Simulation-Aided Performance Evaluation of Server-Side Input/Output Optimizations

Michael Kuhn
University of Hamburg
michael.kuhn@informatik.uni-hamburg.de

Julian M. Kunkel
University of Hamburg

Thomas Ludwig
University of Hamburg

Abstract

The performance of parallel distributed 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.

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

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

This paper compares existing client-side optimizations and newly implemented server-side optimizations and evaluates their usefulness for I/O patterns commonly found in HPC. Server-directed I/O chooses the order of non-contiguous 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 results show that server-side optimizations beat client-side optimizations in terms of performance for many use cases. Integrating such optimizations into parallel distributed 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.

1. Introduction

Parallel distributed 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. This is especially true for large amounts of small requests. 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.

These approaches 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 can be used to cache and batch I/O operations to reduce the load on the servers. Other approaches do not perform preprocessing on the clients and let the servers handle all the work themselves. The servers can then employ their own optimizations.

2. State of the Art and Related Work

Traditionally, sequential programs access data in contiguous regions, identified by an offset and a size. *Non-contiguous* I/O enables applications to access several of these regions without actually requesting each of them separately. This can significantly reduce the overhead introduced by network latency, because multiple operations can be batched together. Using non-contiguous I/O to perform such operations may also give the underlying file system – or a high-level I/O library – the opportunity to optimize these kinds of operations.

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, clients can collaborate, allowing the I/O library to perform optimizations that are impossible to do with individual I/O.

The *Two-Phase* protocol is an optimization for collective I/O implemented in ROMIO, which allows clients to collaborate during I/O. It uses separate communication and I/O phases to optimize I/O accesses of multiple clients, but introduces additional communication overhead. The implementation in ROMIO mainly focuses on making the accesses contiguous.

Simpler client-side optimizations like data sieving and collective I/O are presented in [7]. The usefulness of non-contiguous and collective I/O is described in [8]. Additionally, the four levels of I/O needs in MPI-I/O are introduced. Increasing MPI-I/O levels allow for more advanced optimizations. In [1], ROMIO is extended to use the list I/O interface of PVFS[1] to provide better performance for non-contiguous I/O.

A refined and extended version of the Two-Phase protocol called Multiple-Phase Collective I/O is presented in [5, 6]. 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, which increases the performance of the subsequent I/O phase.

An alternative to client-side optimizations – in particular, to the Two-Phase protocol – is presented in [2]. *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. In the presented scheme, the clients still need to issue a collective call. Once all clients started the collective operation, one of them sends the full non-contiguous requests to the I/O servers. Each server sorts contiguous accesses of all clients, processes them sequentially and sends the results back to the client that started the operation. 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. In contrast, our *server-directed I/O* allows clients to perform non-contiguous I/O independently. Additionally, the server aggregates multiple requests into larger access for the I/O subsystem.

Our goal is to analyze whether comparable performance results can be achieved by avoiding complex client-side optimizations and moving the necessary logic into the file system itself.

3. Software Environment

HDTrace is a framework containing all components necessary to simulate, trace and visualize applications. The framework is described here briefly.

The simulator *PIOsimHD* – part of HDTrace – allows simulating arbitrary network topologies, servers and client applications. PIOsimHD is designed to run programs conforming to the MPI standard, including asynchronous communication and collective operations. One of its goals is to allow easy and fast prototyping of new algorithms for I/O optimization. Simple applications can be implemented directly within PIOsimHD for fast testing of new algorithms. All optimizations presented in this paper are implemented in the 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.

[1] PVFS: `http://www.pvfs.org/`

3.1. PIOsimHD I/O Model

An abstract parallel distributed file system defines how the interaction between clients and servers takes place. File data is partitioned among all servers as defined by a selectable distribution function. Metadata operations are not considered yet. Data is read and written from disk in chunks with a maximum size of the I/O granularity, which defaults to 10 MiB. Data sent across the network is split up into smaller chunks to accommodate the network granularity, which in turn defaults to 100 KiB. The cache layer explicitly manages the free memory as a cache. Non-contiguous I/O requests are explicitly supported.

Clients and servers interact in a similar fashion to the PVFS model: In the write path, a client announces a write operation to the server and then starts to transfer all data. The server acknowledges the completion of the write operation when it is finished. File sizes are updated once a write operation finishes. The network flow to a server is stopped if there is no more cache available. Consequently, the client write activity is stopped if the I/O subsystem can not write back data fast enough and thus fills up the cache. Whenever a data packet is received on the server, it is forwarded to the cache layer. It is important to note that the effective I/O granularity is equal to the network granularity if the cache layer does not combine I/O requests.

In the read path, the client posts a non-contiguous request to all servers. This information is provided completely to the cache layer. The cache layer of a server can then decide in which order the operations are to be performed. Data is then read from disk. The I/O granularity limits the maximum size per operation. Once data is ready, it will be transferred to the client. Note that a buffer is managed to store the data until the transfer is completed.

3.2. Cache Layers

The following cache layers are currently implemented in PIOsimHD:

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.

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.

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 paper, because they do not perform any overlapping accesses. 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.

To put these cache layers into relation with existing file systems, the comparison with PVFS is as follows: In PVFS, the normal buffer size per I/O operation is 256 KiB. Only a subset of reads is announced to the I/O subsystem. Effectively, large reads are fragmented, which might cause the access pattern to look like random accesses when multiple clients use the file system concurrently. This, in turn, can cause a serious performance degradation. The PVFS read performance can be compared to the `NoCache`. As Linux performs write-behind and some sort of aggregation the observable write performance is a bit lower than the one achieved with the `AggregationCache`.

4. Design

The main problem with client-side optimizations is the fact that these optimizations usually do not have enough information to efficiently optimize operations. 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. When using server-side optimizations like the aggregation cache and server-directed I/O the servers themselves decide which operations they should perform next and in which order they should be performed.

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. Effectively, this is comparable to the server-directed approach. Even though the server does not choose which read to issue next, it combines this operation with all overlapping operations in the queue.

4.2. Server-Directed I/O

The server-directed I/O implementation is different from the `AggregationCache`, because it additionally reorders I/O operations. A large number of clients performing many small operations can easily saturate the I/O system. The optimizations implemented in the server-directed I/O cache layer try to satisfy as many client 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 for read operations 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.

For write operations, this is done by discarding unnecessary write requests early and merging 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.

Note that this reordering does not violate the consistency requirements usually found in file systems, since read operations can still return the most recent data from the buffer.

5. Evaluation

The simulated 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 instructions per second 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 0.2 ms. 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. This limits the maximum I/O throughput to 500 MiB/s. The HDDs have an average seek time of 10 ms and a track-to-track seek time of 1 ms. Whenever 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 0. The striping size is set to 64 KiB, which corresponds to PVFS's default striping size.

Additionally, all clients and servers have empty caches at the beginning of each test. Consequently, all data has to be read from the underlying storage device. This means that neither the read nor the write throughput can exceed 500 MiB/s.

As write operations are issued in the network granularity, write performance can be worse than the read performance. This is due to the fact that 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 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.

5.1. Individual and Collective I/O

This comparison uses individual and collective I/O operations. The accessed file is 1.000 MiB in size and divided into data blocks of equal size. The benchmark is executed with varying data block sizes.

In the individual case, either one data block (*Ind.*), 100 data blocks (*Ind. (100)*) or all data blocks (*Ind. (All)*) are accessed per iteration. In the collective case, each client performs only one collective operation to access 1/10 of all data blocks at once. Both the original Two-Phase (*TP*) as well as the Interleaved Two-Phase (*I-TP*) [4] protocol are used. This I/O pattern resembles the I/O patterns often found in High Performance Computing applications, where iterative algorithms perform I/O every n iterations. All tests are performed with the `NoCache` (*NC*), `SimpleWriteBehindCache` (*SWBC*), `AggregationCache` (*AC*) and `ServerDirectedIO` (*SDIO*) cache layers.

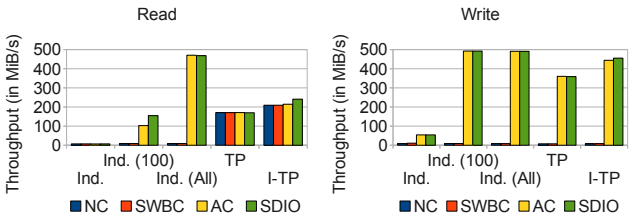

Figure 1. Summary – 5 KiB Data Blocks

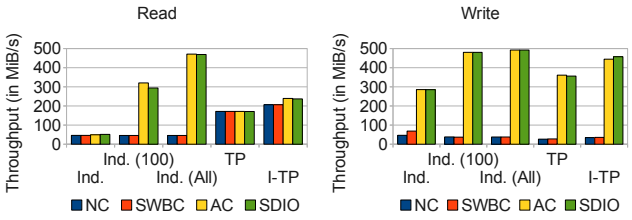

Figure 2. Summary – 50 KiB Data Blocks

Figure 1 shows all results obtained for read and 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. Batching operations results in performance gains. For read operations, 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. For write 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. Individual I/O operations with the `AggregationCache` and `ServerDirectedIO` cache layers beat both Two-Phase implementations. For write operations, the Two-Phase implementations do not perform well when used with non-optimizing cache layers. This is due to the fact that 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.

Figure 2 shows all results obtained for read and write operations with a data block size of 50 KiB. The results look

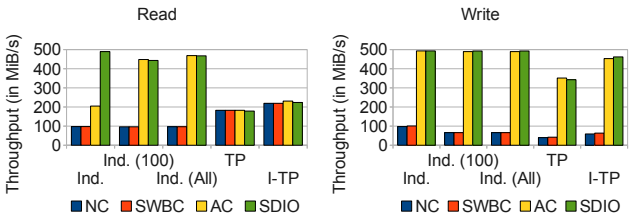

Figure 3. Summary – 512 KiB Data Blocks

like the ones in figure 1 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 3 shows all results obtained for read and write operations with a data block size of 512 KiB. The results look like the ones in figure 2 except for the higher performance with non-optimizing cache layers, which is due to the larger data block size.

Overall, the `ServerDirectedIO` cache layer delivers the maximum performance in almost all tests using individual I/O operations.

6. Conclusion

The results from section 5 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 tested cases. 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.

The results also suggest that even simple server-side optimizations like those in the `AggregationCache` cache layer are better suited for use cases often found in HPC. Integrating such optimizations into parallel cluster file systems could alleviate the need for sophisticated client-side optimizations.

More information, benchmarks and evaluations can be found in [3], which also includes random, database and real-world workloads. These were omitted in this paper due to space constraints. Additionally, visualizations of low-level client and server activities were conducted to analyze the influence of the different optimizations in detail.

7. Future Work

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.

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. However, this could be easily rectified by limiting the amount of memory used to buffer read and write operations. To guarantee fairness, it is possible to add support for priorities, which could then be used to make sure that no operation waits in the queue for too long by raising the priority based on the time an operation has been waiting.

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.

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.

References

[1] A. Ching, A. Choudhary, K. Coloma, W.-k. Liao, R. Ross, and W. Gropp. Noncontiguous I/O Accesses Through MPI-IO. In *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, CCGRID '03. IEEE Computer Society, 2003.

[2] D. Kotz. Disk-directed I/O for MIMD multiprocessors. *ACM Transactions on Computer Systems*, 15(1), 1997.

[3] M. Kuhn. Simulation-Aided Performance Evaluation of Input/Output Optimizations for Distributed Systems. Master's Thesis, Ruprecht-Karls-Universität Heidelberg, 09 2009.

[4] M. Kuhn, J. Kunkel, Y. Tsujita, H. Muguruma, and T. Ludwig. Optimizations for Two-Phase Collective I/O. To be published (ParCo 2011).

[5] D. E. Singh, F. Isaila, A. Calderon, F. Garcia, and J. 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*. IEEE Computer Society, 2007.

[6] D. E. Singh, F. Isaila, J. C. Pichel, and J. Carretero. A collective I/O implementation based on inspector–executor paradigm. *The Journal of Supercomputing*, 47(1), 2009.

[7] R. Thakur, W. Gropp, and E. 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*. IEEE Computer Society, 1999.

[8] R. Thakur, W. Gropp, and E. Lusk. Optimizing Noncontiguous Accesses in MPI-IO. *Parallel Computing*, 28, 2002.