Chapter 1

Filesystem Partitioning via Hierarchical Cluster Analysis

Contributed by Tim Shaffer

1.1 Introduction

As research computation achieves larger scales and takes greater advantage of hardware accelerators such as FPGAs, GPUs, and TPUs, the availability and performance of the storage system becomes a bottleneck in the overall performance of an application. Checkpointing, scratch storage, and data distribution all place storage in the critical path of the application, forcing the storage system to keep up. Parallel filesystems offer a way of delivering large file I/O bandwidth, scaling with storage hardware by striping blocks, files, and volumes across different devices, and maintaining consistent metadata and organization.

Metadata performance becomes a key bottleneck in most storage systems at extreme scale. Strategies such as caching, striping, and larger transactions allow for improvements in file data access. Unfortunately, serving the system's metadata poses distinct challenges that preclude approaches used for data access. Metadata elements are small (e.g. user-visible inode information), require stronger consistency than data, and are accessed through small update transactions rather than large linear reads and writes. It also becomes difficult to maintain a consistent view of directory structures among a large number of nodes with (potentially) an extremely large number of directory entries.

Metadata activity is often extremely unbalanced, as production high performance computing (HPC) applications generate irregular bursts of metadata access. Application startup is particularly problematic for the storage system. While researchers often treat HPC applications as "simple" programs to be launched, the steps required are almost always more complicated. What appears to be a single "application" may actually be a complex collection of interpreted programs, dynamic libraries, configuration files, and calibration data. Loading the complete structure of the application into cluster memory at startup results in tens of thousands of interactions with the filesystem at each node. When thousands of nodes of the cluster attempt to load the same application at the same time, the filesystem must handle thousands of small transactions from each node before any one can make progress.

The problem of metadata handling is in the general case extremely difficult to solve. The consistency semantics of general-purpose filesystems severely limit potential optimizations. The different ways a shared filesystem is used each require somewhat different semantics. Data passed between concurrent processes, for example, requires strong consistency at each filesystem interaction. Between sequential processes, filesystem data requires strong consistency eventually. Static data such as software will not change during the execution of a given task does not require additional consistency guarantees.

For scientific applications at scale, metadata behavior often becomes the limiting factor for performance. Scientific software is likely to use the shared filesystem in all of the previous ways. A single application may store intermediate files, synchronize between steps of an analysis, distribute application software, and collect results from multiple worker nodes. A general-purpose filesystem must adequately support the strain imposed by each. Efficient handling of common cases such as software distribution can be the key factor in attempting to scale up an analytic workflow. Poor choice of filesystem use or missed optimizations will make shared computing resources unusable for a given researcher, or in some cases for all users of a site.

Widely deployed shared filesystems such as Panasas [10], Lustre [1], Ceph [9], Gluster [5], and HDFS [7] rely on separate data and metadata servers to make a filesystem tree and its data visible from anywhere in the system. Access patterns for data permit optimizations to servers that simplify operation and improve throughput and parallel access. Large reads and write of file data are especially suited to bulk access operations. Unlike the generally simple and flat structure of data stores, metadata servers must maintain consistent views of hierarchical files and directories. In order for a node to to read a file's data, path resolution is the responsibility of metadata servers. These lookups and permission checks eventually determine the location where the actual data are stored. Efficient path resolution and metadata lookup is thus critical for the performance of a shared filesystem.

While data servers can easily increase replication and shift load away from overloaded servers, metadata servers must maintain stronger consistency guarantees. It is common to use filesystem partitioning to balance the load of requests Choice of partitioning scheme depends on user activity and filesystem organization. While it is possible in some cases to shift excessive loads between metadata servers, the requirement for consistent semantics of operations makes it difficult to distribute a single part of the filesystem tree across multiple metadata servers. When load on a particular metadata server becomes excessive, all requests (including some data requests) cannot be served efficiently. Users then experience degraded performance or loss of service despite. In this situation, accesses to a single part of the filesystem transalte to load on a single metadata server responsible for a partition. Thus the overall system's performance is degraded while leaving data storage nodes underutilized [11]. For parallel filesyetems at increasing scales, metadata bottlenecks become the limiting factor for performance and usability. These pathological metadata access patterns motivate many of the design choices of modern parallel filesystems like Ceph.

1.2 The Problem as a Graph

Applications at HPC centers are composed of (a possibly large number of) individual processes. Running a process requires, at minimum, path search and library loading. It is also common for processes to search for and read in input, configuration, or calibration data. Some applications use the shared filesystem as a means to synchronize between components. To allow the application to recover from failures, checkpoints can be written out as well. Finally, there is usually some output data flushed to the filesystem. The latter three uses rely on strong filesystem semantics and consistency guarantees, limiting the optimizations available. The former uses, however, are good candidates for further examination.

By identifying parts of the filesystem that are used together, for example all of the libraries a single program loads, it becomes to possible to use more target optimizations such as pre-staging a frequently used subset of static metadata entries on nodes. Unfortunately, determining which parts of a filesystem are relevant to a particular application is difficult. There is generally a set sequence of operations for path search, library loading, etc. The exact operations can be non-deterministic or data dependent, however. Furthermore, each individual process of a complete application has distinct behavior, though some patterns are likely to be repeated across processes. There is thus no way to determine a priori which parts of the filesystem an application relies on. It is also not sufficient to try to identify one or a few "program directories" containing all needed pieces. Research applications can (and very often do) use creative filesystem organizations that are not amenable to automatic dependency tracking.

Lacking an a priori method to determine the filesystem dependencies of an arbitrary research application, it is instead necessary to observe filesystem behavior over multiple application runs. It is possible to trace the behavior of individual processes using tools like `strace` to capture syscalls. Since there is a performance penalty in tracing, it is generally better to trace some fraction of processes.

Using these process-level traces, the next step is interpreting the sequences of accesses and choosing groups of related filesystem entries. It is not sufficient to simply collect every filesystem entry that was accessed. This approach collects broad trees of filesystem entries that in practice include substantial amounts of irrelevant data. Instead, a better approach would take into account the fine-grained behavior of the processes. For example, a directory that is listed once during library search and never accessed again is a poor candidate for optimization. On the other hand, a set of libraries that are accessed consecutively by every process should be grouped together and pre-staged on nodes to reduce traffic to the shared filesystem.

To capture both the frequencies and orderings of filesystem accesses, we can build a directed graph based on the syscall traces of each process. In this representation, filesystem entries are the vertices of the graph. The events comprising the syscall traces are used to derive edge weights, with large numbers of accesses resulting in high edge weights. For a process that most recently accessed filesystem entry A and next accesses B, we increase the edge weight of $A \to B$ by one, creating the edge if it does not exist. This representation includes far fewer vertices and edges than events in the syscall trace. In addition, it is amenable to streaming updates as new traces become available or the input data and configuration change.

1.3 Some Realistic Data Sets

The `strace` utility is widely available on Linux based systems, making it possible for researchers to collect syscall traces from their applications. Aside from the previously mentioned performance overhead, there is little barrier to profiling as the process does not require changes to the application. It might be possible to refer to a graph database or to generate synthetic graphs, but simply profiling the actual application will be more effective. As an example application that researchers actively use in an HPC context, we collected syscall traces of MAKER [2], a complex bioinformatics application. Aside from the application itself and its input data, MAKER depends directly or indirectly on an additional 40 languages and libraries. Each phase of a MAKER analysis uses some subset of these dependencies and inputs. Over the course of an analysis, MAKER spawns a large number of individual processes, each of which goes through library loading, input data selection, etc. In our test run on a small dataset, MAKER performed 3.8 million I/O operations. Of these, 1.1 million were metadata operations. This run included bursts of up to 10,000 metadata I/O operations per second. The total running time was 13.7 seconds. Note that it is not uncommon for HPC applications to run for hours or even days. A longer-running application would produce a corresponding increase in the number of I/O operations. For the following sections, we will explore additional applications and datasets.

Rather than representing each metadata operations directly, the graph representation of the execution trace aggregates these events into edge weights. Each vertex in the graph corresponds to a filesystem entry accessed during the run. For this small run, the resulting graph included roughly 20,000 vertices. An analysis on a larger data set or using an application with more dependencies would result in a larger number of vertices. The graph for this particular run included roughly 60,000 edges. Again, the number of edges and ratio of vertices to edges is strongly dependent on both the application and the dataset. Since the programs spawned during a MAKER analysis exhibit both non-deterministic and data dependent behavior, there can also be limited variation in the properties of the graphs of otherwise identical runs. Streaming events from multiple analysis runs into the same graph will also result in changes to the graph properties, though it is hard to predict behavior in the general case over all applications. As general trends, merging multiple distinct applications into the same graph should result in a largely disconnected cluster for each application's activity. Combining analyses of different data sets in the same application should result in an increased graph size with some core vertices common to all runs. Finally, merging events from analysis of the same data in the same application should not significantly affect the properties of the graph.

1.4 CLUSTERING-A Key Graph Kernel

To detect clusters of related filesystem entries, we propose applying hierarchical cluster analysis to the execution trace graph of an application. Clusters in the graph serve as reasonable groupings of filesystem entries for partitioning or pre-staging at worker nodes. In addition, the dendrogram resulting from hierarchical cluster analysis allows some flexibility in the granularity of clusters. When applied to real-world applications, a completely automated approach is difficult in the general case. Providing a choice in cluster granularity allows the user to apply domain knowledge to make the best decision while still providing some automated assistance.

To illustrate hierarchical cluster analysis, we use the Girvan–Newman algorithm [3], a well-studied method. Girvan–Newman repeatedly removes edges from the graph, with the remaining connected components as the clusters. The edge to be removed each step is chosen based on **edge betweenness**, a centrality measure of the number of shortest paths in the graph passing along a given edge. The underlying assumption is that by iteratively removing non-central edges, you can gradually cut apart the clusters. The algorithm proceeds until all edges have been removed. Thus the user can choose precise cluster granularities from individual vertices up to the entire graph. The algorithm itself (taken from [3]) is given in Figure 1.

Algorithm 1 The Girvan–Newman Algorithm

```
Require: Graph $G = (V, E)$.
  Calculate betweenness $g(e)$ for each edge $e \in E$.
  while $|E| > 0$ do
      Remove the edge $e$ with the highest betweenness $g(e)$.
      Recalculate betweennesses for all edges affected by the removal.
      Identify clusters as connected components in $G$.
```

The betweenness can be calculated using [4] in $O(mn)$ time, where m is the number of edges and

n is the number of vertices in G. Since betweenness is calculated at the removal of each edge, the algorithm runs in $O(m^2n)$ time. Determining connected components can be performed in $O(m+n)$ time. Since only changes to betweenness need to be calculated after the first time, this repeated computation can be confined to a single connected component rather than the whole graph. As the clusters become smaller and smaller, in practice this optimization significantly reduces the computation.

1.5 Prior and Related Work

Several approaches to achieving sufficient metadata performance in shared filesystems have been explored. One approach is to separate metadata from the parallel filesystem completely. This approach maps metadata storage tables to file objects in the parallel filesystem [12, 6]. The total metadata transaction rate of the system is improved, but each client must still make many small transactions while using the service. Another approach is to introduce new operations that access metadata in bulk or with weaker consistency guarantees. Examples of this include the proposed `getlongdir` and `statlite` system calls [8], which are, unfortunately, not widely implemented. This reduction in the transaction rate between clients and servers complements other approaches.

Bibliography

[1] R. Behrends, L. K. Dillon, S. D. Fleming, and R. E. K. Stirewalt. White paper: Lustre file system high-performance storage architecture and scalable cluster file system. Technical report, Sun Microsystems, Menlo Park, California, December 2007.

[2] M. S. Campbell, C. Holt, B. Moore, and M. Yandell. Genome Annotation and Curation Using MAKER and MAKER-P. *Curr Protoc Bioinformatics*, 48:1–39, Dec 2014.

[3] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[4] M. E. J. Newman. Scientific collaboration networks. i. network construction and fundamental results. *Phys. Rev. E*, 64:016131, Jun 2001.

[5] Inc. Red Hat. Gluster. http://www.gluster.org/, 2017. Accessed 2018-09-28.

[6] K. Ren, Q. Zheng, S. Patil, and G. Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248, Nov 2014.

[7] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[8] Murali Vilayannur, Samuel Lang, Robert Ross, Ruth Klundt, Lee Ward, et al. Extending the posix i/o interface: A parallel file system perspective. *Argonne National Laboratory, Tech. Rep. ANL/MCS-TM-302*, 2008.

[9] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[10] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.

[11] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. Characterizing output bottlenecks in a supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 8:1–8:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[12] Q. Zheng, K. Ren, and G. Gibson. Batchfs: Scaling the file system control plane with client-funded metadata servers. In *2014 9th Parallel Data Storage Workshop*, pages 1–6, Nov 2014.