Impact of I/O Coordination on a NFS-based Parallel File System with Dynamic Reconfiguration

Rodrigo Virote Kassick, Francieli Zanon Boito, Philippe O. A. Navaux
Instituto de Informática – Universidade Federal do Rio Grande do Sul
{rvkassick, fzboito, navaux}@inf.ufrgs.br

Abstract

The large gap between processing and I/O speed makes the storage infrastructure of a cluster a great bottleneck for HPC applications. Parallel File Systems *propose a solution to this issue by distributing data onto several servers, dividing the load of I/O operations and increasing the available bandwidth.*

However, most parallel file systems use a fixed number of I/O servers defined during initialization and do not support addition of new resources as applications' demands grow. With the execution of different applications at the same time, the concurrent access to these resources can impact the performance and aggravate the existing bottleneck.

The dNFSp File System *proposes a reconfiguration mechanism that aims to include new I/O resources as application's demands grow. These resources are standard cluster nodes and are dedicated to a single application.*

This paper presents a study of the I/O performance of this reconfiguration mechanism under two circunstances: the use of several independent processes on a multi-core system or of a single centralized I/O process that coordinates the requests from all instances on a node. We show that the use of coordination can improve performance of applications with regular intervals between I/O phases. For applications with no such intervals, on the other hand, uncoordinated I/O presents better performance.

1 Introduction

Cluster architectures and parallel programing are nowadays the standard solution to develop high performance applications. These applications usually access and generate large datasets. Due to the volume and characteristics of these datasets, there is need for a shared storage infrastructure.

On the other hand, the speed of permanent storage is very slow when compared with that of the computing resources. This gap makes the storage system a very significant bottleneck, causing severe contention for parallel applications. *Parallel File Systems* (also called PFS) offer a solution to this issue using several independent I/O devices spread over a set of *data servers*. In this manner, the load of I/O requests is distributed over independent resources, aggregating disk and network bandwidth.

The number of data servers used in a PFS will influence directly its performance. Defining the number of servers to use, though, is no trivial task, given that applications to be executed on a cluster and their demands may be unknown during the setup phase. Additionally, the usage scenario may change during the cluster lifetime. Overestimating the number of I/O resources to fit any imaginable scenario may result in waste of money and resources.

Several adaptation strategies can be applied to parallel file systems. Pre-fetching and caching can be optimized to match application's I/O request patterns [21, 17], data can be distributed according to application's dominant spacial access patterns [3, 11], the number of servers to use can be adapted to the output characteristics of applications [15], etc. The total number of storage devices available, though, is fixed during system's operation.

When multiple applications execute over the same shared storage system, there can be contention over the access to data. When all data servers are in use by at least one application, starting a new one will decrease the individual performance of the ones already in execution. In such cases, it may be necessary to increase the number of available storage resources in order to increase the total capacity of the file system.

The *dNFSp File System* proposes a reconfiguration mechanism that aims to include new I/O resources as applications' demands grow. These resources are standard cluster nodes and are dedicated to a single application, granting to an I/O-demanding application exclusive access to a number of storage resources.

With the popularity of multi-core processors, several independent instances of an application will execute on a single client of the file system. In this situation, every instance

will have its own data to be written or read from the shared file system. Parallel I/O APIs like MPI-I/O provide mechanisms to aggregate data from these processes into a single stream to create long sequential requests and profit from file system optimizations. This paper presents the behavior of dNFSp reconfiguration mechanism when used with several independent I/O processes on each node and when data is combined to avoid concurrency.

The remaining of this paper is divided as follows: Section 2 introduces dNFSp, our working platform. The reconfiguration model is presented on Section 3 and the results with and without data aggregation are discussed in Section 4. Section 5 presents related works and state of the art. Finally, Section 6 presents the conclusions and future works.

2 dNFSp

dNFSp — distributed NFS parallel — is an extension of the traditional NFS implementation that distributes the functionality of the server over several machines on the cluster [2]. The idea behind dNFSp is to improve the performance and scalability without sacrificing compatibility with standard NFS clients – thus simplifying setup and maintenance.

The NFS server functionality is divided onto several meta-data servers (named *meta-servers* or *metas*) and data servers (named *IODs*). Clients mount a volume in one meta-server, so each meta-server handles only a subset of the clients. In dNFSp, if all the clients access the file system at the same time, the load will be distributed among these servers [12].

Upon receiving a request, the meta-server checks for data-distribution information on the file and forwards the request to the correct IOD. The IOD manages the raw data on the disks. Once it receives a read or write request, it performs the operation on the file corresponding to the *inode number* of the request and sends replies (either acknowledge messages or read replies) directly to the clients. *IP Spoofing* is used to trick the clients into thinking that the answer came from the NFS server, keeping full compatibility with the protocol.

In dNFSp each request must be transmitted twice – from client to meta-server and from meta to IOD. This is necessary to keep compatibility with the NFS protocol. For *read operations*, this strategy presents good performance, since the overhead is easily compensated by the higher I/O throughput of the IODs. This situation is illustrated in the lower half of Figure 1. On the other hand, *write* performance is impaired by the re-transmission of large data blocks. This situation is illustrated on the upper part of Figure 1.

When several applications make intensive use of the shared file system, the I/O performance of each application will be tainted by the concurrent access to the limited number of servers. For applications that must execute before specific deadlines it may be impossible to complete some computation if the file system does not present the expected performance.

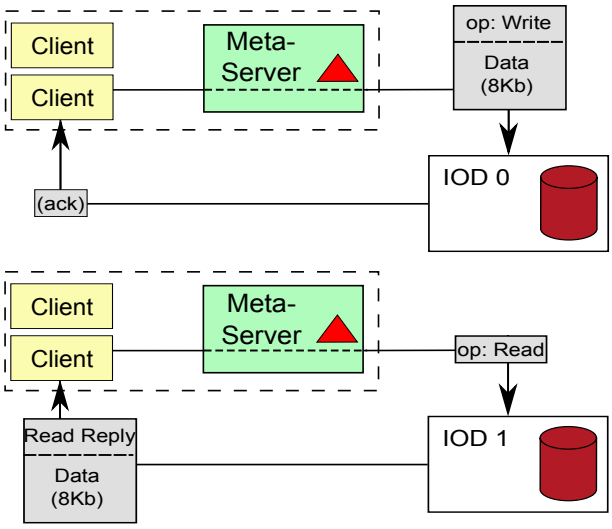

Figure 1. dNFSp Architecture

Application IODs [9] (AppIOD) try to solve this by associating exclusive I/O servers to a set of clients that executes an application. Meta-servers identify requests from these clients and forward them to the exclusive IODs instead of the standard ones. AppIODs do not perform initial synchronization. Instead, requests for data that was not written on the exclusive servers is forwarded to the original ones. After the application finishes, data from the AppIODs can be merged with the one previously stored on the original servers.

In this model, the user must be aware of the application I/O demands and reserve extra nodes to be used as I/O servers. In a cluster that executes several applications, it is not always possible to know if the file system will provide the expected performance. To avoid the need of direct user intervention, the file system must be able to identify applications that could profit from new I/O resources and do the proper reconfiguration during the execution.

3 Automatic I/O Reconfiguration

To avoid user intervention, an automatic reconfiguration mechanism for a PFS needs to be able to recognize that performance is impaired, take actions to solve the causes of the poor performance and avoid that any reconfiguration increase the existing overhead. In this section, we will present the reconfiguration model adopted for dNFSp.

3.1 I/O Server Creation

In dNFSp, *write operations'* performance is limited due to the double transmission of data. Applications dominated by this kind of operation will suffer with the slow I/O bandwidth. Because of this, we choose to focus on improving the performance for write phases.

For write-bound applications, the utilization of AppIODs can increase performance and the cost of server insertion is very low, since there is no replication of data. While there is overhead on the original servers of the system during the merge phase, this task can be postponed until there are few or none applications actively using the system or until the machines used as AppIOD are claimed to be used for other tasks.

As in the original implementation of AppIODs, our reconfiguration model considers that any node of the cluster that is not allocated for an application (i.e. is marked as *free* by the cluster scheduler) is a suitable machine to act as an AppIOD. Once the nodes are selected to act as I/O servers, the scheduler is notified and marks the resources as *busy* until the scheduled end of the application.

3.2 Triggering the Reconfiguration

In dNFSp, any file big enough will be distributed among all available IODs. In this scenario, when a single application uses the file system it may obtain the greatest performance available from the servers. The inclusion of new servers in this case would require merging the data written in the AppIODs and the original servers.

When two or more applications make concurrent requests to the file system, though, the performance obtained by either of them will be a fraction of the peak performance. The inclusion of more I/O resources may bring significant gains in this situation, since independent sets of I/O servers will be responsible for the requests of each.

On the other hand, starting a new application is not reason to include new servers on the system. Parallel applications may present temporal behavior, i.e. distinct I/O and processing intensive phases (*idleness phases*, since the PFS observes few requests from the clients). The length of the I/O and processing phases are part of the *temporal access pattern* of an application.

Overlapping I/O phases is a better metric in this case, since it postpones resource utilization until more than one application actually uses the system. When I/O phases overlap, the observed performance of the applications will be impaired due the concurrency. Figure 2 illustrates this situation. During the period before t_1, the application A_0 executes exclusively over the available servers. In t_1, another application A_1 starts using the file system. During the period of time between t_2 and t_3, the two applications share the available I/O bandwidth, obtaining individual bandwidths below the previously obtained. During this time, the system is called *saturated* – i.e. it has reached its peak performance and any more clients accessing it will result in yet lower performances for each application. After the instant t_3, A_0 stops using the file system and, as a result, A_1 can make full use of the available resources.

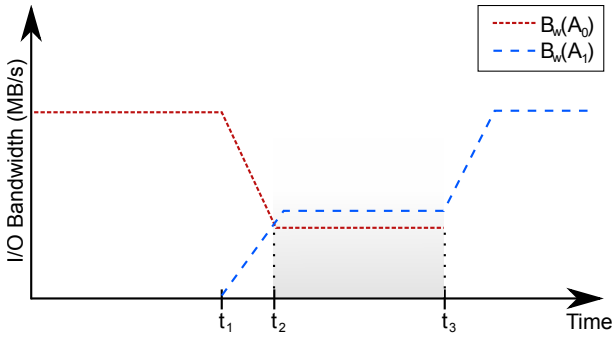

Figure 2. I/O Bandwidth for two concurrent applications. t_1: A_1 starts. t_2: Bandwidth for A_0 and A_1 is stable. t_3: A_0 ends.

During period highlighted in gray, the reconfiguration can take place, given that there are enough resources and that at least one of the applications is *write-bound* – i.e. the original servers of the system will be offloaded from the write requests of one application.

It is important to note that the variations in I/O bandwidth may be caused by events other than application behavior. The reconfiguration system has some parametric safe-guards to avoid unnecessary reconfigurations – decreases in bandwidth that do not last longer than a limit or that do not differ more than a fraction of the best performance observed.

Figure 3 shows the measured bandwidth during the execution of two applications. The application represented by the dashed line presents temporal behavior, while the one represented by the continuous line has only a long I/O phase. The period represented by *(a)* shows a *performance-stable period* in which the bandwidth for the continuous application did not present significant variation. As the temporal application enters a phase with few requests, the reconfiguration model detects a new *peak performance* in *(b)*, i.e., the bandwidth that an application can expect from the file system. With the start of a new I/O phase in *(c)*, the continuous application has a decrease in its performance and the system is thus considered *saturated* in *(d)*. A stable, or saturated period, as well as the peak performance, are only considered if the system does not present significant variations in bandwidth during a pre-defined time (1min, in the case of the example). This acts as safeguard for variations

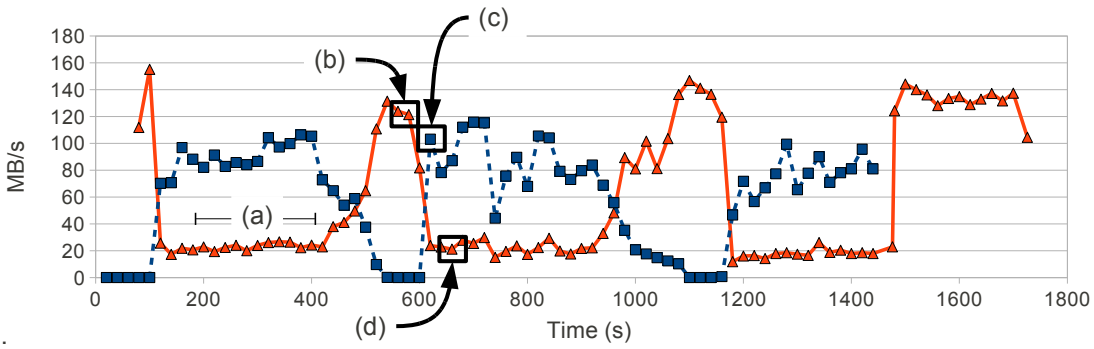

Figure 3. Bandwidth graph for two applications. *(a)*: **Stable period;** *(b)* **Peak Performance;** *(c)* **Start of new I/O Phase;** *(d)* **Saturated performance**

not related to application behavior.

3.3 Application Selection

Since AppIODs redirect all the requests from clients of an application to an exclusive set of servers, the model must choose an application to receive the servers whenever the PFS is saturated. This application must attend the two following requisites:

1. **Be dominated by write operations**, so that most of its requests will be forwarded to the exclusive servers;
2. **Have a known factor** β. This value, called *boundness*, is a ratio of the length of I/O phases over the execution time of the application.

The value of β is a ratio that indicates whether the application presents very long I/O phases (β tends to 1) or if during most of its execution time it would leave the PFS idle. This value is fed to our model as a parameter for each application, since detecting the existence of stable I/O phases and the characteristics of the access patterns of applications is studied in other works [21, 5, 20] and considered outside our scope.

The reconfiguration model selects the application with the greatest *TotalGain*, described in Formula 1. In the formula, $T_r(A_i)$ represents the remaining execution time for an application A_i, $\beta(A_i)$ is the *boundness-ratio* and n_{AppIOD} is the number of exclusive servers to be used. This formula represents the time slot that would be freed by the application if its execution was shortened by the better performance in I/O, minus the cost of utilizing a number of extra resources during the remainder of the application.

$$TotalGain\,(A_i) = \begin{array}{r} \#Nodes(A_i) \times \beta(A_i) \times T_r(A_i) \\ -n_{AppIOD} \times T_r(A_i) \end{array} \quad (1)$$

3.4 Implementation

The proposed reconfiguration model was implemented on dNFSp as a monitoring library – *libPFM* – that collects, for each application, the amount of data transferred. In regular intervals, the *PFM Daemon* communicates with all the meta-servers and collects the partial performance information. For each application, the daemon stores a performance history vector, containing the aggregated bandwidth collected from the meta-servers.

With this information, as well as information about the behavior of the applications, the daemon monitors the performance of dNFSp and, whenever the system performance is considered *saturated* and there are applications that fit the selection criteria, it spawns AppIODs on available cluster nodes.

To request new resources and spawn the exclusive servers, the PFM Daemon must integrate with a scheduler system. We have chosen to implement a simple scheduler called VSS. This scheduler was configured to read submission traces and start parallel applications on a set of nodes of a cluster. For each application, VSS informs dNFSp of the nodes that were allocated to it.

When the PFM daemon decides to spawn AppIODs, it makes a special job submission to VSS, indicating it needs I/O resources for the selected application. VSS will then allocate available nodes, start the necessary processes on them and associate the I/O job with the original application job. This way, when the application finishes, the file system is notified and can take action to either synchronize data and release the exclusive servers or postpone this task to a later time.

The code for dNFSp, libPFM, PFM Daemon and VSS is available on dNFSp project site at `http://sourceforge.net/projects/nfsp/`.

4 Evaluation and Results

4.1 Test Setup

The tests were executed on the *Pastel* cluster of Grid5000 [4], composed by 80 Sun Fire X2200 M2 nodes, each with $8GB$ of RAM and two dual core *2.6GHz* AMD Opteron 2218 processors. The nodes are connected by Gigabit Ethernet network through a Cisco 7006 router. Each node uses an Hitachi HDS7225S SATA disk of *250GB* as primary storage. The brute bandwidth measured on these disks was of $62.69MB/s$[1] for reads and *34.67MB/s*[2] for writes.

We used 4 nodes as servers, each machine executing one instance of the meta-server and one IOD. Clients were evenly distributed on the meta-servers, with mount options *wsize* and *rsize* of $8KB$. The number of application IODs used during the reconfiguration was also 4.

To simulate the load of parallel applications on the file system, we used the MPI-IO Test 21 benchmark [16]. This program uses MPI to execute a parallel application onto several nodes. The benchmark can use either MPI-IO or Posix primitives to simulate applications' accesses to the file system. In our tests, we used the benchmark with one file per process and standard POSIX calls. The benchmark was modified to include a fixed delay between each *read* or *write* operation. This delay aims to simulate the behavior of applications with temporal access pattern, forcing an *idleness phase*.

We executed two concurrent applications. The first one, A_0, using 256 processes over 32 nodes, executes during 20 minutes and presents temporal behavior. The second application, A_1, starts 2 minutes after the first one and executes for 40 minutes, with 64 processes over 8 nodes.

The two applications were executed on two different scenarios. On the *multiple-writer* test, each node executed 8 instances of the benchmark, with A_0 writing $128MB$ in its I/O phases. A_1 writes up to $2GB$ of data on $1MB$-writes. In this test, A_1 imposes no coordination between clients.

On the *single-writer* test, only one instance was executed on each node to simulate the behavior of collective I/O operations present in APIs like MPI-IO. In this case, each I/O operation presents an implicit barrier and data is aggregated on intermediate nodes before being sent to the PFS. Our single-writer tests simulate this situation with one data-aggregator node per client. We prefer not to use the collective MPI-IO operations to avoid API specific costs of aggregating data. This allows us to focus exclusively on the I/O performance of the file system, disregarding the API.

To keep the same behavior as in the multiple-writer case, the write phases of A_0 in the sigle-writer scenario produce *1GB* of data, i.e. 8 times the write size in each instance of multiple-writers test. In A_1, each client writes up to $16GB$ per node on $8MB$-writes. This test introduces barriers between write operations to emulate the gathering of data on a single instance of the application per node.

Each test was repeated from 6 to 10 times, so that standard deviation was not larger than 10% of the average.

[1] Value obtained by average of 6 executions of *hdparm*

[2] Value obtained with the *dd* command, no file system, all caches off

4.2 Results with 128MB Object-size

This section presents the results for the concurrent execution of applications 0 and 1 with 128MB object-size on A_0. Figure 4 shows the bandwidth for A_0 and A_1 for the single and multiple-writer tests. A_0 was executed with different values for the idleness interval I (10, 20, 40, 60 and 80 seconds), represented in the x-axis of the figure.

In this test, the use of a single writer presented better performance for A_0 when compared to the multiple-writers case. In part, this can be attributed to a loss in performance in A_1: with idleness intervals longer than 10 seconds, using a single writer in A_1 provided lower bandwidths than with multiple writers.

On the other hand, there seems to be no direct relation between the lower performance in A_1 and the better performance on A_0. For an idleness interval of 10 seconds on A_0, both applications presented higher throughput with the single-writer strategy. For other values of I, the decrease in A_1's performance is larger than the gains in A_0.

4.3 Results with 8MB Object Size

This section presents the results for the single and multiple writer tests for A_0 using objects of $8MB$. On the multiple-writers scenario, A_0 executes 8 instances of the benchmark on each of the 32 nodes, writing objects of $8MB$ on each I/O phase. On the single-writer scenario, there is a single instance of the benchmark on each node and the I/O phases produce $64MB$-objects. A_1 behaves as in the previous benchmark.

Figure 5 presents the results for this test. On the multiple-writers scenario, A_1 always presents better I/O performance than A_0. Since the number of clients and the write operations size are smaller on A_1, we can conclude that the presence of short I/O phases followed by the idleness interval on A_0 cause A_1 to dominate the use of the file system, even though the reconfiguration mechanism detects contention on A_0 and dedicates more I/O resources accordingly.

The use of a single writer greatly improved the performance for A_0 with all tested idleness intervals. With $I = 40$ to 80, A_1 presented lower performance on the single-writer scenario. Again, gains in A_0 with a single writer are not directly related to the loss in performance of A_1.

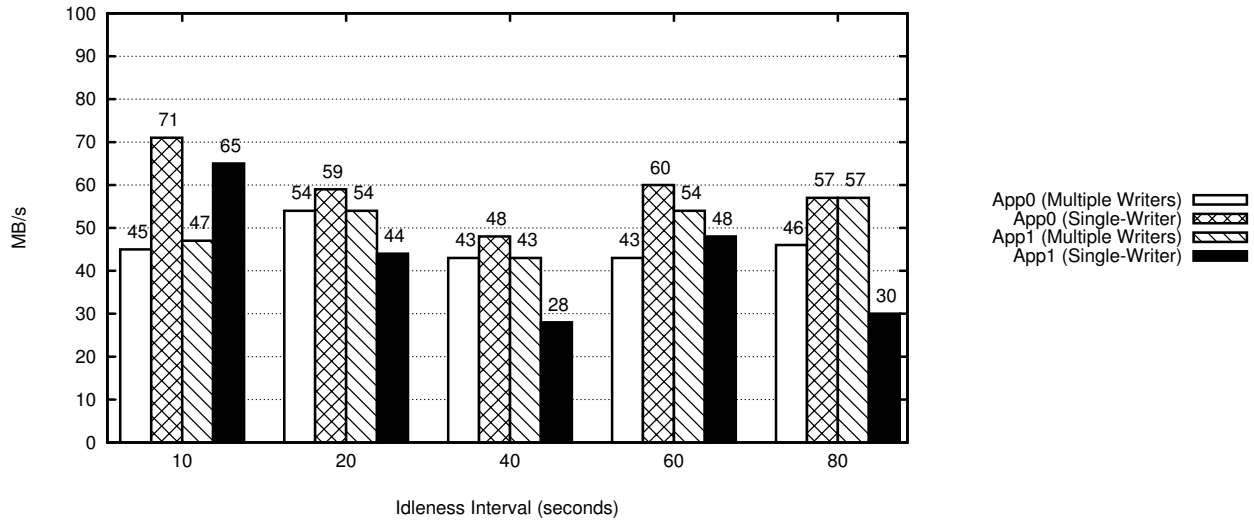

Figure 4. I/O Performance for Single-Writer and Multiple-Writer scenarios with Automatic Reconfiguration, 128MB Write Size.

4.4 Discussion over the Results

In the tests presented in this section, the reconfiguration mechanism detected contention on the I/O of the applications and included new I/O resources. Due to the object sizes, A_1 receives the new resources only on the test with $8MB$-object-size on A_0 and $I \geq 40$; otherwise, A_0 received the new servers. In both cases, each application had access to exclusive I/O resources after some point of the execution.

For the first test, the loss of performance in A_1 with a single writer can be attributed to the presence of a dominant application with temporal behavior. The longer I/O phases on A_0 avoid A_1 to reach high throughput during the overlapping I/O phases. When A_1 runs without coordination, some of its processes manage to profit from the idle interval of A_0 and improve the overall performance. With the inclusion of barriers, this reaction has been slowed down and thus A_1's performance has been worse in the single-writer scenario.

On the second test, the influence of temporal behavior over the NFS protocol has been exacerbated by the smaller object sizes on A_0. As a result, with uncoordinated I/O, A_1's performance has been much higher than A_0's, since the former was able to profit from the idleness intervals and kept A_0 from reaching higher troughput during its I/O phases. In this case, the use of coordination has been able to increase the performance of A_0. This can be attributed to the larger write sizes and to the creation of barriers on A_1, as previously stated.

Despite decreasing the performance for I/O patterns with no temporal behavior, the use of single-writers has been able to grant some level of fairness on the use of the shared file system.

5 Related Work

Cluster storage, being an important aspect for parallel application's performance, is studied in several works on the literature. In all of them, we can observe different approaches to provide high efficiency and scalability, both in number of servers and clients. Some of them make use of adaptation to specific characteristics of applications and file systems, done either during its execution or prior to it.

Kunkel [14] proposes an on-line monitoring library with data migration capability for the *Parallel Virtual File System* (PVFS) [6]. When load unbalances are detected, data is redistributed to improve load balancing. This redistribution can increase the I/O performance of the file system, but imposes the price of moving the data from an already overloaded server.

Lustre File System [18] is a well known solution for data management in cluster environments, being used in several Top-500 ranking clusters. On Lustre, servers can be grouped based on common characteristics – e.g. network interconnection, I/O device speed, storage space, etc [19]. Applications can require that newly created files be stored only on servers of a given *pool*, using the resources that better fit their needs – fault-tolerant I/O, largest storage area, fastest interconnection, etc. On the other hand, Lustre only

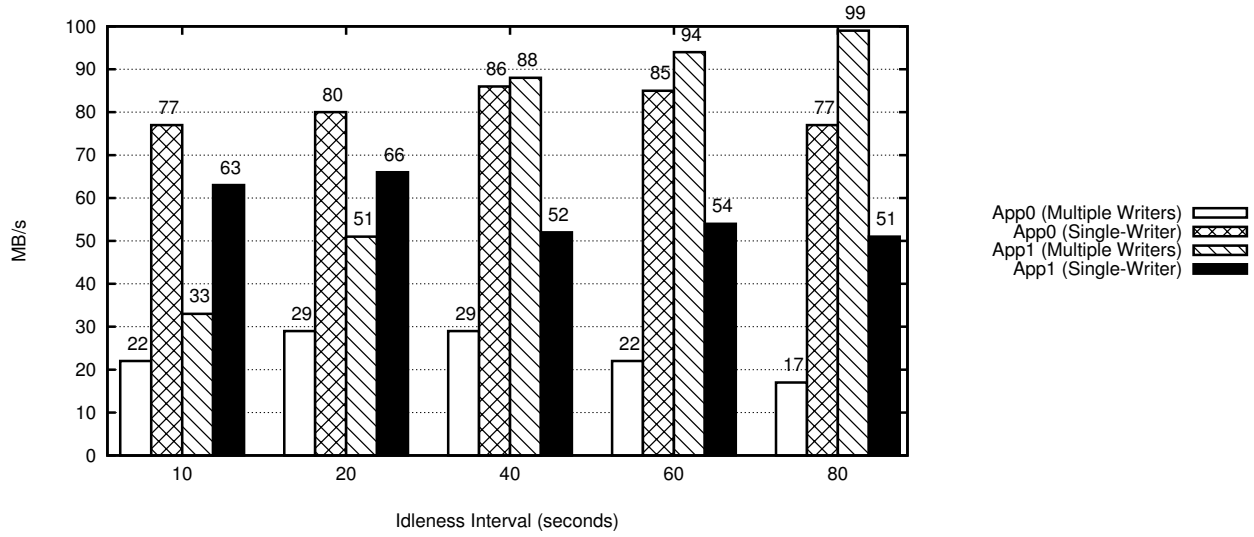

Figure 5. I/O Performance for Single-Writer and Multiple-Writer scenarios with Automatic Reconfiguration, 8MB Write Size.

supports this kind of adaptation during file creation, lacking *dynamic* adaptation to application characteristics and changes in the environment that impact I/O performance, like new applications or RAID maintenance.

Solutions like Google File System [8], HDFS [1] and XTreemFS [10] use the strategy of I/O and processing co-allocation – i.e. every processing node of a cluster is also a storage server. Application's tasks and data are distributed in a way that, for every task, all input data is available on the local disk and output data can be stored locally and replicated later. With this strategy, performance is greatly improved and there is no need for I/O coordination. On the other hand, this forces one application model of independent task with strong data locality – most prominently, the map-reduce model – that can not be applied to many scientific applications.

Aggregating requests in order to profit from PFS-specific characteristics is studied in several works on the literature. Dickens and Logan [7] approach the performance of MPI-IO on a Lustre environment and shows that collective operations perform poorly because of communication overhead between clients and data servers. Yu et al [22] apply Lustre's file joining feature to improve MPI-IO collective operations' performance. Liao et al [13] propose a client-side caching system for MPI-IO that benefits from collective operations. Differently from these works, our study of intra-node I/O aggregation focuses on the behavior of multiple applications with temporal behavior and its performance upon the presented reconfiguration mechanism.

While our work also applies on-line monitoring and tries to scale the number of I/O servers as more applications are executed, the strategy differs from the ones previously presented on its use of per-application exclusive I/O resources. Additionally, it tries to avoid synchronization costs at the moment of reconfiguration, and uses server overload as a trigger instead of failures or manual intervention. Differently from HDFS or GFS, dNFSp reconfiguration model does not require any specific application model.

6 Conclusions and Future Works

Cluster architectures demand file systems that are able to manage large amounts of applications' data with high throughput during their execution. PFSs that use a static setup on the amount of servers will not be able to present good performance on all the situations to which the system will be submitted, specially when concurrent execution of different applications is allowed.

In this paper we have presented a reconfiguration mechanism for dNFSp that aims to automatically dedicate more I/O resources to applications when contention on the PFS is detected and studied its behavior when used with several independent I/O processes or with a single writer that aggregates the requests before contacting the server.

The results show that the use of multiple writers per node benefits applications with no temporal behavior. It has been shown that this kind of application can cause severe contention for other applications with temporal behavior. The use of intra-node coordination, on the other hand, has shown to increase performance of these temporal applications, at

the expense of the I/O-bound executions'. For the case of small object sizes on the temporal application, this loss is easily compensated by better I/O performance and greater fairness of the whole file system.

As future works we intend to look further into the causes of the unbalance in performance observed in the presence of temporal access patterns and propose mechanisms to avoid the observed contention. We also intend upgrade dNFSp to the NFSv4 protocol, profiting from the *FS_LOCATION* attribute to provide dynamic relocation of data and remove the existing bottleneck on the meta-servers.

References

[1] Apache Foundation. The hadoop file system architecture, January 2009. Available in `http://hadoop.apache.org/common/docs/current/hdfs_design.html`.

[2] R. B. Ávila, P. O. A. Navaux, P. Lombard, A. Lebre, and Y. Denneulin. Performance evaluation of a prototype distributed NFS server. In *Proceedings of the 16th Symposium on Computer Architecture and High-Performance Computing*, pages 100–105, Foz do Iguaçu, Brazil, 2004. IEEE.

[3] A. Batsakis and R. Burns. Cluster delegation: high-performance, fault-tolerant data sharing in nfs. In *HPDC-14: Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing*, pages 100–109, Washington, DC, USA, 2005. IEEE Computer Society.

[4] R. Bolze, F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.

[5] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp. Parallel i/o prefetching using mpi file caching and i/o signatures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[6] P. H. Carns, I. I. I. Walter B. Ligon, R. B. Ross, and R. Thakur. Pvfs: a parallel file system for linux clusters. In *Proceedings of the 4th conference on 4th Annual Linux Showcase and Conference (ALS'00)*, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.

[7] P. Dickens and J. Logan. Towards a high performance implementation of mpi-io on the lustre file system. In *OTM '08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems:*, pages 870–885, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[9] E. Hermann, D. F. Conrad, F. Z. Boito, R. V. Kassick, R. B. Ávila, and P. O. A. Navaux. Utilização de recursos alocados pelo usuário para armazenamento de dados no sistema de arquivos dNFSp. In *Anais do VII Workshop em Sistemas Computacionais de Alto Desempenho (WSCAD 2006)*, Ouro Preto - MG, oct 2006.

[10] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. The xtreemfs architecture—a case for object-based file systems in grids. *Concurr. Comput. : Pract. Exper.*, 20(17):2049–2060, 2008.

[11] F. Isaila, D. Singh, J. Carretero, F. Garcia, G. Szeder, and T. Moschny. Integrating logical and physical file models in the mpi-io implementation for "clusterfile". In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, page 462, Washington, DC, USA, 2006. IEEE Computer Society.

[12] R. Kassick, C. Machado, E. Hermann, R. Ávila, P. Navaux, and Y. Denneulin. Evaluating the performance of the dnfsp file system. In *Proc. of the 5th IEEE International Symposium on Cluster Computing and the Grid, CCGrid*, Cardiff, UK, 2005. IEEE Computer Society.

[13] W. keng Liao, A. Ching, K. Coloma, A. Choudhary, and null Lee Ward. An implementation and evaluation of client-side file caching for mpi-io. *Parallel and Distributed Processing Symposium, International*, 0:49, 2007.

[14] J. M. Kunkel. Towards automatic load balancing of a parallel file system with subfile based migration. Master's thesis, Ruprecht-Karls-Universität Heidelberg –Institut für Informatik, 08 2007.

[15] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12. ACM, 2009.

[16] Los Alamos National Laboratory (LANL). MPI-IO Test 21, 2009. Available in `http://institutes.lanl.gov/data/software/#mpi-io`.

[17] T. M. Madhyastha and D. A. Reed. Intelligent, adaptive file system policy selection. *Frontiers of Massively Parallel Computing, 1996. Proceedings 'Frontiers '96'., Sixth Symposium on the*, pages 172–179, Oct 1996.

[18] S. Microsystems. Lustre file system. White Paper, 2008. Available in `http://www.sun.com/software/products/lustre/docs/lustrefilesystem_wp.pdf`.

[19] S. Microsystems. Pools of targets – lustre file system, 2009. Available in `http://arch.lustre.org/index.php?title=Pools_of_targets`.

[20] Y. L. Ribler, H. Simitci, and D. A. Reed. The autopilot performance-directed adaptive control system. *Future Generation Computer Systems*, 18(1):175–187, 2001.

[21] N. Tran and D. A. Reed. Arima time series modeling and forecasting for adaptive i/o prefetching. In *Proceedings of the 15th international conference on Supercomputing (ICS '01)*, pages 473–485, New York, NY, USA, 2001. ACM.

[22] W. Yu, J. Vetter, R. S. Canon, and S. Jiang. Exploiting lustre file joining for effective collective io. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 267–274, Washington, DC, USA, 2007. IEEE Computer Society.