A Case for Small File Packing in Parallel Virtual File System (PVFS2)

Faraz Shaikh and Mikhail Chainani

{fshaikh, mchainan }@andrew.cmu.edu

Carnegie Mellon University

Abstract—Parallel file systems are optimized for large file IO access patterns. Most of contemporary parallel file systems like the Parallel Virtual File system [1] (PVFS) and Lusture [2] are designed to deliver high throughput for large file IO. Due to this inherent bias towards optimizing large IO, small file performance on such file systems takes an adverse hit. This paper explores the possibilities of improving small file IO performance on parallel file systems without heavily degrading the existing performance of large IO. We also provide a quantitative measure of how large file IO performance is affected when our small file IO optimizations are introduced in the file system.

I. Introduction

PARALLEL file systems provide high IO bandwidth to computing nodes by connecting them in parallel to multiple IO resources via multiple IO paths. Application data is thus distributed over multiple IO resources and is then stored / retrieved in parallel. For a single IO operation a parallel file system may need to initiate flow setup between multiple IO resources. The flow setup process depends on type of network fabric used to connect the compute nodes and the data nodes and entails some finite processing time. Workloads characterized by accessing Large IO access patterns amortize this initial flow setup overhead with the performance gained by accessing part of the data in parallel. However for small IO access patterns the initial flow setup presents an unwarranted overhead.

We propose 3 optimizations for small file IO access patterns on Parallel file systems. We decided to choose PVFS2 for implementing and evaluating the following optimizations.

a. *Small File Packing*:

In this optimization the initial N bytes of file data are stored on the metadata server rather than on IO servers. The exact value of N should be chosen to be greater than the size of most small files in a workload. This would means that most of the small files under size N are entirely stored on the Metadata server.

b. *Eager IO for small files:*

For every IO operation, the client issues an initial get attributes request to the metadata server for the purpose of permissions checking or getting the layout of the file data. Eager IO is a hint to the metadata server to piggyback the initial N bytes of file data along with the response to attribute request. Upon receipt of this response clients complete the user IO request entirely from the piggybacked file data and do not contact the IO servers. This saves at least one network access for IO's involving small files.

c. Delayed data handle creation for files:

Creation of file in PVFS2 involves allocating data handles for the new file on IO servers. These data handles are fixed in number depending upon the maximum stripe count. It makes sense to pre-allocate them because of their fixed nature. For small files that never span across multiple servers this handle pre-allocation proves to be an overhead.

Eager IO and small file packing thus are complementary to each other and provide benefits similar to "Inode Stuffing" on a parallel file system. With the above optimizations in place the Metadata server takes an additional role of an IO Server for accesses to head of the file. Thus, we also study the worst case impact of our optimization on bandwidth perceived by application issuing large file IO.

II. Motivation

Parallel file systems are usually employed in scenarios where the storage system limits the performance and scalability of an application. A good use case for parallel file system would be long running scientific applications that require frequent state check pointing for recoverability in cases of failure. Higher frequency of check pointing enhances the fault tolerance of these long running applications. Performing checkpoints at higher rates also means incurring negative performance impacts due to the slow nature of the IO subsystem. In this case it makes sense to employ the parallel usage of IO resources to achieve high aggregate IO throughput. Also checkpoint is characterized by dumping large chunks of data onto permanent storage. Thus, the scenario explained above is a good combination of existing workload requirements (i.e. large chunks of data to write) and available technology (parallel file IO). In such scenarios implementing any sort of optimization for small file IO access will be futile due to the complete lack of small IO accesses in such workload. Scientific application workloads have been studied in quite detail [3] under the CHARISMA project. This study provides great insights into IO access patterns of scientific applications but unfortunately it does not shed light on the distribution of file sizes in such workloads.

On the other extreme are applications that demand high transaction rates in terms of creates/deletes and other metadata operations. Good examples of these types of applications are news transactions, email servers and e-commerce. Here again the solutions are completely biased towards improving small file IO due to its dominance in the workload. Optimizing large file IO access in these situations is again futile.

There exists a complete range of applications where computing performance is harnessed by doing distributed computations. Usually, Parallel/Distributed IO techniques are employed in these cases to club input data and computing resources close to each other. A good example of this is the distributed computing clusters employed at Google that use the distributed Google file system (GFS) [4], MapReduce [5] and the Workqueue framework for intelligent distribution of computing task. The distributed nature of these applications makes them susceptible to higher fault rates and this mandates persistence storage for intermediate or final computed results. This is especially true when re-computing results may incur a significant overhead. One such distributed computing framework in explained in detail by Rob Pike et.al in [6]. Such workloads definitely ask for storage solutions that perform optimally for both small and large file IO access patterns. It can be contested that intermediate results can be stored in core by such applications, but this does not provide for persistence of results required for fault tolerance.

Our optimizations will improve the performance of applications that use parallel file systems and exhibit a well balanced workload consisting of accesses to both large and small files. This is close to the case explained in distributed computed environment example explained above. In general, our optimizations will benefit applications that operate on large chunks of data as input and generate substantial amount of small files as intermediate or final results.

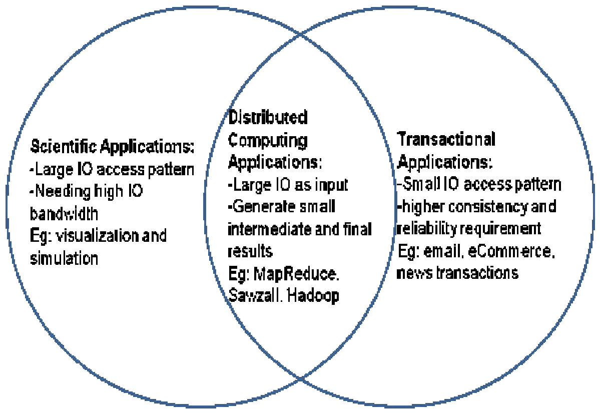

Scientific Applications:
-Large IO access pattern
-Needing high IO bandwidth
Eg: visualization and simulation
Distributed Computing Applications:
-Large IO as input
-Generate small intermediate and final results
Eg: MapReduce, Sawzall, Hadoop
Transactional Applications:
-Small IO access pattern
-higher consistency and reliability requirement
Eg: email, eCommerce, news transactions

Fig 1. Classes of applications and IO characteristics.

III. Related Work

Existing solutions that address the problem of providing optimal data access to both large and small files fall into two categories.

1) *Providing auxiliary optimization for the non-dominant access pattern.*

Currently PVFS2 optimizes small IO requests by eliminating some part of the initial rendezvous between IO server and the client which is required for completing the IO. Small IO requests are piggy backed along with initial setup request/response messages exchanged between the client and the server.

Such optimizations provide the benefit of being completely hidden from the end user. This means that the user need not explicitly route IO through different IO paths depending upon the access characteristics. Our optimization falls into this category of solutions.

2) *Combining 2 complimentary file systems that collectively provide an optimized storage abstraction for different IO access patterns.*

CITI developed a solution for improving small IO on their parallel file systems in [7]. Their implementation combined the advantages of PVFS2 and pNFS. Under their approach allow large IO follows the regular path to a parallel access file system while small IO is routed through a NFS server. pNFS then does a deferred write for this small IO through an PVFS2 like abstraction.

These solutions provides for good performance adaptation based on dynamic IO access patterns. These systems however require intelligent agents to route IO to the optimized storage interface. These Intelligent systems include policy agents that work in conjunction with some Hierarchical Storage Management file system or even system administrators. The effectiveness of such solutions is directly related to implementation of the intelligent agent. Also, combining more than 1 file system for storage creates additional problems due to differences between the consistency semantics and storage formats of the collaborating file systems. For example if a co-operating file system does not provide facilities for memory mapping file data then file data has to be moved between file systems.

Devulapalli A. et al [8] have described multiple options for optimizing file creates in a distributed metadata file system using PVFS2 as their test bed. The strategies suggested by Devulapalli mostly concentrate on parallelizing individual operations done during a file create. Devulapalli describes methods to combine creation of metafile and setting attributes of the metafile in a single operation. But the author does not focus on file creation optimization with specifically small files in mind. We suggest additional optimizations for file creation with 'Inode stuffing' by delaying IO handle allocation until a file grows to size greater than the small file threshold. The IO handle allocation step is completely skipped if the file size remains under the small file threshold for the system.

"Inode stuffing" is a well known method for optimizing access to small files on local file systems. In this method the initial bytes of a file are stuffed / stored in the inode blocks. This is a big win for small files in terms of performance because metadata and data can be transferred in a single disk access. Other techniques for optimizing disk bandwidth for accessing small files on local disk based file systems are suggested by G. Ganger in [9]. Ganger proposes two methods for optimizing disk bandwidth for small files namely "embedded inodes" and "explicit grouping". Embedding inodes refers to the method where child inode pointers within a parent directory are replaced by the inodes themselves. One can see that "embedded inodes" definitely perform better in scenarios where a user wants to list the children in a directory followed by a stat on each of them (namely the ls –ltr command). 'Explicit grouping' tries to group on disk the data for small files falling under the same directory. Well all of the three techniques used for small file optimization for local file systems can be applied in the context of distributed file systems. Our implementation though tries to mimic the "Inode Stuffing" optimization for small files on a local file system in the context of distributed parallel file system. The "embedded inodes" and "explicit grouping" optimizations do not lend themselves directly to be used in a parallel file system.

IV. Design

This section describes the design and implementation of small file optimizations on the PVFS2. Namely, it discusses small file packing at MDS and eager IO for small file reads.

PVFS2 made a good candidate for our testbed because it was open source in nature and is highly customizable. Also, PVFS2 focuses on optimizing large file IO.

PVFS2 clusters have 3 types of nodes.

- **Metadata Server (MDS)**:
  These nodes are responsible for storing the metadata about an single file system volume as well as metadata for individual files. Multiple Meta data servers can be employed in a single PVFS2 cluster to provide for scalability

- **IO Servers:**
  These are the nodes that store the actual data for the files. IO servers can be accessed in parallel by clients while performing file operations.

- **Clients:** They request for file metadata from MDS and data from IO servers using either using the VFS layer or using the user space libpvfs native API

In order to maintain some consistency, PVFS2 does not aggressively cache attributes at the client side. The implications of this design are as follows:

The basic operations for file IO involves accessing the MDS to retrieve the files common attributes and its distribution across IO servers and then accessing the IO server(s) for data. In case of small files, this results in at least two network round trips. Also, while creating small files, we need to allocate data handles on all the IO servers, even though we may never use all of them.

Fig 2 and 3. Show the sequence diagrams that illustrate the IO and create operations in PVFS2.

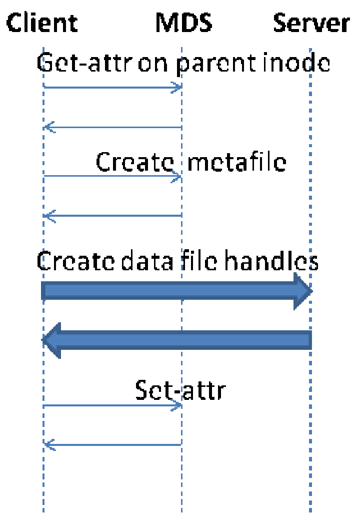


Fig 2: Create Sequence Diagram

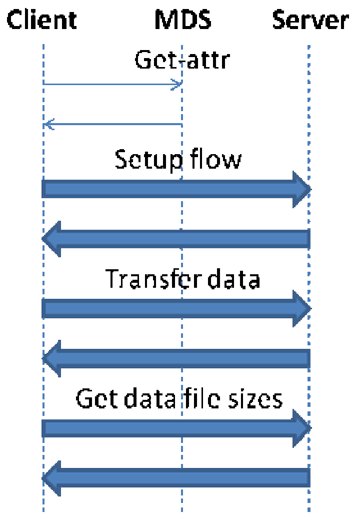


Fig 3: IO Sequence Diagram

The design proposes replication of first N bytes of all files at the MDS either as an extended attribute in the database or as a file, thus serving as a means to perform eager IO by piggybacking the small file data along with metadata. This value of N is referred to as the threshold. Specifically, this is done when the client requests for attributes from the MDS. For this purpose, we introduce a new state machine that will obtain the data from the MDS together with metadata for small files, and cache this data at the client. Subsequent requests for data may be serviced from this cache. We also look at any difference in performance between using the database VS using the local file system at the MDS.

In addition, we optimize creation for small files by deferring the allocation of data handles at the IO servers. Handle allocation will be done when the file exceeds the threshold (n).

Shown below are the sequence diagrams depicting these modifications to the base PVFS implementation.

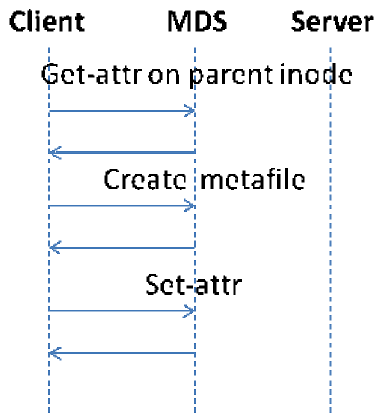


Fig 4: Create Sequence Diagram.

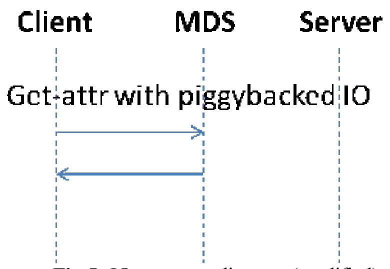


Fig 5: IO sequence diagram (modified)

Some issues with replication of n bytes at the MDS are that the MDS now needs to service client request for both metadata as well as the initial n bytes of file data. This may cause the MDS to form a bottleneck and cause problems with scaling the number of clients as well as files that can be managed by that MDS. One solution is to use multiple MDSs.

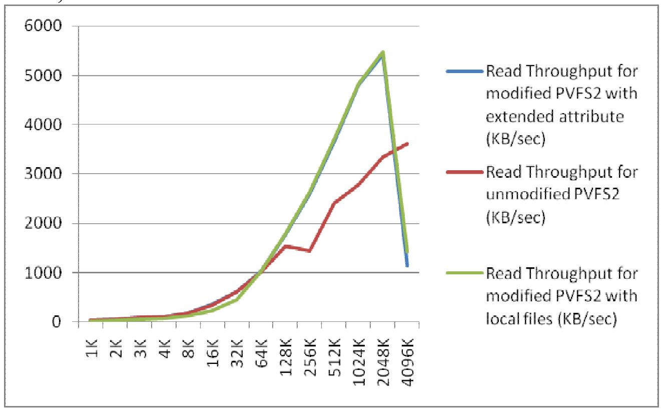
Replicating all but small files may be avoided by carrying out migration of the data when the file grows. We do not implement this, but present a number of considerations while implementing such a mechanism. We will need to consider how often to perform migration. We may choose to do it immediately, or defer it to when the system is less loaded, or perform the operation periodically as a background task. Migration may need to be done in both directions: from MDS to IO servers, when file grows and from IO servers to MDS when the file becomes smaller than n bytes. Also, whose responsibility is it to perform the operation: this may be done by the client writing to the file causing it to cross the threshold, or by a server thread. Migration will also trigger extent creation or deletion at the IO servers.

V. Evaluation

The setup used for the experiments consist of Emulab nodes. Each node has a PIII processor with 512MB RAM. They run Fedora Core 3 images and are equipped with Berkeley DB for the MDS attribute store. Each test was run in a PVFS2 cluster containing 1 MDS, 3 IO Servers and multiple clients and runs over a 100Mbps network with a 5ms delay.

We use Postmark [10] to obtain the read throughput for clients accessing small files on both the implementations of the system: using the database to store the small data at MDS and using the local file system to store the small data in files. We increase the threshold (N) of small files and measure the read throughput seen by a client that has a workload that is primarily small file read intensive. The same test was run on unmodified version of PVFS2, to obtain a comparison between them. The results of the experiment are illustrated in the graph below.

The vertical axis plots the read throughput measured in KB/sec, while the horizontal axis is the value of threshold N.

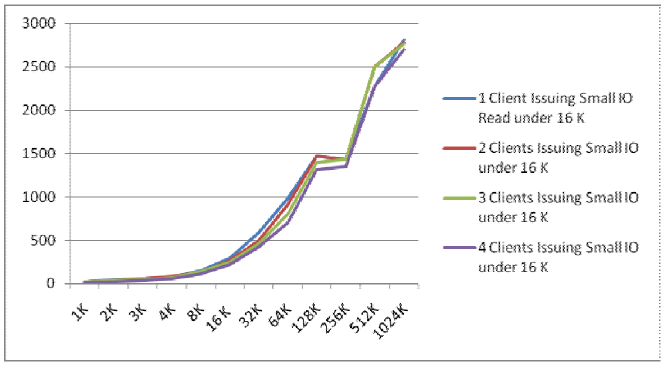


Graph 1: Read throughput vs threshold for original PVFS2, modified PVFS2 with data in extended attribute, and modified PVFS2 with data in local file system at MDS.

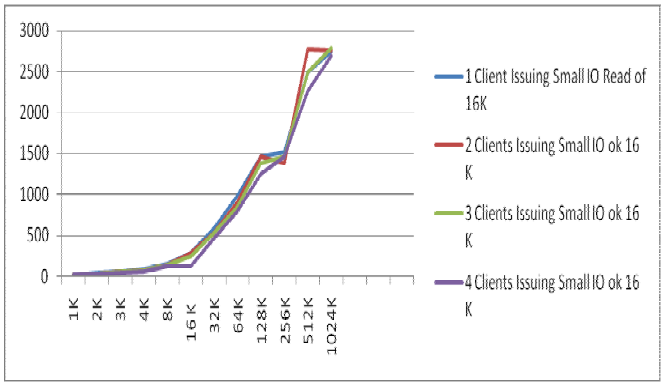
The graph shows that the throupут is almost the same in both setups for values of threshold less than 32KB. As we increase the threshold beyond 32KB, we see the read throughput with small file optimizations using both the database and file system as backends to store the small data, exceeds that of original PVFS2. Finally as the files grow larger, the MDS is more loaded, thus reducing read throughput for the small file optimized cases.

Also, we see that the graphs corresponding to the setup using small file data as an extended attribute in the database is almost alined with that in which the small file data is stored as local files.

The Metadata server now also acts the data server for the initial N bytes of small files. This additional load on the metadata server may constraint or hampers its performance in serving metadata requests. We try to gain some empirical measurements on how conflicting client access patterns affect each other. In our evaluation we are particularly interested in measuring the ill effects of small file IO optimization on existing optimization for accessing large file IO. We try to measure the throughput degradation seen by regular client in the presence of N clients that utilizing the additional services of the MDS in order to access small file. We conduct our measurements in 2 cluster settings. In the first cluster all nodes are running the unmodified copy of PVFS2 file system. We then measure the Read throughput perceived by an application in presence of N applications issuing small IO. We term small IO as accesses that are less than 16K and we iterate over a the range of 16K to 64K worth of small file access using 1 to 4 small IO clients. We then repeat the same experiment by providing "small file packing" and "eager IO" as optimization for client issuing and small IO.



Graph 2: Effects of interfering clients in base PVFS2



Graph 3: Effect of small IO on large IO throughput in Base PVFS2

The results of the two clusters are show in Graph 2 and 3. Graph 2 describes the IO throughput seen by and normal client in presence of 1-4 unmodified clients performing simultaneous small IO. In essence, results described in Graph2 form the baseline for evaluating the effect of our small file optimization on clients that are not gaining from small file IO.

Graph 3 shows the perceived IO throughput seen by a client in presence of N reader nodes issuing IO that is completely satisfied by the Metadata server. The clients used in this setup are running the modified version of the PVFS2 file systems with small file optimizations in place.

Interestingly the perceived read throughput for small reads in the range of 1K to 32K doesn't show much change in presence additional overheads at the MDS due to serving small file IO. This may be because we are using only 4 small IO clients. In the range of 32K to 64K we see large amount of variations in perceived throughput as small IO clients are added and removed from the experiment. This is explainable because the throughput seen by clients performing IO in the range of 32K to 64K do not amortize over parallel access because the stripe size in our setup is 64KB.

Well the perceived changes in IO bandwidth greater than 32K is somewhat erratic. We consistently got data points where application gained bandwidth even in presence of more small IO clients issuing small IO. We can safely conclude that our implementation of small file IO did not contribute to this erratic behavior of data points about 32K because similar points exists in the Graph 2 which was not running our system.

VI. ConclusionS

We would like to have come up with formulae to calculate the right threshold for small files given a set of cluster parameters. Based on our finding we can claim the following

1. Small file threshold value between 32K and 2M provide maximum throughput for small files being served off the MDS.
2. Any thing between 2M and 4M performs badly as a threshold value for small file.
3. For threshold values between 1K to 32K almost perform similar to unmodified PVFS. We believe that the small IO optimization already in place in PVFS2 takes care of this case.
4. We didn't reach any conclusions on regarding the tradeoff of using a file systems or a database to back the data of small files at the MDS.

Acknowledgments

We would like to express our gratitude to Prof. Dave Andersen, our advisor for this project and James Hendricks, who mentored us throughout the project. We would also like to thank the contributors of the Emulab network at University of UTAH for providing the network testbed used in for the evaluations of our implementation.

References

[1] PVFS: A Parallel File System For Linux Clusters (pvfs1.ps) P. H. Carns, R. B. W. B. Ligon III, and R. Thakur, Proceedings of the 4th Annual Linux Showcase and Conference, 2002.
[2] Cluster File System, Inc. Lustre: A Scalable, High Performance File System. http://www.Lustre.org/docs.html.
[3] File-Access Characteristics of Parallel Scientific Workloads http://ieeexplore.ieee.org/iel4/71/11674/00539739.pdf
[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, The Google File System, Proc., 19th Symposium on Operating System Principles, Lake George, New York, 2003, pp. 29-43.
[5] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Proc 6th Symposium on Operating Systems Design and Implementation, San Francisco, 2004, pages 137-149.
[6] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. Scientific Programming, 14, Sept. 2006. Special Issue: Dynamic Grids and Worldwide Computing.
[7] Gregory R. Ganger, M. Frans Kaashoek: Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files. USENIX Annual Technical Conference 1997: 1-17
[8] pNFS/PVFS2 over InfiniBand: Early Experiences, Petascale Data Storage Workshop, held in conjunction with Supercomputing (SC), Nov. 2007.
[9] pNFS/PVFS2 Documentation http://www.citi.umich.edu/projects/asci/pnfs/docs/pnfspvfs2.html
[10] PostMark: A New File System Benchmark, Jeffrey Katcher, www.netapp.com/tech_library/3022.html