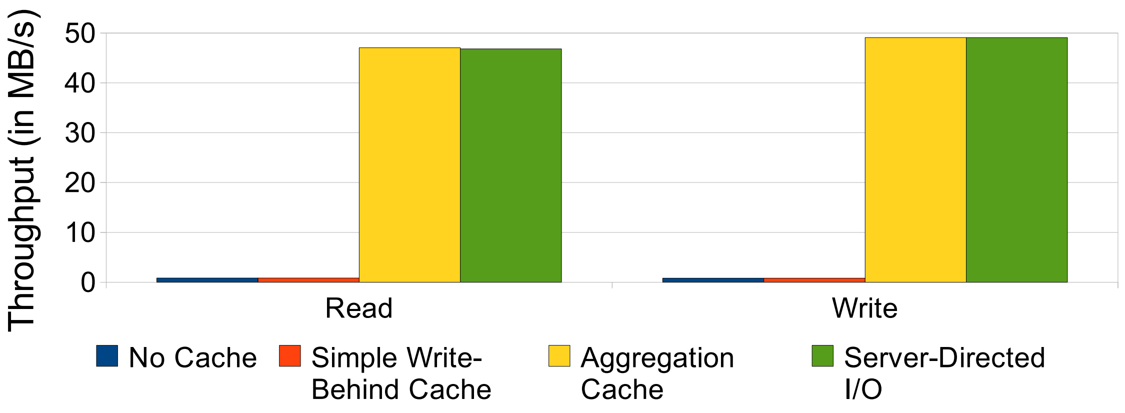
6.4.3 All Data Blocks per Iteration

This test uses non-contiguous I/O, that is, all data blocks are read or written in one iteration. This gives all I/O-related information to the cache layers and allows them to optimize accesses better.

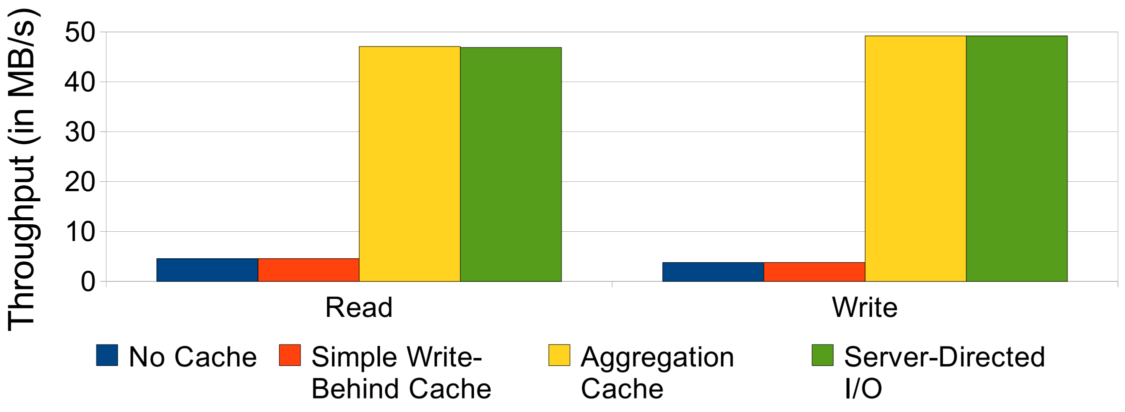
Figure 6.5a (page 71) shows the results of the individual I/O test when run with a data block size of 5 KiB. In the read case, both the `AggregationCache` and `ServerDirectedIO` cache layers improve performance by a factor of 55. The `ServerDirectedIO` cache layer performs slightly worse than the `AggregationCache` one. In the write case, both the `AggregationCache` and `ServerDirectedIO` cache layers increase throughput by a factor of 60.

Figure 6.5b (page 71) shows the results of the individual I/O test when run with a data block size of 50 KiB. In the read case, both the `AggregationCache` and `ServerDirectedIO` cache layers improve performance by a factor of 10.5. The `ServerDirectedIO` cache layer performs slightly worse than the `AggregationCache` one. In the write case, both the `AggregationCache` and `ServerDirectedIO` cache layers increase throughput by a factor of 13.

Figure 6.5c (page 72) shows the results of the individual I/O test when run with a data block size of 512 KiB. In the read case, both the `AggregationCache` and `ServerDirectedIO` cache layers improve performance by a factor of 5. The `ServerDirectedIO` cache layer performs slightly worse than the `AggregationCache` one. In the write case, both the `AggregationCache` and `ServerDirectedIO` cache layers increase throughput by a factor of 7.5.

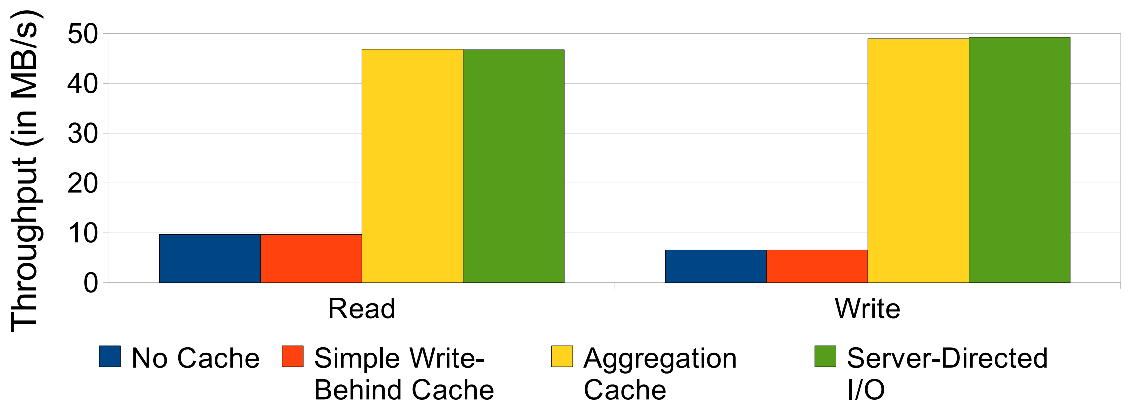


(a) 5 KiB Data Blocks



(b) 50 KiB Data Blocks

Figure 6.5: Individual I/O Test — All Data Blocks per Iteration



(c) 512 KiB Data Blocks

Figure 6.5: Individual I/O Test — All Data Blocks per Iteration

Summary

In the read case, both the `AggregationCache` and `ServerDirectedIO` cache layers provide substantial performance improvements, regardless of the data block size.

In the write case, the `SimpleWriteBehindCache` cache layer provides no performance boost at all. Both the `AggregationCache` and `ServerDirectedIO` cache layers provide substantial performance improvements, regardless of the data block size.

Both the `AggregationCache` and `ServerDirectedIO` cache layers reach the maximum throughput of 50 MiB/s for write operations regardless of the data block size. For read operations the throughput is slightly worse, but still unaffected by the data block size.

6.5 Collective I/O Test

The collective I/O test uses collective I/O operations. Each clients performs only one collective read or write operation, accessing all data blocks at once. It is done this way to ensure that the Two-Phase algorithm needs to do actual work.

6.5.1 Two-Phase

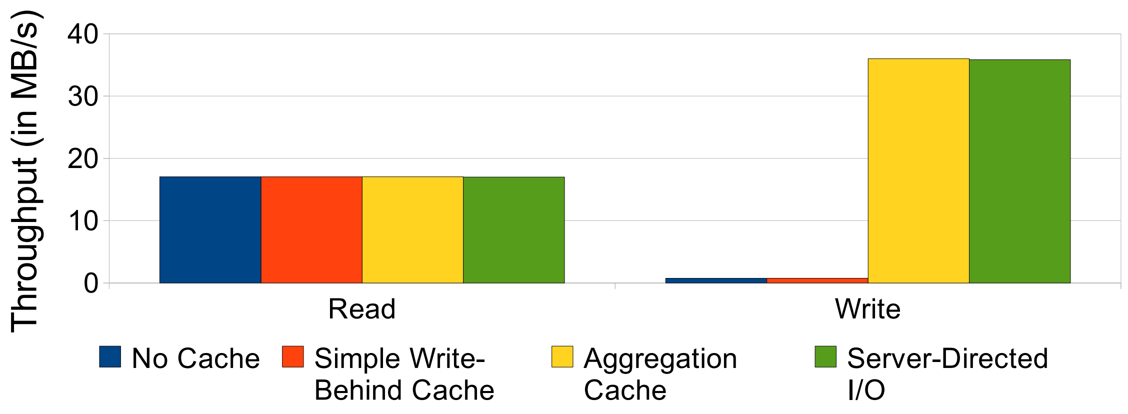
Due to each client performing only one collective read or write operation, each client is responsible for a contiguous region of data blocks. In the read case each clients reads its data blocks from disk — or rather as many data blocks as fit into the Two-Phase buffer — and then sends the data blocks to the corresponding clients. In the write case all clients send their data blocks to the corresponding client which in turn writes them to disk.

The main question is which throughput the collective I/O operations can deliver with optimizing cache layers — that is, the `AggregationCache` and `ServerDirectedIO` cache layers.

Figure 6.6a shows the results of the collective I/O test when run with a data block size of 5 KiB using the original Two-Phase protocol. In the read case, all cache layers deliver the same performance, except for the `ServerDirectedIO` cache layer, which performs slightly worse. They reach a throughput of about 17 MiB/s. In the write case, both the `AggregationCache` and `ServerDirectedIO` cache layers do not reach the maximum throughput of 50 MiB/s. This is due to the fact that the Two-Phase protocol introduces additional network overhead. The `ServerDirectedIO` cache layer performs slightly worse than the `AggregationCache` one. Both reach a throughput of about 36 MiB/s.

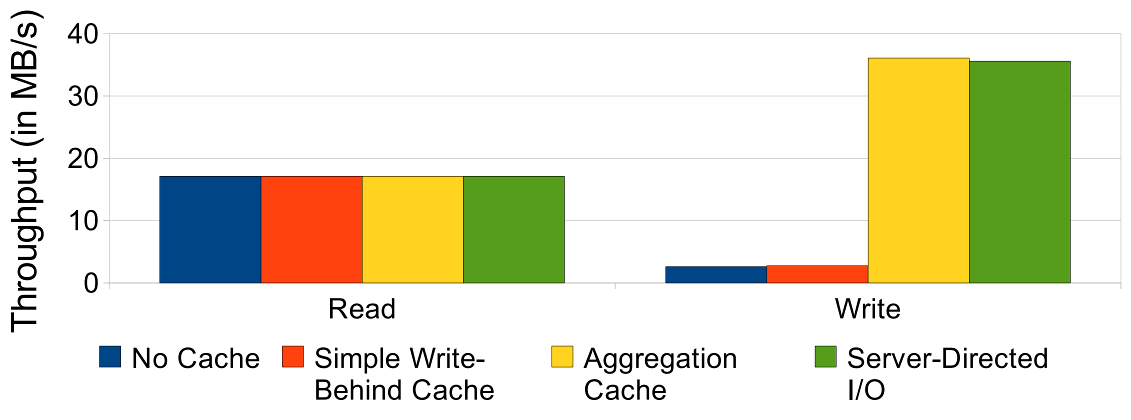
Figure 6.6b (page 74) shows the results of the collective I/O test when run with a data block size of 50 KiB using the original Two-Phase protocol. The figure basically looks like the previous one, with the exception of increased write performance for the non-optimizing cache layers. This is due to the larger data block size. However, the `ServerDirectedIO` cache layer performs worse than in figure 6.6a.

Figure 6.6c (page 74) shows the results of the collective I/O test when run with a data block size of 512 KiB using the original Two-Phase protocol. The figure basically looks like the previous one, with the exception of increased write performance for the non-optimizing cache layers. This is due to the larger data block size. However, the `ServerDirectedIO` cache layer performs worse than in figure 6.6b (page 74).

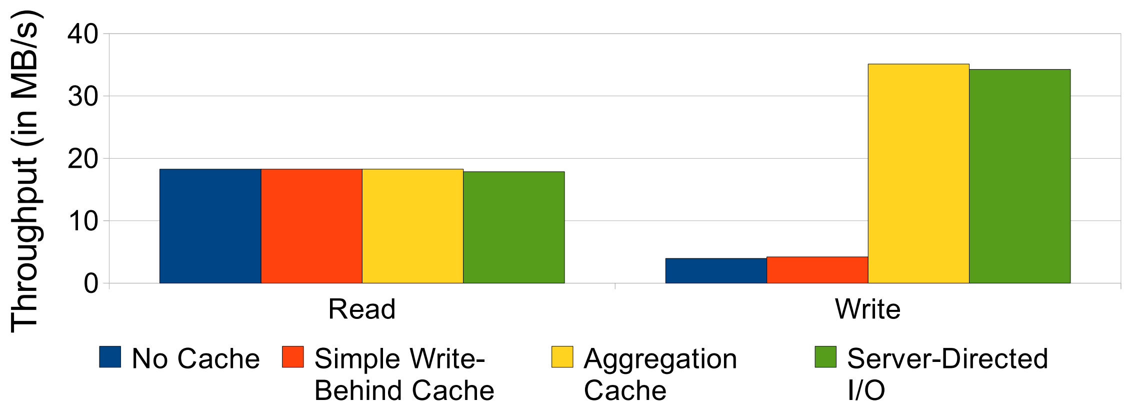


(a) 5 KiB Data Blocks

Figure 6.6: Two-Phase I/O Test



(b) 50 KiB Data Blocks



(c) 512 KiB Data Blocks

Figure 6.6: Two-Phase I/O Test

Summary

In the read case, all cache layers provide the same performance. The throughput is about 16.5 MiB/s, which is significantly worse and not even half the performance delivered in the individual I/O test.

In the write case, the performance of the non-optimizing cache layers — that is, the `NoCache` and `SimpleWriteBehindCache` cache layers — is not as good as it could be, because the current Two-Phase implementation issues too many separate I/O requests instead of combining them. Both the `AggregationCache` and `SimpleWriteBehindCache` cache layers provide a throughput of about 35 MiB/s. This is higher than the throughput in the read case, because the servers perform write-behind. This allows overlapping the Two-Phase communication with actual I/O on the servers. However, the maximum throughput of 50 MiB/s is still not reached.

6.5.2 Interleaved Two-Phase

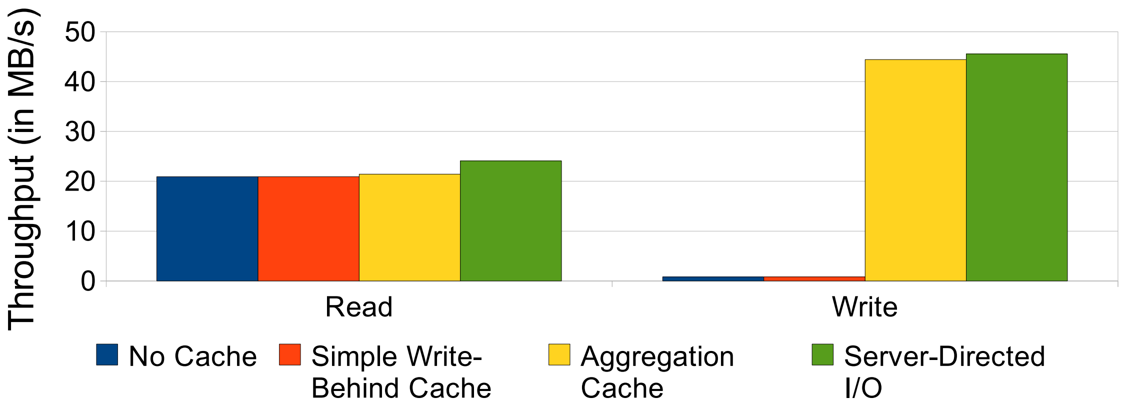
This test uses the Interleaved Two-Phase protocol. In the read case each clients reads as many data blocks as fit into the Two-Phase buffer and then sends the data blocks to the corresponding clients. In the write case all clients send their data blocks to the corresponding client which in turn writes them to disk.

The main question is how the Interleaved Two-Phase protocol compares to the original Two-Phase protocol and whether any performance gains are possible.

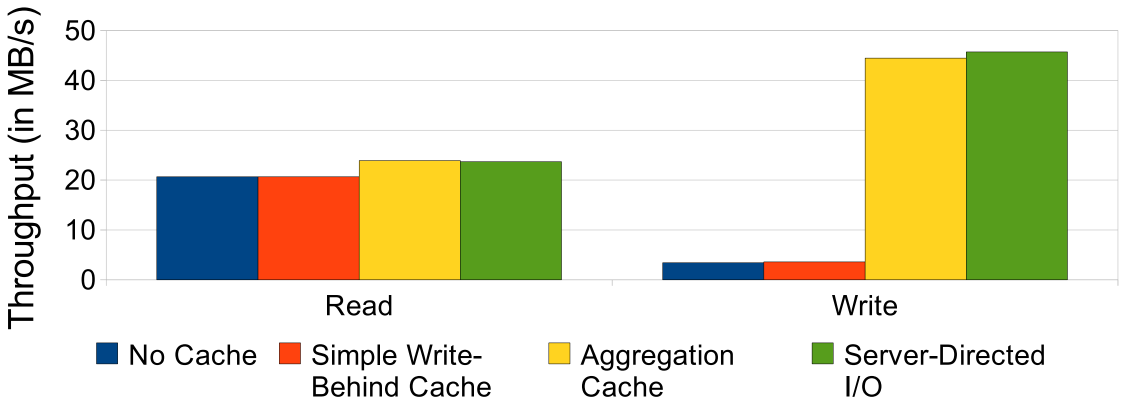
Figure 6.7a (page 76) shows the results of the collective I/O test when run with a data block size of 5 KiB using the Interleaved Two-Phase protocol. In the read case, both the `AggregationCache` and `ServerDirectedIO` cache layers improve performance slightly. This is due to the fact that the Interleaved Two-Phase protocol uses a contiguous access pattern which can be optimized better. They reach throughputs of about 21.5 MiB/s and 24 MiB/s respectively. In the write case, both the `AggregationCache` and `ServerDirectedIO` cache layers do not reach the maximum throughput of 50 MiB/s, if even by a small amount. This is due to the fact that the Two-Phase protocol introduces additional network overhead. The `ServerDirectedIO` cache layer performs slightly better than the `AggregationCache` one. Both reach a throughput of about 45 MiB/s.

Figure 6.7b (page 76) shows the results of the collective I/O test when run with a data block size of 50 KiB using the Interleaved Two-Phase protocol. The figure basically looks like the previous one, with the exception of increased write performance for the non-optimizing cache layers. This is due to the larger data block size. However, in the read case, the `ServerDirectedIO` cache layer performs slightly worse than the `AggregationCache` this time.

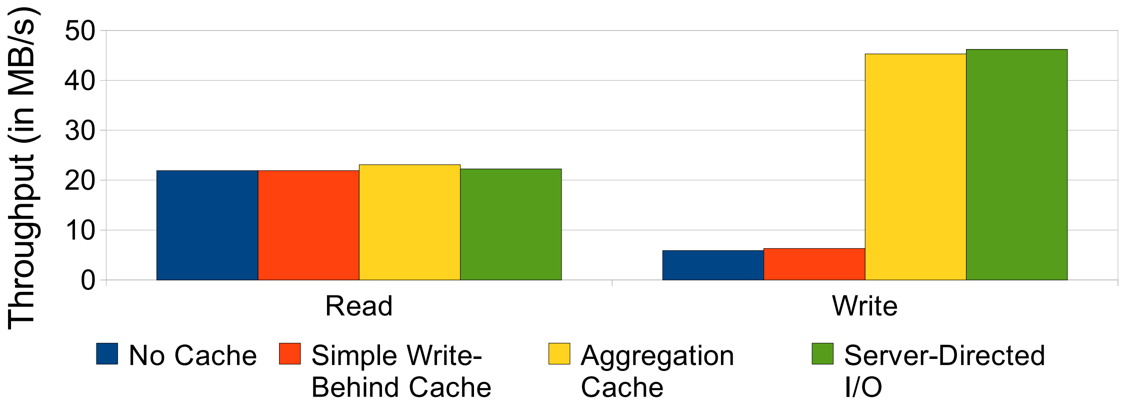
Figure 6.7c (page 76) shows the results of the collective I/O test when run with a data block size of 512 KiB using the Interleaved Two-Phase protocol. The figure basically looks like the previous one, with the exception of increased write performance for the non-optimizing cache layers. This is due to the larger data block size. Additionally, both the `AggregationCache` and `ServerDirectedIO` cache layers' read performance is worse than in the previous cases.



(a) 5 KiB Data Blocks



(b) 50 KiB Data Blocks



(c) 512 KiB Data Blocks

Figure 6.7: Interleaved Two-Phase I/O Test

Summary

In the read case, slight performance improvements are possible with both the `AggregationCache` and `ServerDirectedIO` cache layers. The throughput is about 24 MiB/s, which is significantly worse and about half the performance delivered in the individual I/O test. This results is better than the one obtained with the original Two-Phase protocol.

In the write case, the Interleaved Two-Phase protocol suffers from the same problem as the original one when using non-optimizing cache layers. Both the `AggregationCache` and `SimpleWriteBehindCache` cache layers provide a throughput of about 45 MiB/s. This is higher than the throughput in the read case, because the servers perform write-behind. This allows overlapping the Two-Phase communication with actual I/O on the servers. Even though the maximum throughput of 50 MiB/s is still not reached this result is better than the one obtained with the original Two-Phase protocol.

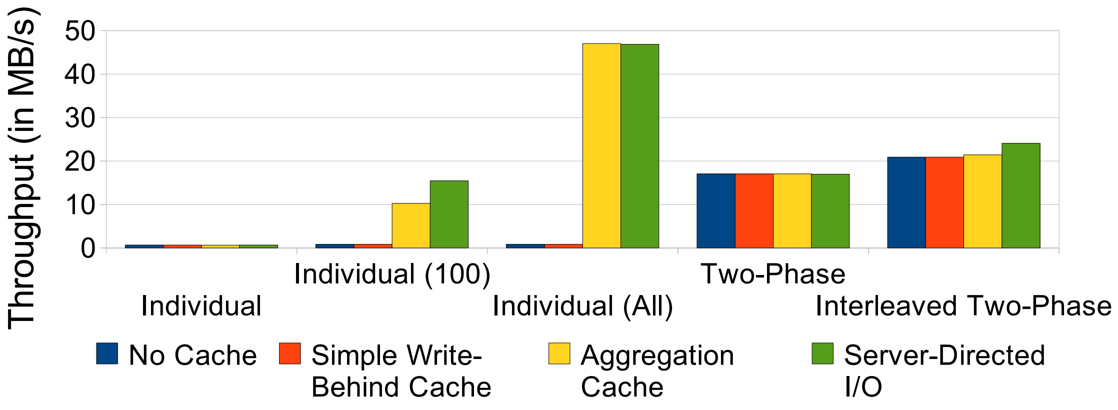
The Interleaved Two-Phase protocol delivers a better performance than the original one in all cases. This is due to the changes done to the access pattern used internally. Since the Interleaved Two-Phase protocol keeps accesses contiguous the cache layers can optimize the accesses better. For a visualization of the inner workings of the Interleaved Two-Phase protocol refer to section 7.2 (page 92).

6.6 Individual and Collective I/O Comparison

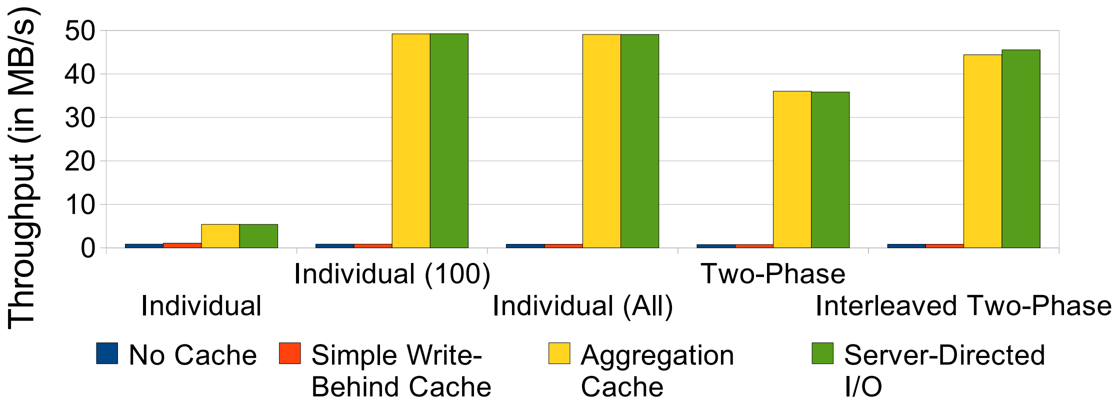
The following figures give an overview of the results obtained so far. The results are summarizes and compared with each other.

Figure 6.8a (page 78) shows all results obtained for read operations with a data block size of 5 KiB. As can be seen, performance suffers when using non-optimizing cache layers or too few operations per iteration. Batching operations results in performance gains. However, the maximum performance is only obtained when batching all operations. This is due to the fact that read operations must be processed when the clients request them and can not be postponed. Individual I/O operations with the `AggregationCache` and `ServerDirectedIO` cache layers beat both Two-Phase implementations.

Figure 6.8b (page 78) shows all results obtained for write operations with a data block size of 5 KiB. As can be seen, performance suffers when using non-optimizing cache layers or too few operations per iteration. In contrast to read operations less batching is required to obtain the maximum performance. This is due to the fact that write operations can be postponed and processed in the background. The Two-Phase implementations do not perform well when used with non-optimizing cache layers, because write operations are not performed in large contiguous regions internally — in contrast to the read case. However, this is a minor implementation detail and even using the optimizing cache layers delivers worse performance than the test with individual I/O operations.



(a) Read Tests

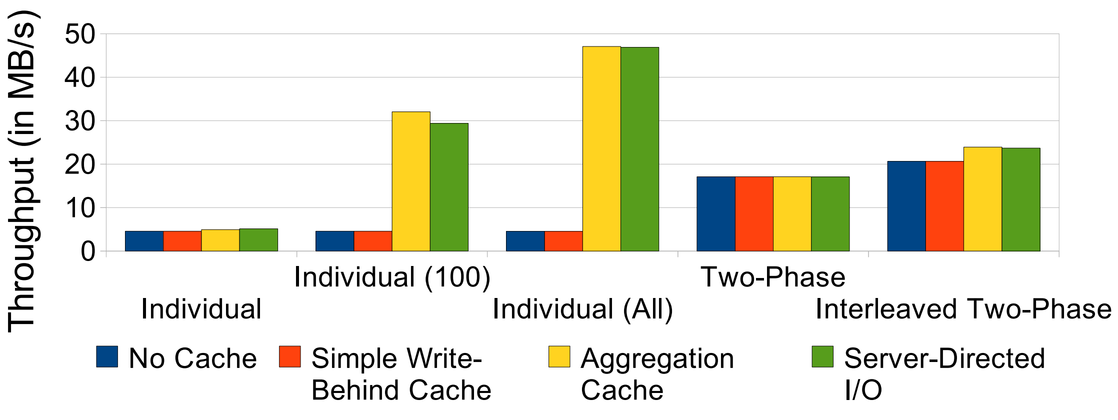


(b) Write Tests

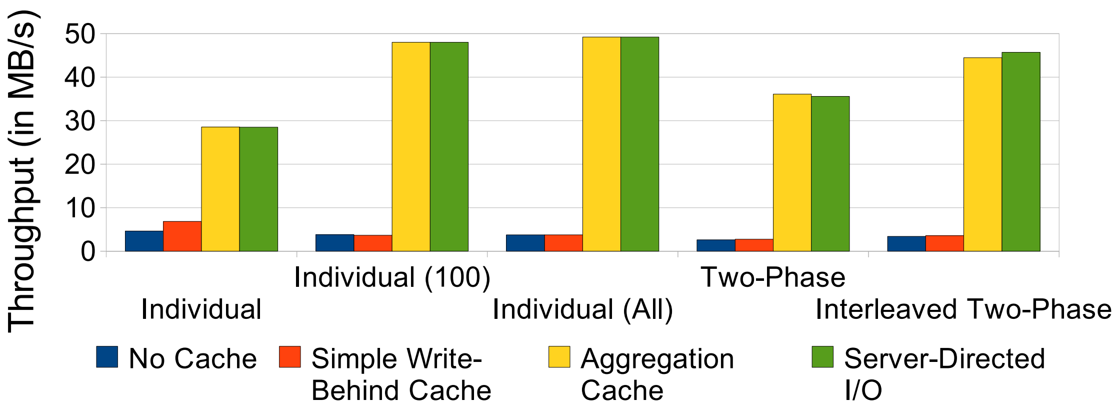
Figure 6.8: Summary — 5 KiB Data Blocks

Figure 6.9a (page 79) shows all results obtained for read operations with a data block size of 50 KiB. The results look like the ones in figure 6.8a except for the fact that better performance is achieved when using non-optimizing cache layers. This is due to the larger data block size.

Figure 6.9b (page 79) shows all results obtained for write operations with a data block size of 50 KiB. The results look like the ones in figure 6.8b except for the fact that better performance is achieved when using non-optimizing cache layers. This is due to the larger data block size.



(a) Read Tests

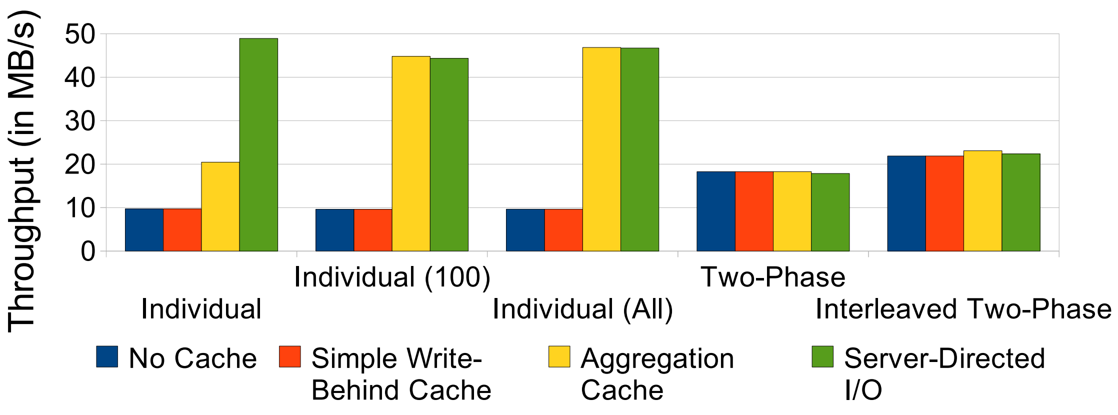


(b) Write Tests

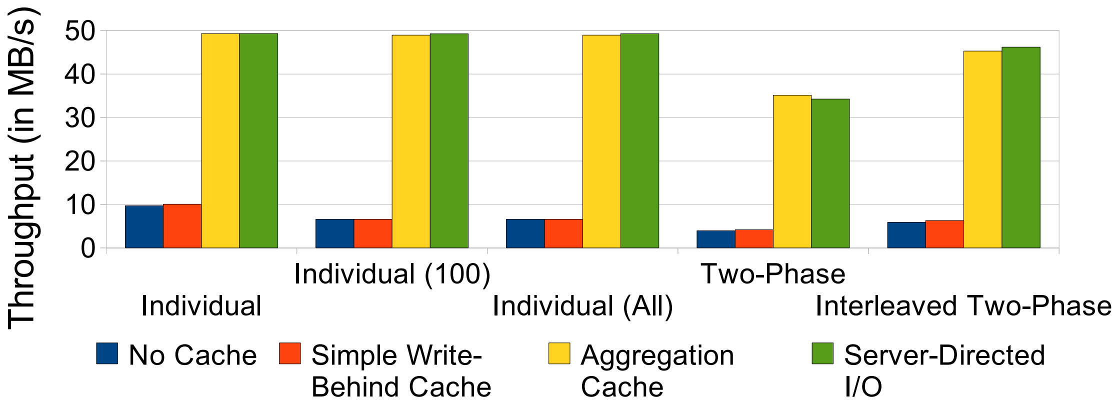
Figure 6.9: Summary — 50 KiB Data Blocks

Figure 6.10a (page 80) shows all results obtained for read operations with a data block size of 512 KiB. The results look like the ones in figure 6.9a except for the higher performance with non-optimizing cache layers, which is due to the larger data block size. The `ServerDirectedIO` cache layer almost delivers the maximum performance in all tests using individual I/O operations.

Figure 6.10b (page 80) shows all results obtained for write operations with a data block size of 512 KiB. The results look like the ones in figure 6.9b except for the higher performance with non-optimizing cache layers, which is due to the larger data block size. The `ServerDirectedIO` cache layer delivers the maximum performance in all tests using individual I/O operations.



(a) Read Tests



(b) Write Tests

Figure 6.10: Summary — 512 KiB Data Blocks

In conclusion, it is easy to see that the server-side optimizations deliver better performance than the client-side optimizations in all cases. Apart from that, the Interleaved Two-Phase protocol clearly outperforms its unmodified version, if even by a relatively small amount. However, to obtain satisfactory performance it is necessary to batch operations or use large operations. Batching operations is only necessary when using small operations, while large operations deliver good performance even when performed individually.

6.7 Database I/O Test

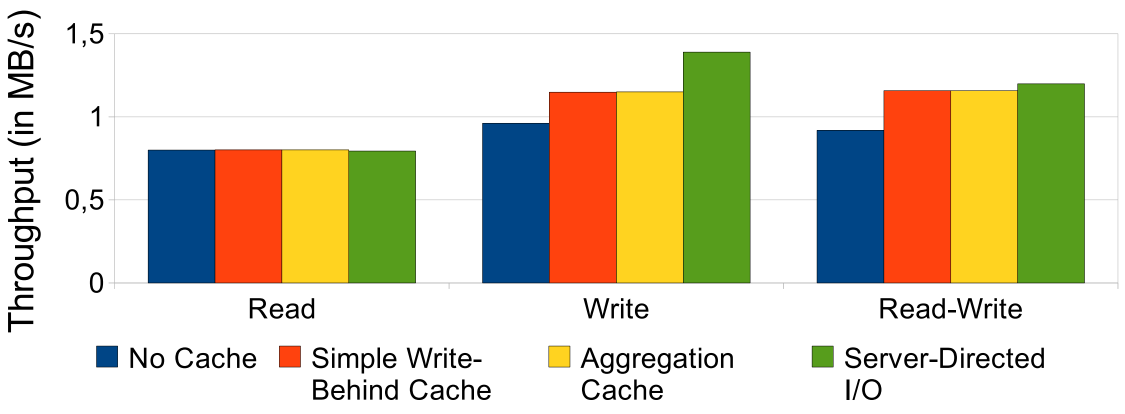
The database I/O tests simulates the load concurrent clients put on a database. It does not use the file layout specified in 6.1.2 (page 62). The test uses ten files that are each 1 250 000 KiB in size. Each client performs ten iterations and uses a data block size of 512 Bytes. In each iteration 1 250 KiB of data are accessed. In total, the clients access 125 000 KiB of data, that is, 1% of the total amount of data. Accesses are performed in blocks. Each block starts at a random offset and has a random size of at most 50 KiB. To make the different runs of the tests comparable, the randomization is done with a pre-defined random seed. All cases use the same random seeds. The database I/O test consists of cases for read-only, write-only and read-write accesses.

To obtain meaningful results ten runs with different random seeds are performed and the average values over these runs are used for comparison. The standard deviation is not shown in the figures, because it is too small.

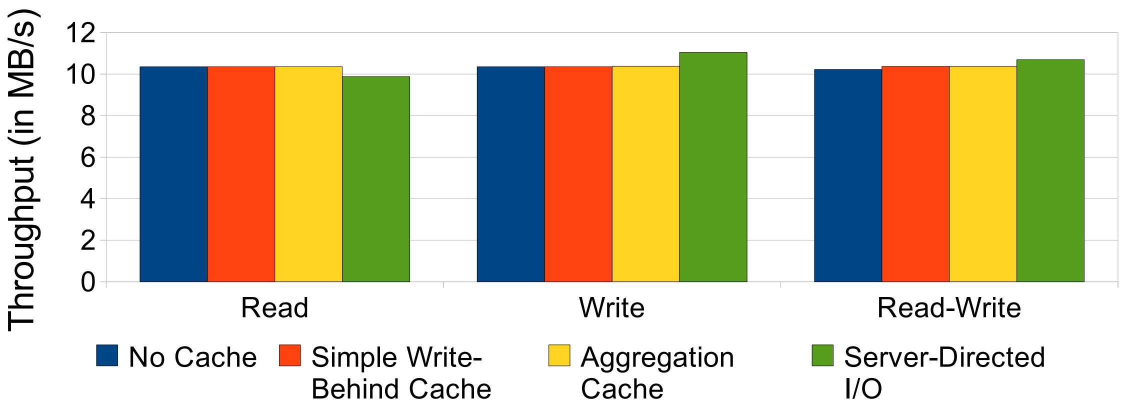
Figure 6.11a (page 82) shows the results of the database I/O test when run with a data block size of 512 Bytes using contiguous I/O operations. The `AggregationCache` cache layer also delivers the same performance, even though it performs read optimizations. The `ServerDirectedIO` cache layer is slightly slower. In the write and read-write cases, the `SimpleWriteBehindCache` and `AggregationCache` cache layers increase the throughput slightly. The `ServerDirectedIO` cache layer increases throughput further, if even by a small amount in the read-write case.

Figure 6.11b (page 82) shows the results of the random I/O test when run with a data block size of 512 Bytes using non-contiguous I/O operations to batch ten operations. In the read case, the `ServerDirectedIO` cache layer again decreases performance slightly. In the write and read-write cases, the `SimpleWriteBehindCache` and `AggregationCache` cache layers increase throughput slightly. The `ServerDirectedIO` cache layer increases throughput further.

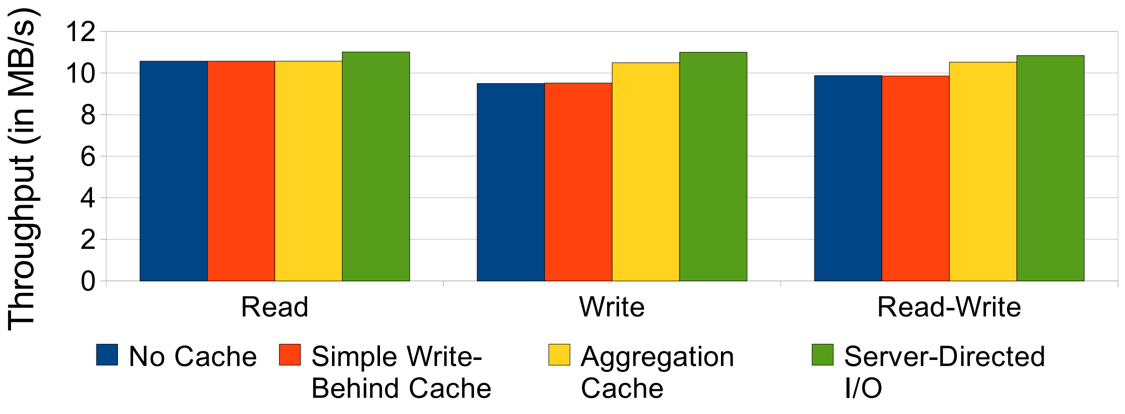
Figure 6.11c (page 82) shows the results of the database I/O test when run with a data block size of 512 Bytes using non-contiguous I/O operations to batch all operations. In the read case, only the `ServerDirectedIO` cache layer again increases performance slightly. In the write case, the `SimpleWriteBehindCache` cache layer does not improve performance at all when compared to the `NoCache` cache layer. The `AggregationCache` and `ServerDirectedIO` cache layers both increase throughput slightly with the `ServerDirectedIO` cache layer being somewhat faster.



(a) 1 Data Block per Iteration



(b) 10 Data Blocks per Iteration



(c) All Data Blocks per Iteration

Figure 6.11: Database I/O Test

Summary

Overall, the benefits provided by the read optimizations performed by the `AggregationCache` and `ServerDirectedIO` cache layers are limited. Batching operations does not significantly change this. This is due to the fact that the read operations are spread randomly across the file.

In the write and read-write cases, the `SimpleWriteBehindCache` cache layer provides a slight performance boost except when all operations are performed as a single non-contiguous I/O operation. Both the `AggregationCache` and `ServerDirectedIO` cache layers provide substantial performance improvements. The `ServerDirectedIO` cache layer's performance is always slightly better than the `AggregationCache` cache layer's performance.

6.8 Parallel Partial Differential Equation Solver

`partdiff-par` is a parallel partial differential equation solver using the Jacobi method. It uses MPI to parallelize the computation. Additionally, MPI-IO is used to write periodic checkpoints, which allow resuming the computation. A visualization file can be used to observe the convergence behavior during the application's runtime. Consequently, this application only performs write operations.

The `partdiff-par` application is used as a real-life benchmark to evaluate the different cache layers. For this purpose it was traced using the PIOsimHD framework using the included versions of MPICH2 and PVFS. The evaluation is done with five application processes and five file servers. The real application has a runtime of about 3.99 s. However, this runtime can not be directly compared to the simulated results, because of differences between the real and simulated hardware environment.

The results below show the application's runtime with varying cache layers. Additionally, internal statistics of the cache layers are presented. For example, the total number of I/O operations (*Accesses*) is shown. These operations are split up into operations that do not require any seeks at all (*No Seek*), operations that only require fast seeks — that is, track-to-track seeks — (*Fast*) and operations that do require slow seeks — that is, average seeks — (*Slow*). The total amount of data accessed per server is also shown.

6.8.1 No Cache

Table 6.1 (page 84) shows the results obtained when using no cache layer. Apart from the application's runtime one of the most interesting statistics is the number of I/O operations. Since this test uses no cache layer at all, no merging of operations can occur. Therefore, the number of operations in this test are suited perfectly to be used as a basis for comparison with the other cache layers.

Server	**Accesses**	**No Seek**	**Fast**	**Slow**	**Amount of Data**
Server 1	615	14	539	62	52 494 376 Bytes
Server 2	612	11	540	61	52 494 380 Bytes
Server 3	611	23	527	61	52 494 376 Bytes
Server 4	612	29	522	61	49 217 576 Bytes
Server 5	658	63	535	60	55 415 736 Bytes
Time					8.04 s

Table 6.1: `partdiff-par` — No Cache

6.8.2 Simple Write-Behind Cache

Table 6.2 shows the results obtained when using the simple write-behind cache layer. This cache layer already decreases the application’s runtime significantly. However, the total number of I/O operations basically stays the same, because this cache layer does not perform explicit merging of operations.

Server	**Accesses**	**No Seek**	**Fast**	**Slow**	**Amount of Data**
Server 1	599	1	536	62	52 494 376 Bytes
Server 2	592	0	544	61	52 494 380 Bytes
Server 3	588	0	524	61	52 494 376 Bytes
Server 4	601	0	540	61	49 217 576 Bytes
Server 5	622	0	564	60	55 415 736 Bytes
Time					4.92 s

Table 6.2: `partdiff-par` — Simple Write-Behind Cache

6.8.3 Aggregation Cache

Table 6.3 (page 85) shows the results obtained when using the aggregation cache layer. This cache layers decreases the application’s runtime again, but not as drastically as the simple write-behind cache layer. However, the number of total I/O operations decreases significantly. This is due to the fact that this cache layer performs explicit merging of operations.

Server	Accesses	No Seek	Fast	Slow	Amount of Data
Server 1	146	21	63	62	52 494 376 Bytes
Server 2	135	38	36	61	52 494 380 Bytes
Server 3	121	50	10	61	52 494 376 Bytes
Server 4	123	50	11	61	49 217 576 Bytes
Server 5	129	41	28	60	55 415 736 Bytes
Time					3.00 s

Table 6.3: `partdiff-par` — Aggregation Cache

6.8.4 Server-Directed I/O

The Server-Directed I/O cache layer is the only cache layer with a differing total amount of data access per server. This is due to the fact that this cache layer does not keep duplicate data in its internal queue for write operations. Whenever a newly arrived operation overwrites old data, this old data is purged from the queue. The `partdiff-par` application overwrites part of its visualization file in each iteration. This is the reason for the decreased amount of data.

Without Delayed Writes

Table 6.4 shows the results obtained when using the Server-Directed I/O cache layer without delayed writes. This cache layer does not decrease the application's runtime. The reason for this is the fact that application runtime is dominated by its computation in this case. This is analyzed in detail in section 7.1 (page 88). However, the total number of operations is decreased slightly.

Server	Accesses	No Seek	Fast	Slow	Amount of Data
Server 1	142	51	29	62	52 494 336 Bytes
Server 2	118	50	7	61	52 494 336 Bytes
Server 3	111	50	0	61	52 494 336 Bytes
Server 4	112	50	1	61	49 217 536 Bytes
Server 5	121	50	11	60	55 415 700 Bytes
Time					3.00 s

Table 6.4: `partdiff-par` — Server-Directed I/O (Without Delayed Writes)

With Delayed Writes

Table 6.5 (page 86) shows the results obtained when using the Server-Directed I/O cache layer with delayed writes. As can be seen, delayed writes significantly decrease the number of total operations. This is due to the fact that the write operations are held in memory for a longer time period, which makes it possible to merge more of these operations. Even though delayed writes do not decrease the execution time of the `partdiff-par` application the reduction of write operations can have advantageous effects in larger systems. For example, NCQ provides an operation queue which is used to speed up disk accesses[3]. However, this queue is limited to 32 entries. Therefore, reducing the number of operations can increase performance by allowing the underlying storage device to perform additional optimizations.

Server	**Accesses**	**No Seek**	**Fast**	**Slow**	**Amount of Data**
Server 1	61	7	2	52	52 494 336 Bytes
Server 2	63	6	4	53	52 494 336 Bytes
Server 3	59	6	0	53	52 494 336 Bytes
Server 4	60	7	1	52	49 217 536 Bytes
Server 5	64	7	4	53	55 415 700 Bytes
Time					3.00 s

Table 6.5: `partdiff-par` — Server-Directed I/O (With Delayed Writes)

6.8.5 Cache Layer Comparison

Figure 6.12 (page 87) gives an overview of the results obtained with the `partdiff-par` application when simulated with the various cache layers. As can be seen, the `SimpleWriteBehindCache` improves performance by a factor of 1.6. Both the `AggregationCache` and `ServerDirectedIO` cache layers improve performance by a factor of 2.7. No further improvements are possible, because the `partdiff-par` application's runtime is already dominated by computation when using these cache layers. Due to the additional computation and communication overhead the overall throughput is lower than in the previous tests. For a visualization of these results see section 7.2 (page 92).

[3]For more information see `http://en.wikipedia.org/wiki/Native_Command_Queuing`.

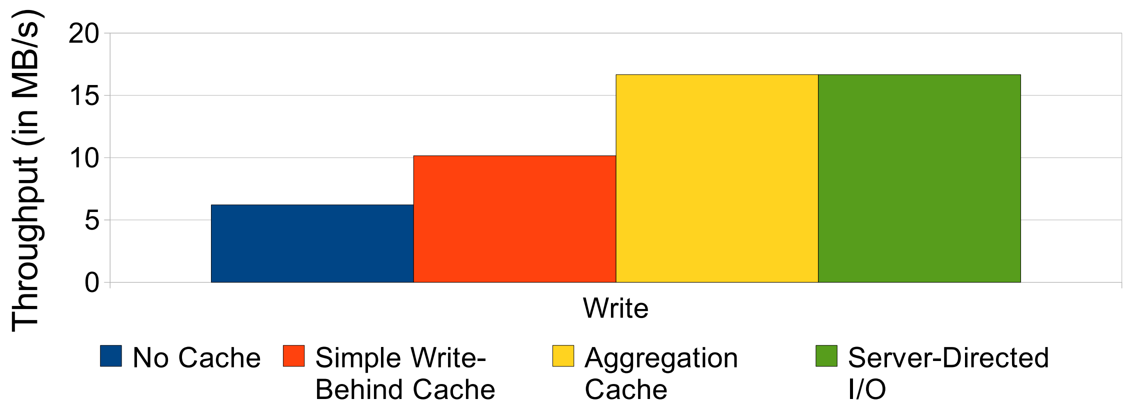


Figure 6.12: `partdiff-par` — Cache Layer Comparison

Summary

Random I/O operations are hard to optimize, because it is rarely possible to merge those operations. The results obtained in this chapter show that the server-side optimizations deliver better performance than the client-side optimizations in all tested cases. The Interleaved Two-Phase protocol outperforms the original Two-Phase protocol, if even by a relatively small amount. Obtaining satisfactory performance is possible by either batching smaller operations or using larger operations. Delayed writes can be used to reduce the number of I/O operations the I/O subsystem has to perform.

Chapter 7

Visualization

This chapter presents the possibility to visualize application traces with HDSunshot. Traces generated by the `partdiff-par` benchmark and collective I/O test from chapter 6 (page 61) are used as examples. Traces of a real `partdiff-par` application run and its simulated counterpart are compared. The simulated application run uses different cache layers to show their effects on the underlying I/O subsystem. Additionally, the traces of the collective I/O test are shown to demonstrate the Interleaved Two-Phase protocol.

7.1 Parallel Partial Differential Equation Solver

The `partdiff-par` application from section 6.8 (page 83) is used to demonstrate PIOsimHD's ability to compare traces of real-life applications with those of simulated applications. Additionally, the simulated `partdiff-par` application is run with varying cache layers to visualize their effects on the application's behavior and runtime.

7.1.1 Real Application Run

Figure 7.1 (page 89) shows the traces of the real `partdiff-par` application visualized with HDSunshot. The figure contains a screenshot of just the timeline window, all other parts of HDSunshot are excluded to save space. The five timelines represent the five clients executing the `partdiff-par` application. The PVFS servers are not traced and are therefore not available in this figure. Each timeline contains a multitude of blocks that represent the different operations the clients perform internally. The colors on the client timelines have the following meanings:

- **Yellow:** The clients communicate with each other.
- **Orange:** The clients perform an I/O operation.

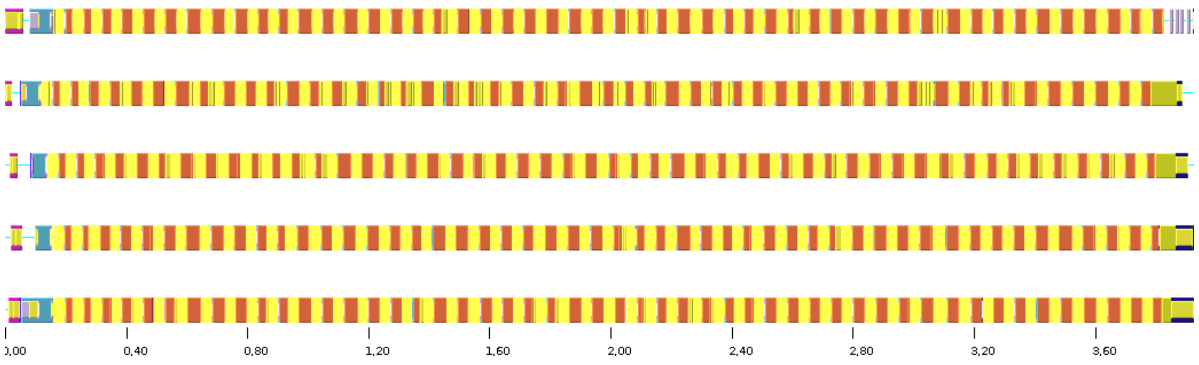


Figure 7.1: Real Application Run

7.1.2 Simulated Application Run

The following figures show the visualized traces of a simulated run of the `partdiff-par` application. The figures themselves are structured as follows: The first five timelines show the clients' activities, while the last five timelines show the servers' activities. The colors on the client timelines have the following meanings:

- **Turquoise:** The clients communicate with each other.
- **Blue:** The clients perform an I/O operation.

The colors on the server timelines have the following meanings:

- **Green:** The I/O subsystem performed an I/O operation without seeking.
- **Yellow:** The I/O subsystem performed an I/O operation with a fast seek.
- **Red:** The I/O subsystem performed an I/O operation with a slow seek.

More information about the different operation types can be found in subsection 6.1.1 (page 61).

No Cache

Figure 7.2 (page 90) shows the visualized traces of the simulated `partdiff-par` application as simulated with the `NoCache` cache layer. Since no caching is used the clients have to wait for each write operation to finish before continuing. Therefore, the client timelines contain many long write operations. The server timelines contain many write operations with relatively large gaps between them. This is due to the fact that there are no cached operations that could be executed while the clients are computing.

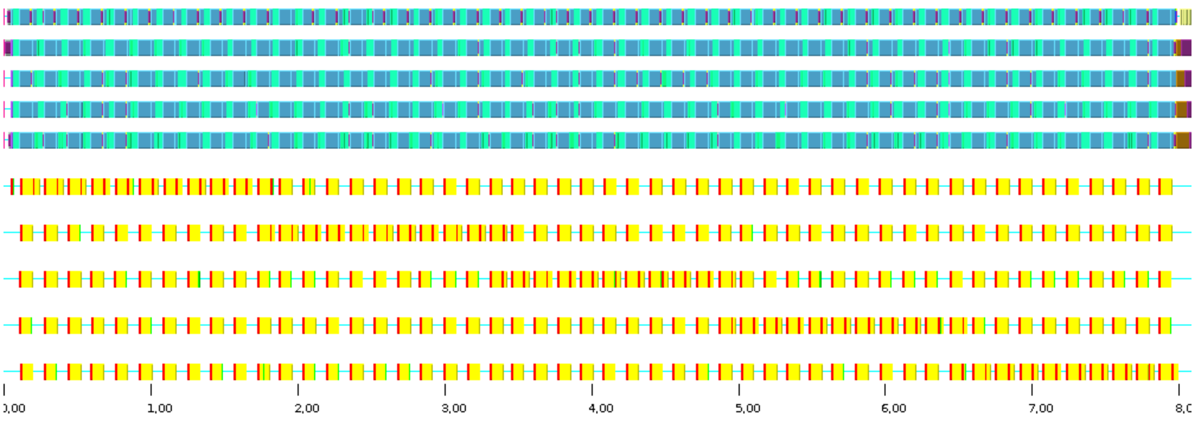


Figure 7.2: Simulated Application Run — No Cache

Simple Write-Behind Cache

Figure 7.3 shows the visualized traces of the simulated `partdiff-par` application as simulated with the `SimpleWriteBehindCache` cache layer. The clients do not have to wait for the actual write operations to finish before continuing and can compute without interruption. This also means that the servers can perform the write operations without waiting for new operations from the clients. As can be seen, the clients finish long before the servers. This means that the servers can not execute the write operations fast enough and therefore slow down the whole application.

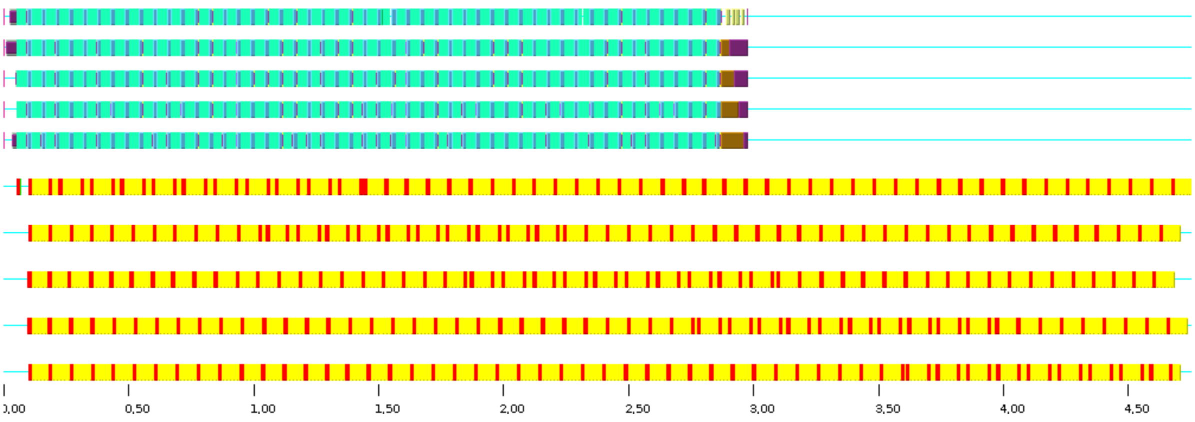


Figure 7.3: Simulated Application Run — Simple Write-Behind Cache

In this case the application user would not realize this inefficiency, because it is hidden by the cache layer performing write-behind. Once the cache is full the application's performance degrades, because old operations need to be finished before it is possible to accept new ones.

Aggregation Cache

Figure 7.4 (page 91) shows the visualized traces of the simulated `partdiff-par` application as simulated with the `AggregationCache` cache layer. As can be seen, the servers do not cause a longer runtime as opposed to the `SimpleWriteBehindCache` cache layer. Additionally, there are gaps between the write operations on the server timelines. This suggests that the clients' computation takes longer than the servers' I/O operations, that is, the application's total runtime can not be improved further by speeding up the I/O operations.

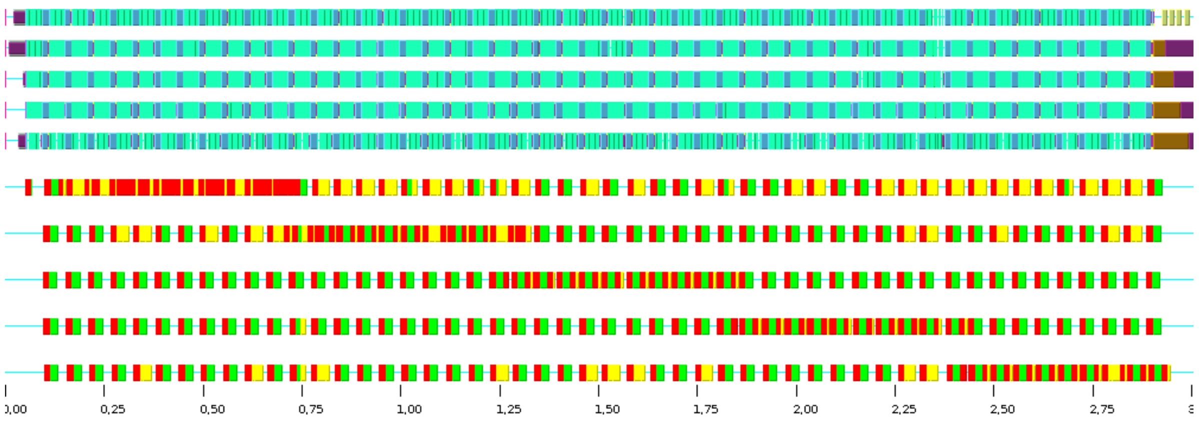


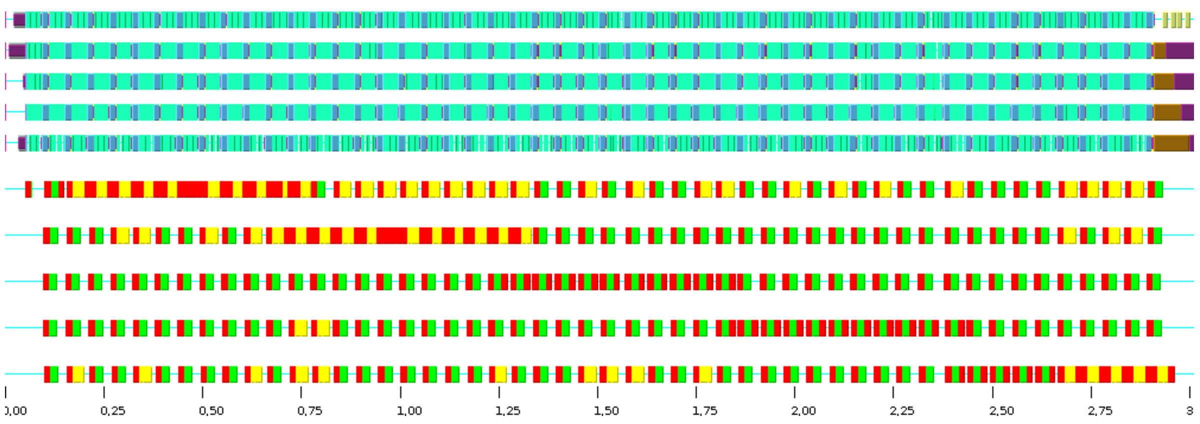
Figure 7.4: Simulated Application Run — Aggregation Cache

Server-Directed I/O

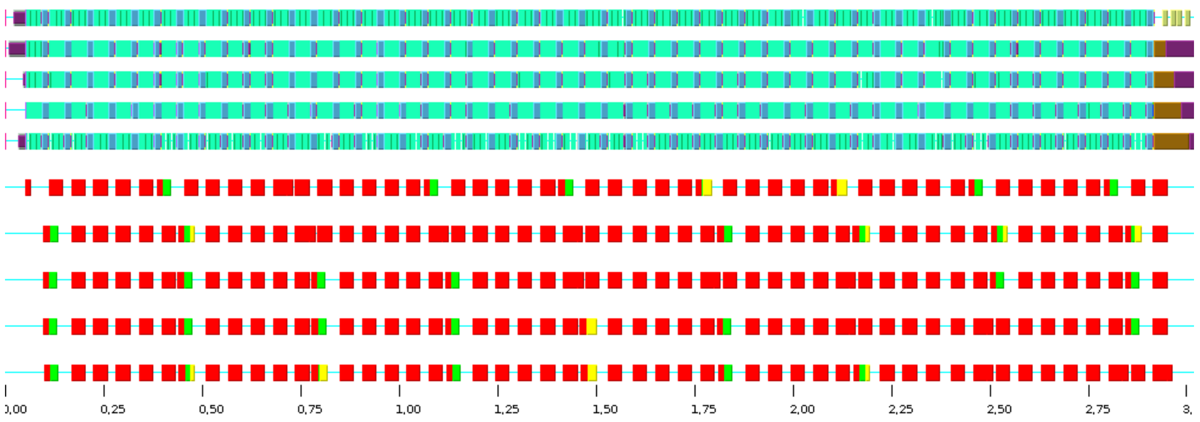
Figures 7.5a (page 92) and 7.5b (page 92) show the visualized traces of the simulated `partdiff-par` application as simulated with the `ServerDirectedIO` cache layer.

The `ServerDirectedIO` cache layer in figure 7.5a (page 92) is configured to not use delayed writes. Overall, the figure basically looks like figure 7.4. No further improvements are possible without the use of more advanced techniques.

The `ServerDirectedIO` cache layer in figure 7.5a (page 92) is configured to use delayed writes. Using delayed writes leads to less and larger write operations. This can be seen in the form of larger continuous write operations and gaps in the server timelines. However, this does not reduce the application's runtime, because it is already at its minimum, dictated by the clients' computation time.



(a) Without Delayed Writes



(b) With Delayed Writes

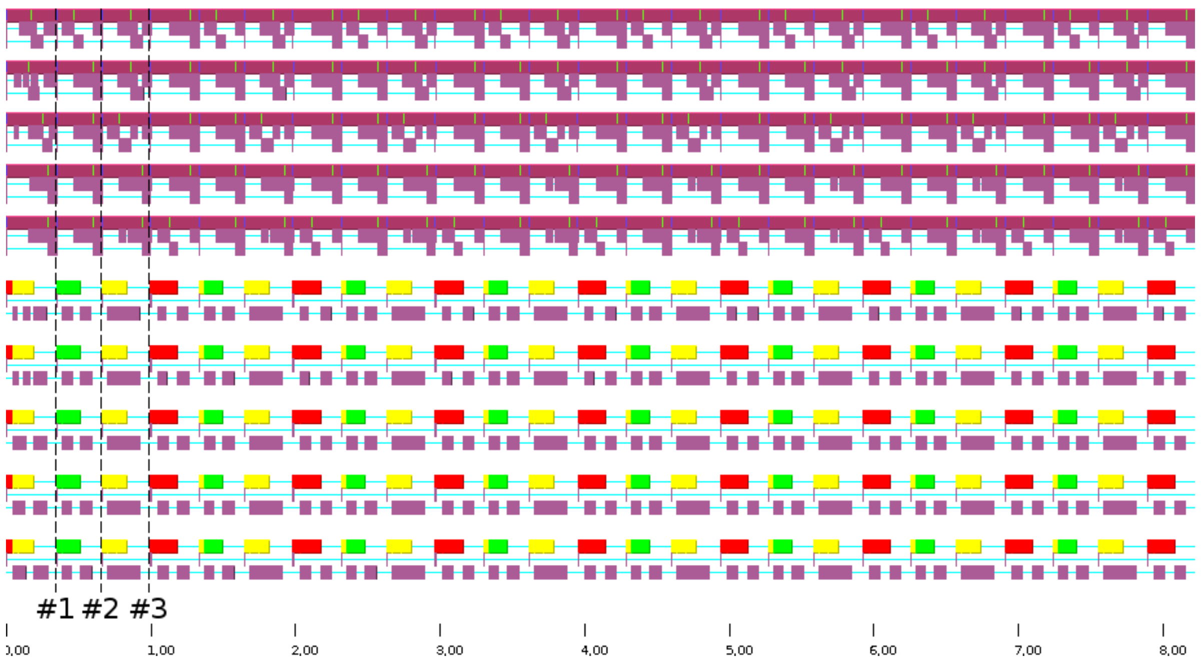
Figure 7.5: Simulated Application Run — Server-Directed I/O

7.2 Interleaved Two-Phase

The following figures show the visualized traces of the Interleaved Two-Phase protocol. The used application is equal to the one used in subsection 6.5.2 (page 75), except for the fact that only five clients and server are used. It uses the `ServerDirectedIO` cache layer, a data block size of 512 KiB and a file size of 1 000 MiB.

The figures are structured as follows: Each of the ten timelines is split into three subtimelines. The first one shows the operations on the respective node. For the clients this only shows the collective I/O operations, because they do not perform any other operations. The server subtimeline shows the actual I/O operations performed by the I/O subsystem. The colors on the server subtimeline mean the same as in subsection 7.1.2 (page 89). The second subtimeline shows received data and the third one shows sent data. The server timelines are actually split into four subtimelines, but the first one is always empty and ignored here. The first five timelines show the clients' activities, while the last five timelines show the servers' activities. Additionally, the first three iterations are pointed out by dashed lines with the appropriate iteration number. For example, #1 signifies the end of the first iteration, and so on.

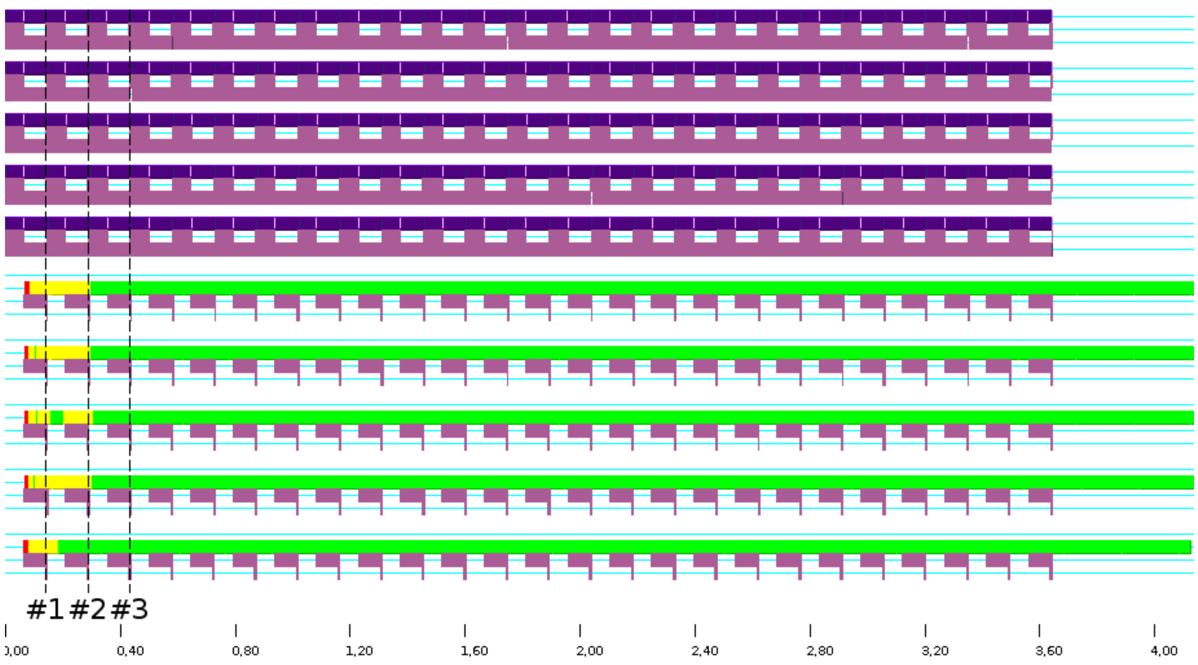
Figure 7.6a shows the Interleaved Two-Phase protocol used for read operations. The 25 iterations can be seen clearly by looking at the server timelines. Since the Two-Phase buffer is set to 8 MiB, each server reads $25 \cdot 8 = 200$ MiB, resulting in the 1 000 MiB being read completely by the five servers. Overall, there are a lot of gaps where both the clients and servers are idle, because the clients have to wait for the servers to read and send back the requested data and the servers can not do any work while waiting for the clients. Due to this the whole application takes about 8 s to complete.



(a) Read

Figure 7.6: Interleaved Two-Phase

Figure 7.6b (page 94) shows the Interleaved Two-Phase protocol used for write operations. This time, the 25 iterations are not as clearly visible when looking at the servers' I/O activity, because the servers continually write out data. Looking at the servers' receive subtimelines is useful here. As can be seen, the clients finish before the servers, because the clients finish as soon as the last iteration is sent to the servers. Overall, there are almost no gaps, because the clients do not have to wait for the servers and the servers can simply write out the data in the background. Due to this the application runtime is almost halved.



(b) Write

Figure 7.6: Interleaved Two-Phase Visualization

Summary

HDSunshot can be used to visualize traces produced by PIOsimHD and to compare traces of real-life applications with their simulated counterparts. It is also possible to evaluate different optimization techniques by analyzing the resulting traces with HDSunshot. Additionally, PIOsimHD allows users to obtain traces of the data flow within the network, as opposed to existing tracing mechanisms.

Chapter 8

Summary, Conclusion and Future Work

This chapter summarizes and concludes the thesis. Additionally, work that may prove useful to be done in the future is presented. This mainly includes missing functionality in PIOsimHD and the optimizations implemented in this thesis.

8.1 Summary and Conclusion

The main focus of this thesis is to compare existing client-side optimizations and the newly implemented server-side optimizations like Server-Directed I/O. Additionally, a modification of the Two-Phase protocol called Interleaved Two-Phase is proposed.

The results from section 6.6 (page 77) show that client-side optimizations like the Two-Phase protocol do not necessarily beat server-side optimizations in terms of performance. In fact, the server-side optimizations found in the `AggregationCache` and `ServerDirectedIO` cache layers deliver better performance in all comparable tests.

The results also suggest that even simple server-side optimizations like those in the `AggregationCache` cache layer are good enough for many use cases.

Integrating such optimizations into parallel cluster file systems could alleviate the need for sophisticated client-side optimizations. Additionally, due to their additional knowledge of internal workflows server-side optimizations may be better suited to provide high performance in general.

Modifications of the Two-Phase protocol also promise better performance. An example of a straightforward modification of the original Two-Phase protocol is the so-called Interleaved Two-Phase protocol. The Interleaved Two-Phase protocol beats its unmodified counterpart in all tests, but still performs worse than the server-side optimizations.

Visualizing the application traces is useful to comprehend the inner workings of distributed systems and the influence of I/O optimizations.

8.2 Future Work

Even though the implementation of Server-Directed I/O is promising a lot of work remains to be done, both for the Server-Directed I/O cache layer and PIOsimHD itself. The following is a list of features that would be interesting to implement.

At the moment the Server-Directed I/O cache layer does not influence the order in which the clients send their data. Clients first announce their write operations and then send their data when this announcement is acknowledged. This could be used to further increase performance, because data could be received in the order in which it is needed, avoiding needless buffering. For completeness this should be implemented.

Server-Directed I/O should make better use of the read optimization's data sieving support. For example, the maximum hole size could be dynamically determined by either using the respective sizes of the involved read requests, or by estimating the number of potential merge operations that can be performed with a given hole size. A static hole size often delivers greatly varying performance, depending on the use case. For this reason this particular optimization is disabled for now.

The cache layers do not cope well with memory exhaustion at the moment. For example, too many write operations could fill up the whole main memory, causing read operations to be deferred. This can also be potentially used to starve read operations by sending many large write operations. This situation gets worse when the I/O subsystem is slower than the network and large files are processed.

The Aggregation and Server-Directed I/O cache layers are currently too slow to be used with massive amounts of operations. For example, using a data block size of 512 Bytes and reading or writing a 1 000 MiB file takes too long. This is mostly due to their inefficient handling of their internal operation queues. However, several starting points exist to improve their performance. This should be done to allow simulation of larger long-running applications with huge numbers of simulation events.

PIOsimHD-Simulator should support priorities for read and write operations. The cache layers could use this information to make sure that no operation waits in the queue for too long. For example, this could be solved by raising the priority based on the time an operation has been waiting. This should be a relatively easy to achieve by extending the `IOJob` interface.

The implementation of the Two-Phase protocol could be optimized to use double buffering. Currently the Two-Phase buffer is first filled with data and then this data is distributed to the other clients or written to disk. This could be optimized by using two Two-Phase buffers in an alternating fashion, that is, the second buffer could already be filled while the data of the first buffer is still being sent.

Depending on the workload and hardware characteristics this could potentially double the performance.

Additionally, it may prove useful to improve the logic used to decide whether to make use of the Two-Phase protocol when performing collective I/O. For example, this new logic could also use information about the underlying I/O subsystem. This could be integrated into the new Interleaved Two-Phase protocol.

A component simulating the RAM of the nodes is missing in PIOsimHD. For example, the latency caused by RAM accesses is currently not simulated at all. Additionally, this would make it possible to implement a real read cache in the cache layers themselves.

PIOsimHD currently only includes a component for simulating a simple abstraction of SSDs. However, current SSDs are very complex. For example, read and write operations use different block sizes and write operations need to consider the erase block size, because this can have a significant impact on performance.

Additional benchmarks using SSDs as the underlying I/O subsystem would be interesting to perform. It would be especially interesting to see a comparison of how the different optimizations behave when using HDDs and SSDs respectively.

Bibliography

[BDK97] Rajive Bagrodia, Stephen Docy, and Andy Kahn. Parallel simulation of parallel file systems and I/O programs. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–17, New York, NY, USA, 1997. ACM.

[BMLX03] Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Lan Xue. Efficient Metadata Management in Large Distributed Storage Systems. In *MSS '03: Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, page 290, Washington, DC, USA, 2003. IEEE Computer Society.

[DW07] Ananth Devulapalli and Pete Wyckoff. File Creation Strategies in a Distributed Metadata File System. In *Proceedings of IPDPS '07*, Long Beach, CA, March 2007.

[KKL08] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. Directory-Based Metadata Optimizations for Small Files in PVFS. In *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 90–99, Berlin, Heidelberg, 2008. Springer-Verlag.

[KKL09] Michael Kuhn, Julian Martin Kunkel, and Thomas Ludwig. Dynamic file system semantics to enable metadata optimizations in PVFS. *Concurrency and Computation: Practice and Experience*, 2009.

[Kot97] David Kotz. Disk-directed I/O for MIMD multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, 1997.

[KRVP07] Dries Kimpe, Rob Ross, Stefan Vandewalle, and Stefaan Poedts. Transparent Log-Based Data Storage in MPI-IO Applications. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 233–241. Springer Berlin / Heidelberg, 2007.

[Kuh07] Michael Kuhn. Directory-Based Metadata Optimizations for Small Files in PVFS. Bachelor's Thesis, Ruprecht-Karls-Universität Heidelberg, September 2007.

[Kun] Julian Kunkel. Simulation of Parallel I/O and MPI Communication with a Discrete Event Simulator. Technical Report (Unpublished).

[Kun07] Julian Martin Kunkel. Towards Automatic Load Balancing of a Parallel File System with Subfile Based Migration. Master's Thesis, Ruprecht-Karls-Universität Heidelberg, July 2007.

[LCC$^+$07] Wei-keng Liao, Avery Ching, Kenin Coloma, Arifa Nisar, Alok Choudhary, Jacqueline Chen, Ramanan Sankaran, and Scott Klasky. Using MPI file caching to improve parallel write performance for large-scale scientific applications. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.

[LCCW05] Wei-keng Liao, Kenin Coloma, Alok N. Choudhary, and Lee Ward. Cooperative Write-Behind Data Buffering for MPI I/O. In Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra, editors, *PVM/MPI*, volume 3666 of *Lecture Notes in Computer Science*, pages 102–109. Springer, 2005.

[PTH$^+$01] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges. MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 17–17, New York, NY, USA, 2001. ACM.

[SIC$^+$07] David E. Singh, Florin Isaila, Alejandro Calderon, Felix Garcia, and Jesus Carretero. Multiple-Phase Collective I/O Technique for Improving Data Access Locality. In *PDP '07: Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 534–542, Washington, DC, USA, 2007. IEEE Computer Society.

[SIPC09] David E. Singh, Florin Isaila, Juan C. Pichel, and Jesús Carretero. A collective I/O implementation based on inspector–executor paradigm. *The Journal of Supercomputing*, 47(1):53–75, 2009.

[TGL99] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *FRONTIERS '99: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, page 182, Washington, DC, USA, 1999. IEEE Computer Society.

[TLC$^+$99] Brian L. Tierney, Jason Lee, Brian Crowley, Mason Holding, Jeremy Hylton, and Fred L. Drake Jr. A Network-Aware Distributed Storage Cache for Data Intensive Environments. In *HPDC '99: Proceedings of the*

8th IEEE International Symposium on High Performance Distributed Computing, page 33, Washington, DC, USA, 1999. IEEE Computer Society.

[VNS05] Murali Vilayannur, Partho Nath, and Anand Sivasubramaniam. Providing tunable consistency for a parallel file store. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[WPBM04] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 4, Washington, DC, USA, 2004. IEEE Computer Society.

[Zuk05] Marcin Zukowski. Improving I/O Bandwidth for Data-Intensive Applications. In *Proceedings of the British National Conference on Databases (BNCOD)*, July 2005.

Appendices

Appendix A

Used Tools

- TeX Live[1]
  - TEX distribution used to compile the LATEX sources.
- BibTeX[2]
  - Reference management software used to organize the bibliography.
- Vim[3]
  - Advanced text editor used to write the LATEX sources.
- VIM-LATEX[4]
  - Vim addon providing LATEX support.
- OpenOffice.org[5] Draw
  - Drawing application used to produce the figures.
- unoconv[6]
  - Conversion application used to automatically convert OpenOffice.org drawings to PDF[7].
- Graphviz[8]
  - Graph visualization software used to produce state machine figures.
- Git[9]
  - Distributed version control system used to manage the LATEX sources.

[1]TeX Live is available at `http://www.tug.org/texlive/`.
[2]BibTeX is part of TeX Live.
[3]Vim is available at `http://www.vim.org/`.
[4]VIM-LATEX is available at `http://vim-latex.sourceforge.net/`.
[5]OpenOffice.org is available at `http://www.openoffice.org/`.
[6]unoconv is available at `http://dag.wieers.com/home-made/unoconv/`.
[7]Portable Document Format
[8]Graphviz is available at `http://www.graphviz.org/`.
[9]Git is available at `http://git-scm.com/`.

Appendix B

Usage Instructions

B.1 Source Code Repository

B.1.1 Setup

Cloning The `git svn clone` command clones the given SVN[1] repository and converts it into a Git[2] repository.

```
$ git svn clone \
  svn+ssh://user@pvs-cluster.informatik.uni-heidelberg.de\
  /home/sighpio/PIOsimHD
```

`.gitignore` The `.gitignore` file causes Git to ignore certain files.

Listing B.1 shows the `.gitignore` file for the PIOsimHD repository. It causes Git to ignore all trace files produced by PIOsimHD-Simulator.

Listing B.1: `PIOsimHD-Simulator/.gitignore`

```
run-trace*
```

B.1.2 Committing and Updating

Committing Locally The `git add -i` command is used to add changes to Git's index. The `git commit` command creates a commit containing these changes.

```
$ git add -i
$ git commit
```

[1]Subversion

[2]Git is available at `http://git-scm.com/`.

Committing Remotely The `git svn dcommit` command puts the local Git commit into the SVN repository. The `git stash` commands are needed, because `git svn dcommit` needs a clean working tree.

```
$ git stash
$ git svn dcommit
$ git stash pop
```

Updating The `git svn rebase` command updates the repository to the current SVN revision. The `git stash` commands are needed, because `git svn rebase` needs a clean working tree.

```
$ git stash
$ git svn rebase
$ git stash pop
```

B.2 PIOsimHD-Simulator

`partdiff-par`

Executing `CommandLineInterface.java` with the following arguments runs the `partdiff-par` benchmark from section 6.8 (page 83).

Listing B.2 shows `CommandLineInterface` arguments used to simulate a run of the `partdiff-par` application.

Listing B.2: `CommandLineInterface` Arguments

```
-i ../HDTraceFormat/Example/partdiff-mpi-io-synced/pvs-cluster-model
   ↪ .xml --load-program-on-demand
```

B.3 HDTrace

B.3.1 Setup

MPD

MPD[3] is a daemon that allows MPI applications to be distributed and executed on several machines — a so-called MPD ring.

[3]Multi-Purpose Daemon

Configuration The .mpd.conf file contains the MPD password.

```
$ touch .mpd.conf
$ chmod 600 .mpd.conf
$ echo 'MPD_SECRETWORD=42' > .mpd.conf
```

Participating Machines The mpd.hosts file contains the hostnames of all machines participating in the MPD ring.

```
$ echo 'node01' > mpd.hosts
$ echo 'node02' >> mpd.hosts
$ echo 'node03' >> mpd.hosts
```

HDTrace

The module command loads all configuration settings — like environment variables — to use the PIOsimHD framework for tracing.

```
$ module load hdtrace/1.0
```

B.3.2 Compiling and Executing

Compiling The following commands compile the b_eff_io benchmark application using the mpicc compiler.

```
$ tar xf b_eff_io_v2.1.tar.gz
$ cd b_eff_io
$ mpicc -lm -o b_eff_io b_eff_io.c
```

Executing The mpdboot command starts a MPD ring. The mpirun command starts the b_eff_io application on the MPD ring.

```
$ mpdboot -n 3 -f mpd.hosts
$ mpirun -n 3 ./b_eff_io
```

Cleanup The mpdallexit command shuts down the MPD ring.

```
$ mpdallexit
```

Appendix C

Evaluation Results

It is important to note that these results represent the aggregated throughput. The figures in chapter 6 (page 61) use per-client throughput. Therefore, the results below must be divided by 10 when comparing them with the figures.

C.1 Random I/O Test

```
NoCache
  5120 READ  1048576000 B, 480.43993272859996 s (0.40954037050021325 s)
  5120 READ  2.0814256515286353 MB/s
  5120 WRITE 1048576000 B, 379.74895692810003 s (0.5796268021213603 s)
  5120 WRITE 2.6333186220952163 MB/s
  51200 READ  1048576000 B, 70.3209508305 s (0.2461663978933605 s)
  51200 READ  14.22051306459688 MB/s
  51200 WRITE 1048576000 B, 62.064023903499994 s (0.33664790269020894 s)
  51200 WRITE 16.112393897547573 MB/s
  524288 READ  1048576000 B, 23.3586019284 s (0.05966495834943108 s)
  524288 READ  42.81078135862977 MB/s
  524288 WRITE 1048576000 B, 23.391223519500002 s (0.04309578760319361 s)
  524288 WRITE 42.75107709378066 MB/s
SimpleWriteBehindCache
  5120 READ  1048576000 B, 480.43993272859996 s (0.40954037050021325 s)
  5120 READ  2.0814256515286353 MB/s
  5120 WRITE 1048576000 B, 300.2951743493 s (0.26085124337011006 s)
  5120 WRITE 3.330056842128309 MB/s
  51200 READ  1048576000 B, 70.3209508305 s (0.2461663978933605 s)
  51200 READ  14.22051306459688 MB/s
  51200 WRITE 1048576000 B, 53.18417199830001 s (0.23894914448050142 s)
  51200 WRITE 18.802586604750832 MB/s
  524288 READ  1048576000 B, 23.3586019284 s (0.05966495834943108 s)
  524288 READ  42.81078135862977 MB/s
  524288 WRITE 1048576000 B, 23.372220795599997 s (0.09325321500748063 s)
  524288 WRITE 42.785835746864834 MB/s
AggregationCache
  5120 READ  1048576000 B, 480.2652491281 s (0.5114650114861422 s)
  5120 READ  2.0821827142718634 MB/s
  5120 WRITE 1048576000 B, 35.0486731206 s (0.0809730654070207 s)
  5120 WRITE 28.531750590359607 MB/s
  51200 READ  1048576000 B, 70.14761168519999 s (0.2261228904126729 s)
  51200 READ  14.2556528437159 MB/s
  51200 WRITE 1048576000 B, 7.8072400707 s (0.04723829342018465 s)
  51200 WRITE 128.08623674234468 MB/s
  524288 READ  1048576000 B, 23.2630255663 s (0.09417641088986031 s)
  524288 READ  42.98666986157857 MB/s
  524288 WRITE 1048576000 B, 4.0203020197 s (0.03879319908671092 s)
  524288 WRITE 248.73753143417352 MB/s
ServerDirectedIO
  5120 READ  1048576000 B, 470.54163953590006 s (1.0166849689938773 s)
  5120 READ  2.125210429806616 MB/s
  5120 WRITE 1048576000 B, 28.2049480228 s (0.15768397860774702 s)
  5120 WRITE 35.45477194964625 MB/s
  51200 READ  1048576000 B, 69.24346647129998 s (0.1613120203687557 s)
  51200 READ  14.441795752881317 MB/s
  51200 WRITE 1048576000 B, 6.839998547799999 s (0.05849529045609294 s)
  51200 WRITE 146.19886144882847 MB/s
  524288 READ  1048576000 B, 23.068002521300002 s (0.17455843410094402 s)
  524288 READ  43.35009063210579 MB/s
  524288 WRITE 1048576000 B, 3.5731329932 s (0.02568778555751315 s)
  524288 WRITE 279.86643707443625 MB/s
```

C.2 Individual I/O Test

C.2.1 Contiguous Access — 1 Operation

```
NoCache
  5120 READ  1048576000 B, 146.027949339 s
  5120 READ  6.848004128843353 MB/s
  5120 WRITE 1048576000 B, 119.094858293 s
  5120 WRITE 8.39666812096771 MB/s
  51200 READ  1048576000 B, 21.773296 s
  51200 READ  45.92781910465003 MB/s
  51200 WRITE 1048576000 B, 21.45670225 s
  51200 WRITE 46.60548430735669 MB/s
  524288 READ  1048576000 B, 10.278591798 s
  524288 READ  97.28959177020525 MB/s
  524288 WRITE 1048576000 B, 10.280323109 s
  524288 WRITE 97.27320721316056 MB/s
SimpleWriteBehindCache
  5120 READ  1048576000 B, 146.027949339 s
  5120 READ  6.848004128843353 MB/s
  5120 WRITE 1048576000 B, 92.950948768 s
  5120 WRITE 10.758362483162383 MB/s
  51200 READ  1048576000 B, 21.773296 s
  51200 READ  45.92781910465003 MB/s
  51200 WRITE 1048576000 B, 14.580759687 s
  51200 WRITE 68.5835320975481 MB/s
  524288 READ  1048576000 B, 10.278591798 s
  524288 READ  97.28959177020525 MB/s
  524288 WRITE 1048576000 B, 9.931221594 s
  524288 WRITE 100.69254728986768 MB/s
AggregationCache
  5120 READ  1048576000 B, 154.06681688 s
  5120 READ  6.490690339756178 MB/s
  5120 WRITE 1048576000 B, 18.541083015 s
  5120 WRITE 53.93428200450781 MB/s
  51200 READ  1048576000 B, 20.305640744 s
  51200 READ  49.24739941020991 MB/s
  51200 WRITE 1048576000 B, 3.505185092 s
  51200 WRITE 285.2916390299425З MB/s
  524288 READ  1048576000 B, 4.885013716 s
  524288 READ  204.7077159117643 MB/s
  524288 WRITE 1048576000 B, 2.027372673 s
  524288 WRITE 493.24922512655377 MB/s
ServerDirectedIO
  5120 READ  1048576000 B, 146.10821002 s
  5120 READ  6.844242358886713 MB/s
  5120 WRITE 1048576000 B, 18.546249682 s
  5120 WRITE 53.9192568387854 MB/s
  51200 READ  1048576000 B, 19.433986418 s
  51200 READ  51.45624672629119 MB/s
  51200 WRITE 1048576000 B, 3.508523379 s
  51200 WRITE 285.02018997092165 MB/s
  524288 READ  1048576000 B, 2.043356537 s
  524288 READ  489.3908536726402 MB/s
  524288 WRITE 1048576000 B, 2.027372673 s
  524288 WRITE 493.24922512655377 MB/s
```

C.2.2 Non-Contiguous Access — 100 Operations

```
NoCache
  5120 READ  1048576000 B, 117.746240842 s
  5120 READ  8.492840135269105 MB/s
  5120 WRITE 1048576000 B, 117.887454933 s
  5120 WRITE 8.482666799180105 MB/s
  51200 READ  1048576000 B, 21.85244504 s
  51200 READ  45.761469628205965 MB/s
  51200 WRITE 1048576000 B, 26.261753244 s
  51200 WRITE 38.078188866863606 MB/s
  524288 READ  1048576000 B, 10.414065977 s
  524288 READ  96.02397394145105 MB/s
  524288 WRITE 1048576000 B, 15.241344531 s
  524288 WRITE 65.61100944644737 MB/s
SimpleWriteBehindCache
  5120 READ  1048576000 B, 117.746240842 s
  5120 READ  8.492840135269105 MB/s
  5120 WRITE 1048576000 B, 117.861254075 s
  5120 WRITE 8.484552517688794 MB/s
  51200 READ  1048576000 B, 21.85244504 s
  51200 READ  45.761469628205965 MB/s
  51200 WRITE 1048576000 B, 27.254636535 s
  51200 WRITE 36.69100480264394 MB/s
  524288 READ  1048576000 B, 10.414065977 s
  524288 READ  96.02397394145105 MB/s
  524288 WRITE 1048576000 B, 15.241344531 s
  524288 WRITE 65.61100944644737 MB/s
AggregationCache
  5120 READ  1048576000 B, 9.751169836 s
  5120 READ  102.5517980733076 MB/s
  5120 WRITE 1048576000 B, 2.030927439 s
  5120 WRITE 492.3858828222764 MB/s
  51200 READ  1048576000 B, 3.120902 s
  51200 READ  320.4201862153954 MB/s
  51200 WRITE 1048576000 B, 2.082627976 s
  51200 WRITE 480.16256937095903 MB/s
  524288 READ  1048576000 B, 2.231920598 s
  524288 READ  448.04461274119217 MB/s
  524288 WRITE 1048576000 B, 2.043116687 s
  524288 WRITE 489.448305308663 MB/s
ServerDirectedIO
  5120 READ  1048576000 B, 6.476519658 s
  5120 READ  154.4039164252005 MB/s
  5120 WRITE 1048576000 B, 2.030927439 s
  5120 WRITE 492.3858828222764 MB/s
  51200 READ  1048576000 B, 3.402955863 s
  51200 READ  293.8621716704881 MB/s
  51200 WRITE 1048576000 B, 2.082627976 s
  51200 WRITE 480.16256937095903 MB/s
  524288 READ  1048576000 B, 2.255097161 s
  524288 READ  443.4398735868924 MB/s
  524288 WRITE 1048576000 B, 2.030179569 s
  524288 WRITE 492.5672661027553 MB/s
```

C.2.3 Non-Contiguous Access — All Operations

```
NoCache
  5120 READ  1048576000 B, 117.828092033 s
  5120 READ  8.48694044642538 MB/s
  5120 WRITE 1048576000 B, 122.735387396 s
  5120 WRITE 8.147609432099209 MB/s
  51200 READ  1048576000 B, 21.934033634 s
  51200 READ  45.59124950232125 MB/s
  51200 WRITE 1048576000 B, 26.504183134 s
  51200 WRITE 37.72989323776531 MB/s
  524288 READ  1048576000 B, 10.351081997 s
  524288 READ  96.60825798595981 MB/s
  524288 WRITE 1048576000 B, 15.240600495 s
  524288 WRITE 65.6142125323783 MB/s
SimpleWriteBehindCache
  5120 READ  1048576000 B, 117.828092033 s
  5120 READ  8.48694044642538 MB/s
  5120 WRITE 1048576000 B, 122.908887398 s
  5120 WRITE 8.136108146206132 MB/s
  51200 READ  1048576000 B, 21.934033634 s
  51200 READ  45.59124950232125 MB/s
  51200 WRITE 1048576000 B, 26.504183135 s
  51200 WRITE 37.729893236341766 MB/s
  524288 READ  1048576000 B, 10.351081997 s
  524288 READ  96.60825798595981 MB/s
  524288 WRITE 1048576000 B, 15.240600495 s
  524288 WRITE 65.6142125323783 MB/s
AggregationCache
  5120 READ  1048576000 B, 2.125372651 s
  5120 READ  470.505724974627 MB/s
  5120 WRITE 1048576000 B, 2.036433315 s
  5120 WRITE 491.05462606321584 MB/s
  51200 READ  1048576000 B, 2.123721621 s
  51200 READ  470.87150693937394 MB/s
  51200 WRITE 1048576000 B, 2.031638057 s
  51200 WRITE 492.2136581142022 MB/s
  524288 READ  1048576000 B, 2.134629005 s
  524288 READ  468.4654793210777 MB/s
  524288 WRITE 1048576000 B, 2.043177719 s
  524288 WRITE 489.4336849412364 MB/s
ServerDirectedIO
  5120 READ  1048576000 B, 2.1351463649999998 s
  5120 READ  468.3519670558979 MB/s
  5120 WRITE 1048576000 B, 2.036433315 s
  5120 WRITE 491.05462606321584 MB/s
  51200 READ  1048576000 B, 2.132903426 s
  51200 READ  468.84448110028967 MB/s
  51200 WRITE 1048576000 B, 2.031638059 s
  51200 WRITE 492.2136576296536 MB/s
  524288 READ  1048576000 B, 2.1398960750000002 s
  524288 READ  467.3124137582242 MB/s
  524288 WRITE 1048576000 B, 2.029435533 s
  524288 WRITE 492.7478521684088 MB/s
```

C.3 Collective I/O Test

C.3.1 Two-Phase

```
NoCache
  5120 READ  1048576000 B, 5.867568858 s
  5120 READ  170.4283365395147 MB/s
  5120 WRITE 1048576000 B, 135.350505283 s
  5120 WRITE 7.388225096826439 MB/s
  51200 READ  1048576000 B, 5.839443908 s
  51200 READ  171.24918327068895 MB/s
  51200 WRITE 1048576000 B, 38.019368463 s
  51200 WRITE 26.302383243771875 MB/s
  524288 READ  1048576000 B, 5.474242602 s
  524288 READ  182.67367245190277 MB/s
  524288 WRITE 1048576000 B, 25.231989084 s
  524288 WRITE 39.63223020868045 MB/s
SimpleWriteBehindCache
  5120 READ  1048576000 B, 5.867568858 s
  5120 READ  170.4283365395147 MB/s
  5120 WRITE 1048576000 B, 132.031460822 s
  5120 WRITE 7.573952403269728 MB/s
  51200 READ  1048576000 B, 5.839443908 s
  51200 READ  171.24918327068895 MB/s
  51200 WRITE 1048576000 B, 36.212464892 s
  51200 WRITE 27.61480067657362 MB/s
  524288 READ  1048576000 B, 5.474242602 s
  524288 READ  182.67367245190277 MB/s
  524288 WRITE 1048576000 B, 23.777266643 s
  524288 WRITE 42.05697883673264 MB/s
AggregationCache
  5120 READ  1048576000 B, 5.867568858 s
  5120 READ  170.4283365395147 MB/s
  5120 WRITE 1048576000 B, 2.775984705 s
  5120 WRITE 360.2325323330627 MB/s
  51200 READ  1048576000 B, 5.839443908 s
  51200 READ  171.24918327068895 MB/s
  51200 WRITE 1048576000 B, 2.769466451 s
  51200 WRITE 361.08038053283497 MB/s
  524288 READ  1048576000 B, 5.474242602 s
  524288 READ  182.67367245190277 MB/s
  524288 WRITE 1048576000 B, 2.8463754249999997 s
  524288 WRITE 351.32400006580303 MB/s
ServerDirectedIO
  5120 READ  1048576000 B, 5.88493782 s
  5120 READ  169.9253298142749 MB/s
  5120 WRITE 1048576000 B, 2.789814135 s
  5120 WRITE 358.44681817844474 MB/s
  51200 READ  1048576000 B, 5.851306495 s
  51200 READ  170.90200297224388 MB/s
  51200 WRITE 1048576000 B, 2.8092997889999998 s
  51200 WRITE 355.9605863053728 MB/s
  524288 READ  1048576000 B, 5.596070551 s
  524288 READ  178.69681786290258 MB/s
  524288 WRITE 1048576000 B, 2.918934791 s
  524288 WRITE 342.59072970157354 MB/s
```

C.3.2 Interleaved Two-Phase

```
NoCache
  5120 READ  1048576000 B, 4.782318441 s
  5120 READ  209.10359950662266 MB/s
  5120 WRITE 1048576000 B, 124.214700582 s
  5120 WRITE 8.050576906876273 MB/s
  51200 READ  1048576000 B, 4.842799575 s
  51200 READ  206.49213012289488 MB/s
  51200 WRITE 1048576000 B, 29.233955314 s
  51200 WRITE 34.206797857459435 MB/s
  524288 READ  1048576000 B, 4.566702312 s
  524288 READ  218.97639295915639 MB/s
  524288 WRITE 1048576000 B, 17.016398263 s
  524288 WRITE 58.76684269751568 MB/s
SimpleWriteBehindCache
  5120 READ  1048576000 B, 4.782318441 s
  5120 READ  209.10359950662266 MB/s
  5120 WRITE 1048576000 B, 123.464284581 s
  5120 WRITE 8.099508318488168 MB/s
  51200 READ  1048576000 B, 4.842799575 s
  51200 READ  206.49213012289488 MB/s
  51200 WRITE 1048576000 B, 27.969692393 s
  51200 WRITE 35.752985265231985 MB/s
  524288 READ  1048576000 B, 4.566702312 s
  524288 READ  218.97639295915639 MB/s
  524288 WRITE 1048576000 B, 15.838040735 s
  524288 WRITE 63.13912287080628 MB/s
AggregationCache
  5120 READ  1048576000 B, 4.665140413 s
  5120 READ  214.35582029071932 MB/s
  5120 WRITE 1048576000 B, 2.251897167 s
  5120 WRITE 444.0700111240914 MB/s
  51200 READ  1048576000 B, 4.180562118 s
  51200 READ  239.20228231853292 MB/s
  51200 WRITE 1048576000 B, 2.248735177 s
  51200 WRITE 444.6944265505213 MB/s
  524288 READ  1048576000 B, 4.331482415 s
  524288 READ  230.8678425051392 MB/s
  524288 WRITE 1048576000 B, 2.207208941 s
  524288 WRITE 453.0608686040113 MB/s
ServerDirectedIO
  5120 READ  1048576000 B, 4.148568066 s
  5120 READ  241.04702733350308 MB/s
  5120 WRITE 1048576000 B, 2.195084736 s
  5120 WRITE 455.56327899316227 MB/s
  51200 READ  1048576000 B, 4.21927462 s
  51200 READ  237.00756411062903 MB/s
  51200 WRITE 1048576000 B, 2.186729407 s
  51200 WRITE 457.30395210256575 MB/s
  524288 READ  1048576000 B, 4.497086945 s
  524288 READ  222.36616997406082 MB/s
  524288 WRITE 1048576000 B, 2.164988756 s
  524288 WRITE 461.8961633073719 MB/s
```

C.4 Database I/O Test

C.4.1 Contiguous Access — 1 Operation

```
NoCache
  512 READ  128000000 B, 15.248816146100001 s (0.20380218337377867 s)
  512 READ  8.005232100015869 MB/s
  512 WRITE 128000000 B, 12.7023798924 s (0.14625509091076325 s)
  512 WRITE 9.610034775690835 MB/s
  512 RW  128000000 B, 13.2885156322 s (0.15308890506887798 s)
  512 RW  9.186151100594406 MB/s
SimpleWriteBehindCache
  512 READ  128000000 B, 15.248816146100001 s (0.20380218337377867 s)
  512 READ  8.005232100015869 MB/s
  512 WRITE 128000000 B, 10.6350764841 s (0.26458582784259627 s)
  512 WRITE 11.478085059613962 MB/s
  512 RW  128000000 B, 10.5406452184 s (0.16086971773060044 s)
  512 RW  11.580914637645822 MB/s
AggregationCache
  512 READ  128000000 B, 15.248816146100001 s (0.20380218337377867 s)
  512 READ  8.005232100015869 MB/s
  512 WRITE 128000000 B, 10.615109818500002 s (0.26506907738117447 s)
  512 WRITE 11.49967495270336 MB/s
  512 RW  128000000 B, 10.5381344951 s (0.1632812950770421 s)
  512 RW  11.583673804577082 MB/s
ServerDirectedIO
  512 READ  128000000 B, 15.3814292401 s (0.15593419792632227 s)
  512 READ  7.936213897584876 MB/s
  512 WRITE 128000000 B, 8.783565597 s (0.3087759709139298 s)
  512 WRITE 13.897580789024076 MB/s
  512 RW  128000000 B, 10.185301735499998 s (0.14414745586564162 s)
  512 RW  11.984948082051842 MB/s
```

C.4.2 Non-Contiguous Access — 10 Operations

```
NoCache
  512 READ  128000000 B, 1.1791882332999999 s (0.04768848491539928 s)
  512 READ  103.52063313791889 MB/s
  512 WRITE 128000000 B, 1.1786016051999997 s (0.04364732414330116 s)
  512 WRITE 103.57215870182495 MB/s
  512 RW  128000000 B, 1.1935244082 s (0.04876219865044266 s)
  512 RW  102.27718148144027 MB/s
SimpleWriteBehindCache
  512 READ  128000000 B, 1.1791882332999999 s (0.04768848491539928 s)
  512 READ  103.52063313791889 MB/s
  512 WRITE 128000000 B, 1.1790767756 s (0.04043855024574582 s)
  512 WRITE 103.53041890582719 MB/s
  512 RW  128000000 B, 1.1769992763 s (0.04505433253442518 s)
  512 RW  103.71315850230486 MB/s
AggregationCache
  512 READ  128000000 B, 1.1791882332999999 s (0.04768848491539928 s)
  512 READ  103.52063313791889 MB/s
  512 WRITE 128000000 B, 1.1755616720000002 s (0.04262144657833614 s)
  512 WRITE 103.83999020002072 MB/s
  512 RW  128000000 B, 1.1769992763 s (0.04505433253442518 s)
  512 RW  103.71315850230486 MB/s
ServerDirectedIO
  512 READ  128000000 B, 1.2360401953 s (0.05519337068379282 s)
  512 READ  98.75917705926403 MB/s
  512 WRITE 128000000 B, 1.1049812159 s (0.040050454423372725 s)
  512 WRITE 110.47274898747895 MB/s
  512 RW  128000000 B, 1.1411116579000002 s (0.03621351549579512 s)
  512 RW  106.97490614077792 MB/s
```

C.4.3 Non-Contiguous Access — All Operations

```
NoCache
  512 READ  128000000 B, 1.1543980794000002 s (0.0394784391535595 s)
  512 READ  105.74368987468014 MB/s
  512 WRITE 128000000 B, 1.2848065985 s (0.04108401400612664 s)
  512 WRITE 95.01065190863433 MB/s
  512 RW  128000000 B, 1.2361368582 s (0.06623419477593881 s)
  512 RW  98.75145433148285 MB/s
SimpleWriteBehindCache
  512 READ  128000000 B, 1.1543980794000002 s (0.0394784391535595 s)
  512 READ  105.74368987468014 MB/s
  512 WRITE 128000000 B, 1.2828997951 s (0.04004674110205048 s)
  512 WRITE 95.15186842046757 MB/s
  512 RW  128000000 B, 1.2378648553 s (0.0662451082669207 s)
  512 RW  98.61360226631196 MB/s
AggregationCache
  512 READ  128000000 B, 1.1543980794000002 s (0.0394784391535595 s)
  512 READ  105.74368987468014 MB/s
  512 WRITE 128000000 B, 1.1627331322 s (0.04246794069729553 s)
  512 WRITE 104.9856662027266 MB/s
  512 RW  128000000 B, 1.1596055462000003 s (0.05061499832368804 s)
  512 RW  105.26882429979875 MB/s
ServerDirectedIO
  512 READ  128000000 B, 1.1086887981 s (0.03897905988644655 s)
  512 READ  110.10331547427583 MB/s
  512 WRITE 128000000 B, 1.1100156154999998 s (0.036601560409424215 s)
  512 WRITE 109.97170742054305 MB/s
  512 RW  128000000 B, 1.1253318360000002 s (0.051472910533868585 s)
  512 RW  108.47494809522121 MB/s
```

List of Acronyms

ATA	Advanced Technology Attachment
CPU	Central Processing Unit
FIFO	First In, First Out
GPFS	General Parallel File System
GPL	GNU General Public License
HDD	Hard Disk Drive
I/O	Input/Output
IPS	Instructions per Second
KB	Kilo Byte (1 000 Bytes)
KiB	Kibi Byte (1 024 Bytes)
LGPL	GNU Lesser General Public License
MB	Mega Byte (1 000 000 Bytes)
MiB	Mebi Byte (1 048 576 Bytes)
MPD	Multi-Purpose Daemon
MPI	Message Passing Interface
MPI-IO	Message Passing Interface Input/Output
NIC	Network Interface Card
NCQ	Native Command Queuing
OS	Operating System
PDF	Portable Document Format
PIOsimHD	Parallel I/O Simulator

POSIX	Portable Operating System Interface (for Unix)
PVFS	Parallel Virtual File System
RAID	Redundant Array of Inexpensive Disks *or* Redundant Array of Independent Disks
RAM	Random Access Memory
RPM	Revolutions Per Minute
SAN	Storage Area Network
SATA	Serial Advanced Technology Attachment
SCSI	Small Computer System Interface
SLOC	Source Lines of Code
SSD	Solid State Drive
SVN	Subversion
UI	User Interface
ZFS	Zettabyte File System

List of Figures

List of Tables

List of Listings