Dynamic I/O Reconfiguration for an NFS-based Parallel File System

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

***Abstract*—The large gap between the speed in which data can be processed and the performance of I/O devices makes the shared storage infrastructure of a cluster a great bottleneck. *Parallel File Systems* try to smooth such difference by distributing data onto several servers, increasing the available bandwidth of the system.**

However, most implementations use a fixed number of I/O servers, defined during the initialization of the system, and can not add new resources without a complete redistribution of the existing data. With the execution of different applications at the same time, the concurrent access to these resources can agravate the existing bottleneck, making very hard to define an initial number of servers that satisfies the performance requirements of different applications.

In this paper, we present a reconfiguration mechanism for the dNFSp file system that uses online monitoring of application's I/O behavior to detect performance contention on the file system and dedicate more I/O resources to applications with higher demands. This extra resources are taken from the available nodes of the cluster, using their I/O devices as a temporary storage. We show that this strategy is capable of increasing the I/O performance in up to 200% for access patterns with short I/O phases and 47% for longer I/O phases.

I. Introduction

Cluster architectures and parallel programing are nowadays the standard solution to develop high performance applications. Through the aggregation of the processing capacity from several independent nodes, clusters offer enough computing power to solve large and complex systems on areas that range from computer-aided design to climatological analysis and galaxy simulations. This kind of application is usually associated with large datasets used either as input or generated as result of their execution. Due to the volume and characteristics of these datasets, there is need for a *permanent storage facility* shared among all the application's instances on the cluster.

On the other hand, the speed of permanent storage is very slow when compared with that of the computing resources. While processing speed has followed a seemingly constant rate of increase during the last 30 years, the speed of I/O devices like physical disks has no followed such tendency: access speeds of devices available nowadays are just one order of magnitude above the speeds of their 1970's counterparts. This gap in the performance of these two essential components of cluster architecture makes the storage system a very significant bottleneck, causing severe contention for parallel applications.

Parallel File Systems 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. I/O requests from the clients are directed to the servers according to data distribution policies previously defined for the files in order to maximize the use of bandwidth.

The number of data servers used in a parallel file system will influence directly the performance that applications can expect of it. 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 lifetime of the cluster. Overestimating the number of I/O resources to fit any imaginable scenario, though, may result in waste of money that could be, otherwise, used to acquire more processing resources. In this scenario, the initial setup may need to be dynamically adapted to the needs of the running applications.

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 [22], [11], [18], data can be distributed according to application's dominant spacial access patterns [3], [10], the number of servers to use can be adapted to the output characteristics of application [16], 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, there may be the need to increase the number of available storage resources in order to increase the total capacity of the file system.

Changing the number of I/O resources to improve *running application's I/O performance* brings forth several issues that need to be studied.

1) How to minimize the overhead of data replication and reorganization over the new set of servers?
2) Data from which application should be placed in the new resources, in order to improve overall performance?
3) Where should an automated reconfiguration tool search for available disk devices to incorporate in the file

system?

4) When should these devices be released?

To study some of these questions and propose strategies to improve I/O performance we developed an on-line performance monitoring and reconfiguration tool for the dNFSp file system.

The remaining of this paper is divided as follows: Section II introduces dNFSp, our working platform. The reconfiguration model is presented on Section III. On Section IV we present the selection of the application that will use the new servers and the model implementation, whose results are discussed in Section V. Section VI presents related works and state of the art. Finally, Section VII presents the conclusions and future works.

II. 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 of the system without sacrificing compatibility with standard NFS clients – thus simplifying setup and maintenance.

The NFS server functionality is divided onto several metadata servers (named *meta-servers* or *metas*, for simplicity) and data servers (named *IODs*). Each client knows one meta-server as its *nfsd* daemon, so each meta is assigned to only a subset of the clients. When using dNFSp, if all the clients access the file system at the same time, the load will be distributed among these servers [13].

When a request for a given piece of data is received, the meta-server checks for data-distribution information about the file being accessed and forwards the request for the correct IOD. The last manages the actual data on the disks. Once it receives a read or write request, it looks for the data corresponding to the *inode number* of this request and performs the operation. Confirmation messages and read replies are sent directly to the clients. *IP Spoofing* is used to trick the clients into thinking that the answer came from the NFS server associated to them, keeping full compatibility with the protocol.

The synchronization of the metas is done via an LRC [1] based algorithm, in which data is updated in one node only when it needs access to it. The employment of LRC allows nodes to have outdated information, leaving the system partially unsynchronized. Metadata information for each file is assigned to an specific meta-server via a hashing function. The designed server will be responsible for keeping track of the changes done to this specific file and every time other server needs up-to-date information, it will request a copy of the metadata for this one.

dNFSp design requires that each request from the client to be transmitted twice – one from client to meta-server and other from the last to IOD. This is necessary to keep compatibility with standard NFS clients. For read operations, this strategy presents good performance, given the small size of these requests. Since the IOD sends the reply directly to the client, the double transmission overhead is easily compensated by the higher I/O throughput of several independent IODs. This situation is illustrated in the lower half of Figure 1. *Write operations* performance, on the other hand, is impaired by the duplicated transmission of large data blocks. This situation is illustrated on the upper part of Figure 1.

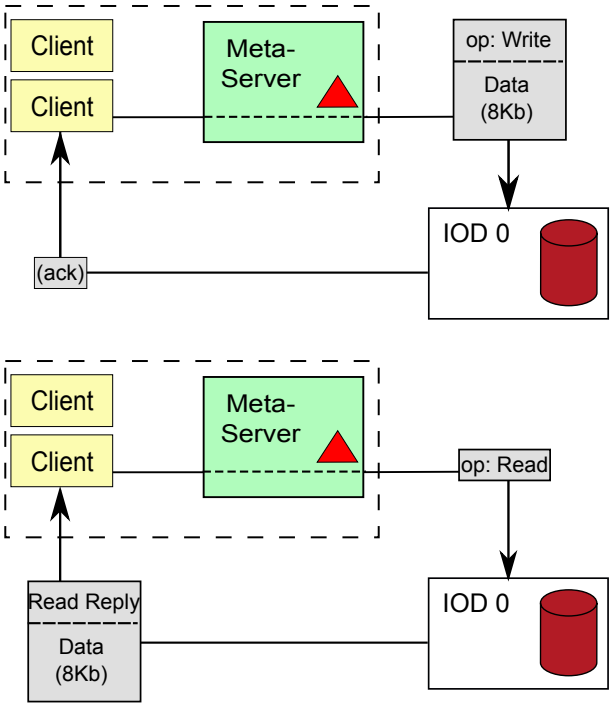

Fig. 1. dNFSp Architecture

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 in the system. For some applications with tight deadlines, though, it may be impossible to complete some computation if the file system does not present the expected performance.

Trying to improve this situation, the concept of *Application IOD* (AppIOD) was introduced in dNFSp in [8]. Application IOD's are a special kind of I/O server associated with a set of clients belonging to an application[2]. Every request from an application previously registered in the file system that arrives on the meta-servers is forwarded to the AppIOD's of that application. Write operations are always processed on the AppIOD; read operations for offsets that are not stored in the exclusive servers are forwarded to the original ones. The application IOD model does not requires the replication of data on the new servers. Instead, it aims to merge data stored on both set of servers when the application to which the servers are associated finishes.

[1] Lazy Release Consistency

[2] An *application*, in the AppIOD model, constitutes a set of IP addresses

The user must instruct the file system as of which clients belong to each application and provide machines to be used as servers – typically, a set of nodes from the cluster. In this model, the user must be aware of the application need for good I/O performance and then make an allocation with extra nodes to be used as exclusive servers. On the other hand, it's not always possible to know if the file system will be able to provide the expected performance. One way to avoid the need of direct user intervention is to allow the file system to decide if the application could profit from new I/O resources and do the proper reconfiguration during the execution.

III. Automatic I/O Reconfiguration

As stated in Section I, 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 increases the existing overhead. In this section, we'll present the reconfiguration model adopted for dNFSp.

To improve I/O performance of applications during their execution, we have defined a model that takes into account two common situations on clusters and parallel file systems: the concurrent execution of applications and the temporal behavior of I/O operations. These aspects have been taken into account to define the moment where the system will trigger reconfiguration.

A. 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 AppIOD's 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 AppIOD's, our reconfiguration model considers 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. We delegate to the scheduler the decision of using these resources as I/O exclusive or if other applications will be able to execute processes on the machines.

B. Triggering the Reconfiguration

In dNFSp, any file big enough will be distributed among all available IOD's. In this scenario, when a single application uses the file system it may obtain the greatest performance

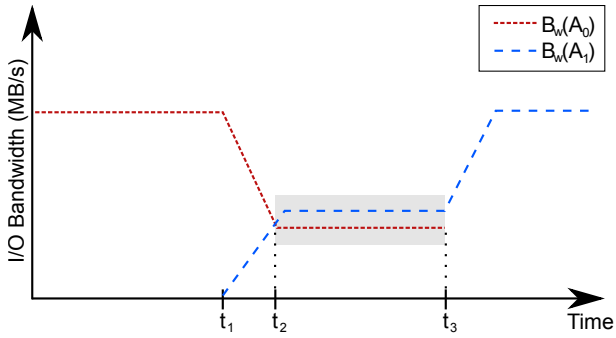

Fig. 2. I/O Bandwidth for two concurrent applications. t_1: A_1 starts. t_2: Bandwidth for A_0 and A_0 is stable. t_3: A_0 ends.

available from the servers. The inclusion of new servers in this case would require merging the data written in the AppIOD's 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 is 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 it's 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.

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

It's 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 that a limit or that do not differ more than a fraction of the best performance observed.

Figure 3 shows the measured bandwidth along the execution of two applications. The application in *blue* presents temporal behavior, while the one in *red* has only a long I/O phase. The period represented by *(a)* shows a *performance-stable period* in which the bandwidth for the red application didn't present much variation. As the blue application enters a phase with few requests, the reconfiguration model detects a new *peak performance*, i.e., the bandwidth that an application can expect of the file system. With the start of a new I/O phase in *(c)*, the application in red has a decrease in it's performance and the system is thus considered *saturated*. 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 not related to application behavior.

IV. Application Selection

Since AppIODs redirects 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 it's 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 length of the idleness phases.

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 it's 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 [22], [5], [21] 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_{AppIDO} is the number of exclusive servers to be used. This formula represents the time slot that would be freed by the application if it's 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.

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

A. Implementation

The proposed reconfiguration model was implemented on dNFSp as a monitoring library – *libPFM* – that collects, for each application, the amount of data transfered. 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 will spawn AppIOD's on available cluster nodes.

To request new resources and spawn the exclusive servers, the PFM Daemon must integrate with a scheduler system. We chose 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 chooses to spawn AppIOD's, it makes an special job submission to VSS, indicating it needs I/O resources for the chosen application. VSS will then select available nodes, start the necessary processes on them and associate the I/O job with the original application job. This way, when application finishes, the file system is notified and can take action to either release the exclusive servers or postpone it 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/.

V. Evaluation and Results

A. Test Setup

We executed our tests on the *Pastel* cluster of Grid5000 [4]. The cluster is composed by 80 Sun Fire X2200 M2 nodes, each with $8GB$ of RAM and two dual core AMD Opteron 2218 processors clocked at *2.6GHz*. The nodes are connected by Gigabit Ethernet network and a Cisco 7006 router.

Each node has a Hitachi HDS7225S SATA disk of *250GB*. The brute bandwidth measured was of $62.69MB/s$[3] for reads and *34.67MB/s*[4] for writes.

In all tests, 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 IOD's used during the reconfiguration was also 4.

To simulate the load of parallel applications acessing the file system, we used the MPI IO Test 21 benchmark [17]. This program uses MPI to execute a parallel application onto several nodes. This application can use either MPI-IO or Posix-I/O primitives to simulate applications access to the file system. In our tests, we used the benchmark with one file per process and standard POSIX calls.

[3] Value obtained by average of 6 executions of *hdparm*
[4] Value obtained with the *dd*, no file system, all caches off

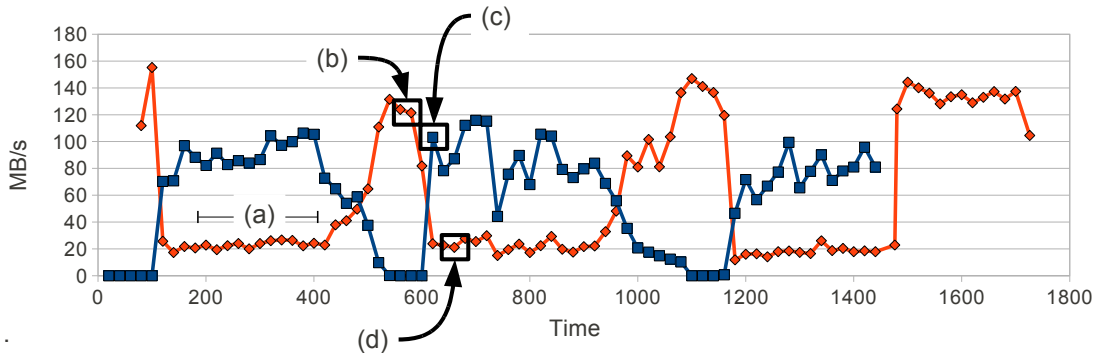

Fig. 3. Bandwidth graph for two applications. *(a)*: Stable period; *(b)* Peak Performance; *(c)* Start of new I/O Phase; *(d)* Saturated performance

Write Size	I=10	20	40	60	80
128MB	0,97	0,94	0,92	0,86	0,82
16MB	0.81	0.69	0.52	0.42	0.36

TABLE I
VALUES FOR $\beta\,(A_0)$ FOR INACTIVITY INTERVALS OF I=10, 20, 40 60 AND 80 SECONDS

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 clear 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. In the I/O phase, each process writes an object of $128MB$ for the first test and $16MB$ for the second one. Table I presents the values of β for each idleness interval and write size tested. These values were obtained by executing each application individually and measuring the ratio between the effective and operation bandwidth. As a consequence, the boundness ratio represents a *behavioral* characteristic of the application, in contrast with a simple atribute measured along the execution. At the end of each I/O phase, we placed a barrier to sincronize the clients.

The second application starts 2 minutes after the first one and executes for 40 minutes. It uses 64 processes over 8 nodes, with each process writing no more than 2GB. This execution used no barriers between I/O phases, thus presenting no coordination between the clients.

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

B. 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 obtained results for this test. Due to the boundness-ratio of A_0, this application is always selected for reconfiguration – dispite having a lower boundness-ratio, it has more clients and thus has priority over A_1.

The first thing we can notice is that for intervals from 10 to 40 seconds performance without reconfiguragion (*NoRec*) for both applications was not significantly different. Despite the intervals with few requests from A_0, the A_1 didn't present an increased performance as would be expected. With longer idleness intervals, though, A_1 presented increase in it's I/O bandwidth at the price of only a slight decrease in A_0's one.

When the automatic reconfiguration mechanism is enabled (*Reconf*), both applications presented increased bandwidth for intervals 10 and 20, presenting an improvement of 47% over the situation without reconfiguration. For intervals 60 and 80, only A_0 presented significant gain in performance, with a 17% increase in the performance of the system. In all cases, the reconfiguration mechanism has been able to detect that performance was sub-optimal and spawn new servers to accomodate the load.

C. Results with 16MB Object Size

Figure 5 presents the results for A_0 with write size of $16MB$. Since in this case the boundness-ratio for A_0 is smaller, with idleness intervals longer than 40 A_1 presents greater *TotalGain* and is selected to receive the exclusive servers. This situation is still equivalent to the previous one, since each application has access to exclusive servers.

Differently from the previous test, we can observe that A_0 presented very poor performance in the case without reconfiguration. This decrease can not be explained by a dominance of A_1's requests on the I/O servers, given that, with longer intervals, it's performance also decreases abruptly, going from 46 to mere $19MB/s$ with intervals of 80 seconds. This behavior had been previously reported in [12] and is due to the temporal behavior of the application.

In this situation, the automatic reconfiguration mechanism has been able to improve the performance of both applications on all intervals. For A_0 we have observed more than 100% improvement for all test cases. For A_1, the improvement was smaller on shorter idleness intervals, but presented performance improvement of 200% with intervals of 80 seconds.

D. Discussion over the Results

The results have shown that the proposed reconfiguration mechanism is able to detect opportunities to improve I/O per-

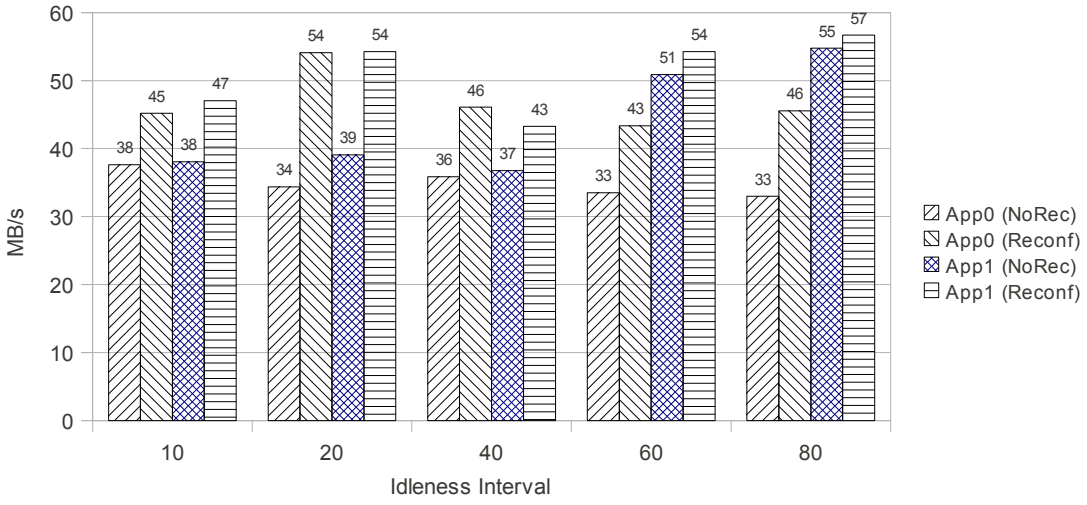

Fig. 4. I/O Performance With (Reconf) and Without (NoRec) Automatic Reconfiguration, 128MB Write Size.

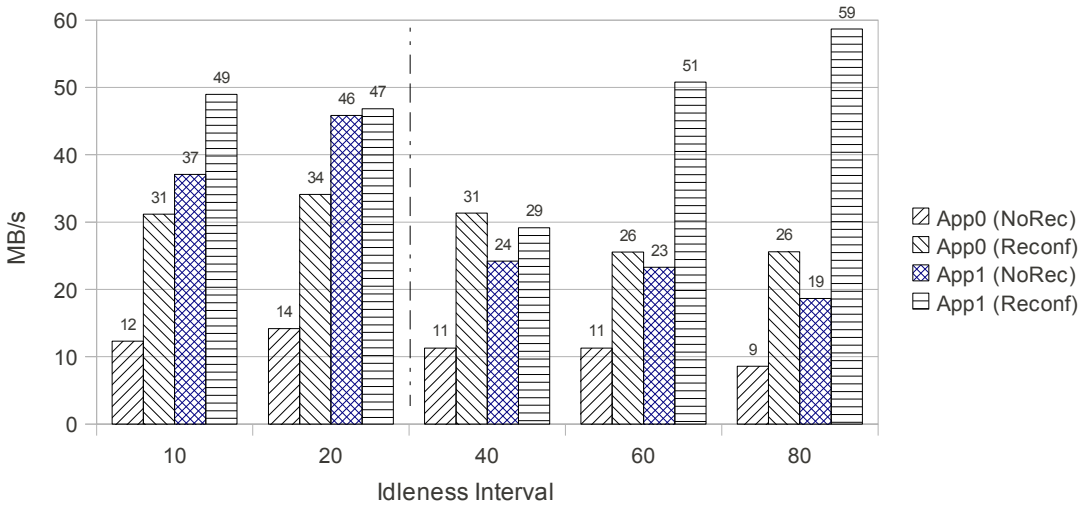

Fig. 5. I/O With (Reconf) and Without (NoRec) Automatic Reconfiguration, 16MB Write Size.

formance of concurrent applications and request new resources to be used as servers.

The aggregated performance presented in the tests was of $104MB/s$ in the best case. The same configuration, without temporal behavior on any application, presented bandwidth of $174MB/s$. As shown in [12], the presence of temporal behavior has shown to decrease performance for all executing applications. Despite this, the reconfiguration mechanism was able to detect the contention and server exclusiveness has shown to provide a much improved performance.

VI. 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 aproaches to provide high efficiency and scalability, both in number of servers and clients. Some of them make use of adaptation to specific characteristics of each application, done either during it's execution or prior to it.

[14] proposes an online monitoring library with data migration capability for the *Parallel Virtual File System* (PVFS) [6]. In this work, server performance is observed during the execution of applications. When load disbalances are detected, the system can decide to move data from one server to another in order 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, contibuting to the load imbalance. Furthermore, if application finishes it's execution (or no longer demands the file whose data was moved) before the migration completes, then the cost of migration may be in vain if the cause for the imbalance

was a temporary one (e.g. a RAID rebuild on the server). The monitoring and detection of imbalance has been extended in [15] to allow for *post-mortem* analisys of the performance. While this can provide important insights into the behavior of the system that can be used by the administrator for fine-tuning, it requires manual intervention from the administrator.

In attempting to provide tera-byte scale storage, the *Ceph File System* [24], [23] considers dynamicity of servers as something that needs to be treated in the core of the storage model. With this in mind, the *CRUSH* algorithm [25] was developed to keep data always available in the presence of failures and keep new resources busy. When new chunks of data are created (either by replication or by writing to a file), CRUSH maps them to a number of available servers. One of the metrics used to choose these servers is an *overload* factor. In this way, CRUSH can prevent over-utilization of any specific server, avoiding unbalance created by different applications. On the other hand, the original mapping is changed only when there is a change in the set of available servers – be it by failure or by manual addition or removal of resources. Server unbalance, by itself, does not activate remapping. In this scenario, concurrent execution of applications still can cause contention that will not be corrected during execution.

Lustre File System [19] is a well known solution for data management in cluster environments, being used in several Top-500 ranking clusters. Lustre storage is *object-based*, i.e. data is divided into fixed size contiguous *objects* and mapped to a given number of *Object Storage Targets* (OST's). Object sizes and the desired number of servers to use is application defined. OST's can be grouped based on common characteristics – e.g. network interconection, I/O device speed, storage space, etc [20]. During file creation, applications can request servers from a given *pool*, using the resources that better fit their need – fault-tolerant I/O, largest storage area, fastest interconection, etc. Lustre, on the other hand, lacks *dynamic* adaptation to application characteristics, since all the parameters must be set up before or during the creation of a file. Changes in the environment that impact I/O performance, like new applications or RAID maintainances, won't be treated by Lustre.

File systems like Google File System [7], Hadoop [1] and XTreemFS [9] use the more scalable strategy of I/O and processing co-alocation. In this case, 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 executed, all the necessary input data is available on the local disk of the node. On the same way, output data can be stored on the local device and be replicated later. Since this strategy avoids the need of communicating with external resources and employs as many I/O resources as processing resources, performance is greatly improved. This strategy, on the other hand, forces one application model of independent task with strong data locality – most proeminently, the map-reduce model. On the other hand, few scientific applications have been shown to fit this application model.

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 it's use of per-application exclusive I/O resources. Additionally, it tries to avoid synchronization costs on the moment of reconfiguration, and uses server overload as a trigger to the need of action instead of failures or manual intervention. Differently from HDFS or GFS, dNFSp reconfiguration model does not require any specific application model, neither expects IOD's to execute applications.

VII. Conclusions and Future Works

Cluster architectures demand file systems that are able to manage large amount of applications data with high throughput during their execution. PFS 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. The proposed mechanism has been shown to work on the presence of temporal behavior of applications, detecting that there was contention on the file system and including more servers to one of the running applications.

The results have shown performance increases ranging from 15% to 47% in the case of $128MB$ write sizes and 126% to 198% with smaller write units. While the total bandwidth obtained with the reconfiguration mechanism was still lower than the case without temporal behavior, the proposed tool was able to correct a situation presenting very poor performance without intervention from either administrator or user.

As future works we intend too look further at the causes of the contention when applications present temporal behavior. 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, Janeiro 2009. Disponível e http://hadoop.apache.org/common/docs/current/hdfs_design.html. Last Access: October 2009.

[2] R. B. Ávila. *Uma Proposta de Distribuição do Servidor de Arquivos em Clusters*. Tese de doutorado, Universidade Federal do Rio Grande do Sul, Brasil, 2005.

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

[8] E. Hermann, D. F. Conrad, F. Z. Boito, R. V. Kassick, R. B. Avila, and P. O. A. Navaux. Utilizao de recursos alocados pelo usurio 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.

[9] 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.

[10] 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.

[11] S. Jiang, F. Petrini, X. Ding, and X. Zhang. A locality-aware cooperative cache management protocol to improve network file system performance. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 42, Washington, DC, USA, 2006. IEEE Computer Society.

[12] R. Kassick, F. Boito, and P. Navaux. Interaction of access patterns on dnfsp file system. In U. d. L. A. Facultad de Ingeniería, editor, *Memórias de Conferencia Latinoamericana de Computación de Alto Rendimiento*, 2009. Cópia disponível em http://eventos.saber.ula.ve/eventos/documentos/clacr2009/pdf_completo.pdf.

[13] 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. Los Alamitos, IEEE Computer Society Press. CD-ROM Proceedings, ISBN 0-7803-9075-X.

[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] J. M. Kunkel and T. Ludwig. Bottleneck detection in parallel file systems with trace-based performance monitoring. *Lecture Notes in Computer Science*, 5168:212–221, 2008.

[16] 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.

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

[18] 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.

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

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

[21] 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.

[22] 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.

[23] S. A. Weil. *CEPH: Reliable, Scalable, and High-Performance Distributed Storage*. PhD thesis, University of California, Santa Cruz, December 2007.

[24] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320, 2006.

[25] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. Crush: controlled, scalable, decentralized placement of replicated data. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 122, New York, NY, USA, 2006. ACM.