To appear in Proc. of the 2007 Int'l Conf. on Parallel Processing (ICPP-07).

Multi-Layer Event Trace Analysis for Parallel I/O Performance Tuning*

Pin Lu and Kai Shen
Department of Computer Science, University of Rochester

Abstract

The complexity of parallel I/O systems lies in the deep I/O stack with many software layers and concurrent I/O request handling at multiple layers. This paper explores multi-layer event tracing and analysis to pinpoint the system layers responsible for performance problems. Our approach follows two principles: 1) collect generic (layer-independent) events and I/O characteristics to ease the analysis on cross-layer I/O characteristics evolution; 2) perform bottom-up trace analysis to take advantage of the relatively easy anomaly identification at lower system layers. Our empirical case study discovered root causes for several anomalous performance behaviors of MPI-IO applications running on a parallel file system. First, we detect an anomaly with the asynchronous I/O implementation in the GNU C runtime library. Additionally, we find that concurrent I/O from multiple MPI processes may induce frequent disk seek/rotation and thus degrade the I/O efficiency. We also point out that lack of asynchronous support at the parallel file system client side may result in inefficiency for fine-grained writes. Using an aggressive I/O prefetching strategy and a corrected asynchronous I/O implementation, we achieve 39–156% read I/O throughput improvement for four out of five applications that we experimented.

1 Introduction

The system architecture for data-intensive high-end applications must scale to meet the rapidly growing demands on the performance and I/O capacity. One approach is to support high-performance parallel I/O using clusters of commodity servers, storage devices, and communication networks. In this architecture, storage resources are directly attached to dedicated or general-purpose compute nodes. Cluster nodes are connected to each other through a backplane LAN like Myrinet or Gigabit Ethernet. Such cluster-based parallel I/O is supported by parallel file systems like PVFS [1], GPFS [14], and Lustre [7].

*This work was supported in part by the National Science Foundation grants CCR-0306473, ITR/IIS-0312925, CNS-0615045, CCF-0621472, NSF CAREER Award CCF-0448413, and an IBM Faculty Award.

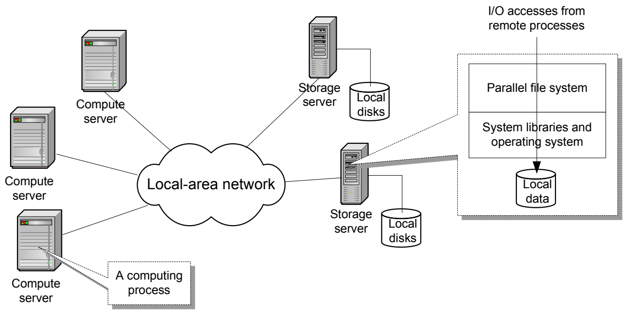

Figure 1. An illustration of parallel application on cluster-based parallel storage.

Typically, MPI parallel applications access data in the parallel file system through the MPI-IO library [2, 17]. The MPI-IO interface allows multiple processes to access I/O data simultaneously with correct semantics. Application dataset is often partitioned or striped over multiple storage servers for high aggregate I/O throughput. Each parallel I/O operation often consists of sub-actions at multiple storage servers, managed by I/O middleware and the host OS at those nodes. Figure 1 provides an illustration of the targeted system architecture.

Despite the promises of scalable I/O performance, specific implementations of these systems under particular workloads may not achieve intended high performance. Performance analysis for these systems are inherently complex for at least two reasons. First, the application I/O workload is transformed several times in the deep I/O stack (by the MPI-IO library, by the parallel file system, by system libraries, and by storage server host OS) before reaching the storage devices. Second, two independent flavors of parallelism (one for the parallel program and the other for the parallel storage) co-exist in one system.

Event tracing at multiple layer boundaries in the deep I/O stack may assist pinpointing the system layers responsible for performance problems. This paper discusses issues in such multi-layer event tracing and analysis. First, to facilitate the analysis on cross-layer I/O characteristics evolution, we focus on generic events and I/O characteristics that apply to all layers (*i.e.*, not specific to any particular layer functions). Specifically, we use a few generic I/O events such as request admission, issuance, completion, and callback to derive a rich set of useful characteristics like the I/O request granularity, sequentiality, and workload concurrency, and idleness. Second, since the lower layer is closer to the raw storage device whose characteristics are relatively well understood, it is easier to realize anomalous I/O characteristics (that trigger worse-than-expected performance) at a lower layer. Consequently we perform multi-layer anomaly identification in a bottom-up fashion.

The rest of this paper is organized as follows. Section 2 evaluates the performance of five MPI-IO applications on a parallel I/O system. Section 3 presents our approach for multi-layer event tracing and analysis. In section 4, we provide a specific case study that has identified three performance anomalies and pinpoint their root causes. Section 5 presents our performance tuning techniques that address some of the discovered problems. Section 6 discusses related work and Section 7 concludes the paper.

2 Application Performance Evaluation

Through application performance evaluation, we underline the existence of performance anomalies in real systems.

We set up a computing and storage cluster with 14 machines each with dual 2 GHz Xeon processors and 2 GB memory. The native OS kernel and C runtime library on all machines are Linux 2.6.12 and GNU libc 2.4 respectively. Among the 14 machines, eight are used as compute servers and the remaining six are set up as dedicated storage servers. Each compute server can run up to two MPI processes (one per processor) and therefore a total of 16 MPI processes can run simultaneously on the cluster. MPI programs in our platform are supported by MPICH2 1.0.3 compiled with ROMIO [17]. Each storage server is equipped with an IBM 10 KRPM SCSI drive with raw seek time in the range of 1.3–9.5 ms (depending on seek distance) and raw sequential transfer rate in the range of 33.8–66.0 MBytes/sec (depending on the disk zone where the data is located). We install PVFS2 1.4.0 on the storage servers. Large files are striped at 64 KB blocks across the six storage servers by PVFS2. All machines are connected through a switched Gigabit Ethernet with TCP/IP roundtrip latency at around 80 us.

2.1 Evaluated Applications

Five MPI-IO applications with different I/O workload characteristics are collected to benchmark the MPI-IO and parallel file system performance. All applications support both read and write modes.

1. `mpi-io-test` is an MPI-IO benchmark provided in the PVFS2 software package[1]. For a read/write test run with N MPI processes, all processes open the same file and each process i ($0 \leq i < N$) reads/writes block i, block $N + i$, block $2N + i$, $\cdots$, until the end of the file. The block size, representing the data access granularity, can be adjusted for each test. Our tests use five block sizes: 64 KB, 256 KB, 1 MB, 4 MB and 16 MB. The write tests have two modes depending on whether the `sync()` operation is called after each write or only once at the end.

2. `ior_mpiio` [3] is an MPI-IO benchmark as part of the ASCI Purple Benchmarks developed at Lawrence Livermore National Laboratory. Each test run consists of writing data into a file and reading the data back. Each MPI process reads/writes an equal-sized block of contiguous bytes that constitutes the whole data file without gaps or overlaps. The I/O request size is adjustable and our tests use five request sizes: 64 KB, 256 KB, 1 MB, 4 MB and 16 MB. Each write test performs one `sync()` operation at the end.

3. `mandelbrot-par` [18] is an MPI-IO application that generates the Mandelbrot dataset, stores it as a single data file, and reads it back for visualization. The data file is partitioned into equal-sized blocks of contiguous bytes for each MPI process. For N MPI processes manipulating a data file of size S, each process i ($0 \leq i < N$) reads/writes the block from file offset $S \times i/N$ to $S \times (i + 1)/N - 1$. A data file of 8 GB large is used for our tests. Each write test performs one `sync()` operation at the end.

4. `mpi-tile-io` [9] is an MPI-IO program from the Parallel I/O Benchmarking Consortium at Argonne National Laboratory. It divides a data file into tiles in a two dimensional space. Each MPI process is assigned one tile for read/write operations. In our tests, we only split the file vertically (*i.e.*, each process gets a strip tile in the space). Our test uses an 8 GB data file. Each write test performs one `sync()` operation at the end.

5. `NPB3.2IO-MPI` [10] a program to solve the Block-Tridiagonal problem in NAS Parallel Benchmarks (NPB) from NASA. The original program only tests the write performance. We extended it to support read tests by having it read over the data file with the same access patterns of the original writes. We use problem size C in the test and the data file size is around 6.8 GB. `NPB3.2IO-MPI` only allows a perfect square number of MPI processes. We use 1, 4, 9 and 16 processes in the tests. Each write test performs one `sync()` operation at the end.

[1] The original `mpi-io-test` in the PVFS2 software package does not clean memory cache at the storage servers before each read throughput test and thus it provides inflated throughput results. We fix it by flushing storage server memory cache before each test.

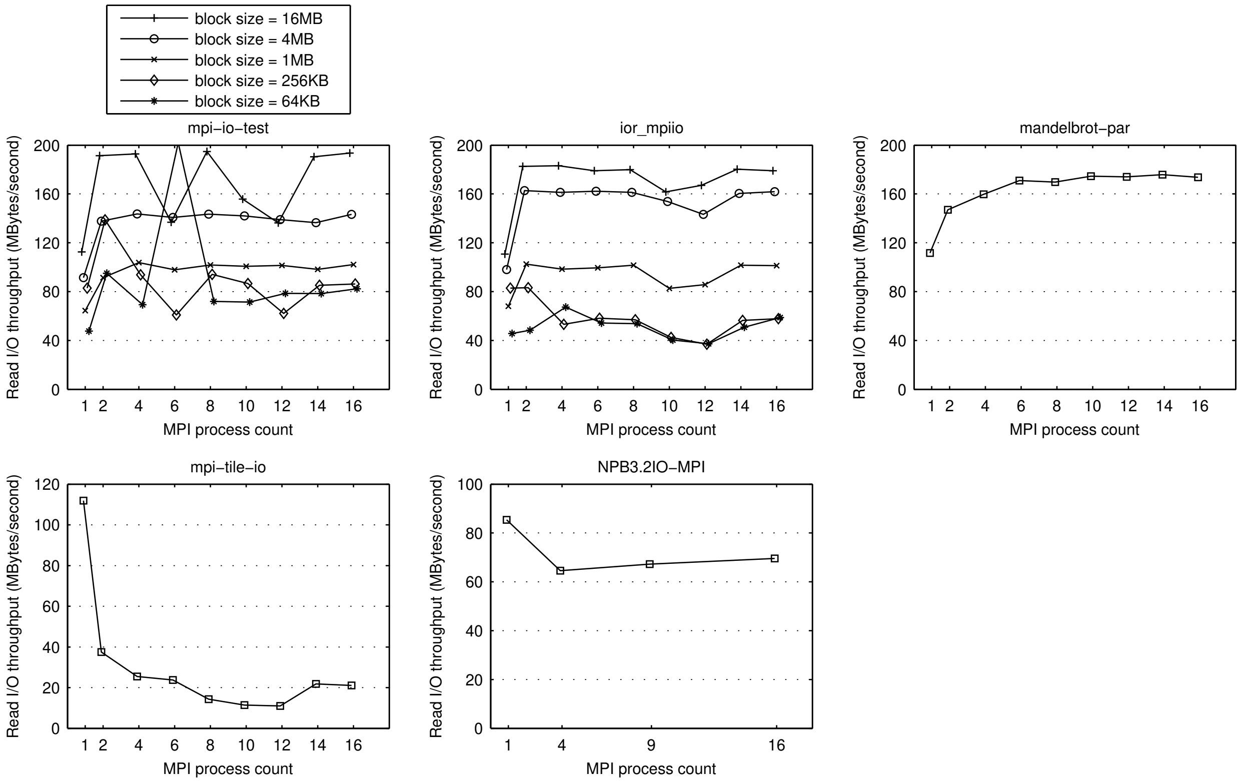

Figure 2. Read throughput for five MPI-IO applications. Both `mpi-io-test` **and** `ior_mpiio` **allow configurable data access block sizes at the application level and we show results at five different block sizes. The parallel file system stripes data over six storage servers for all tests.**

2.2 Experiment Results and Observations

To isolate the read and write I/O performance, we run the five applications in read-only mode and write-only mode separately. Results are reported and analyzed below.

Read I/O performance Figure 2 presents the read throughput for the five applications at up to 16 MPI processes. For `mpi-io-test`, when the block size is large (*e.g.*, 4 MB or larger), the performance curve indicates good scalability of the system with increasing number of MPI processes. Understandably, the read throughput for block size 16 MB with one MPI process is limited by the bandwidth of Gigabit Ethernet. The top read throughput across all tests can be interpreted as the maximum aggregate throughput that the entire system can possibly achieve, which is around 202 MB/sec. It is observed from the figure that the read throughput can be two or three fold higher for larger-granularity accesses. One obvious exception to the above performance trend is the high read throughput at 6 MPI processes and 64 KB block size. This is due to a coincidental absence of I/O concurrency for the following reason. At this setting, the parallelism of the MPI program equals that of the storage servers and the program I/O block size equals the storage server striping block size. Therefore all I/O accesses from each MPI process is directed to one dedicated storage server (*i.e.*, no I/O concurrency and contention at each storage server).

From `ior_mpiio`, we observe similar performance trend that large read I/O request granularity has considerably better performance compared with small granularity I/O. The difference can be up to four to five folds.

The performance results for `mandelbrot-par` perfectly match our expectation from scalable MPI-IO and parallel file system. The read I/O throughput climbs as the number of MPI processes increases.

In `mpi-tile-io`, the read throughput dramatically drops while the system starts to have concurrent MPI process executions. And it continues to decrease when the concurrency increases. This is due to the particular application access pattern. In our parameter setting, high MPI process count in `mpi-tile-io` results in highly fine-grained (usually below the strip size) concurrent I/O requests to multiple storage servers. The coordination and the communication overhead are therefore more significant. Note that this particular application parameter setting is used for examining the system performance and may not be practical

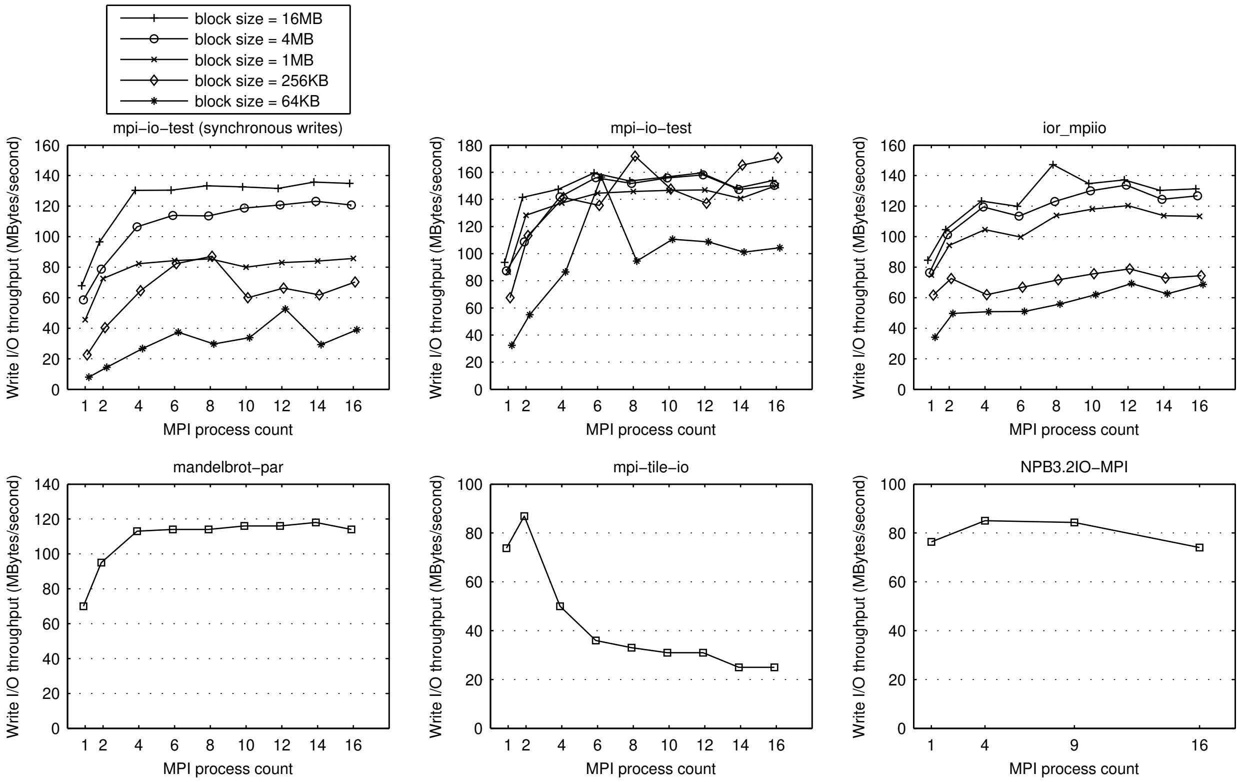

Figure 3. Write throughput for five MPI-IO applications. Note that `mpi-io-test` **has two write modes and the results are presented separately.**

in a real system.

The performance results for `NPB3.2IO-MPI` are similar but slightly different from those of `mpi-tile-io`. The difference is that the read throughput is actually increasing instead of decreasing when the concurrency level is being raised. It suggests that the I/O scheduler works well in this case when highly concurrent I/O operations are presented to the host OS at storage servers.

Overall, the read test results from four out of the five applications exhibit dissatisfying I/O performance. They include `mpi-io-test` with up to 1 MB block size, `mpi-tile-io`, `NPB3.2IO-MPI`, and `ior_mpiio` with up to 1 MB block size. Specifically, the applications tend to exhibit poor read performance when the I/O access granularity is not large (*e.g.*, up to 128 KB per storage server). In some cases the performance keeps dropping when the number of MPI processes increases. This suggests there might be inefficient I/O operations taking place at the storage servers with fine-grained, concurrent read I/O requests.

Write I/O performance Figure 3 illustrates the write I/O throughput results for the five applications. For `mpi-io-test` with synchronous writes, when the block size is small, the write I/O throughput is low. This is understandable because each synchronous write has to wait the disk head to seek and disk plate to rotate to the exact position where the data should be written to. More fine-grained writes result in more time spent in seek and rotation, which greatly reduces the I/O efficiency. Note that an important difference between synchronous writes and reads is that OS prefetching may increase the access granularity of read requests but it cannot help writes directly.

We expect that the rest of the write tests have write I/O throughput that are independent of the I/O granularity since they synchronize the writes only at the end of the test. However, the results of `mpi-io-test` and `ior_mpiio` show that the write performance for fine-grained MPI-IO accesses (*e.g.*, up to 1 MB MPI-IO access size) is still visibly inferior. Therefore, our investigation focus for write is `mpi-io-test` (with write synchronizations only at the end) and `ior_mpiio` with block size up to 1 MB.

In `mpi-tile-io`, the write I/O throughput sharply reduces as the number of MPI processes increases, due to the same reason as in the read test.

3 Multi-Layer Event Tracing and Analysis

Figure 4 provides a high-level illustration of system components involved in an I/O operation for MPI-IO and

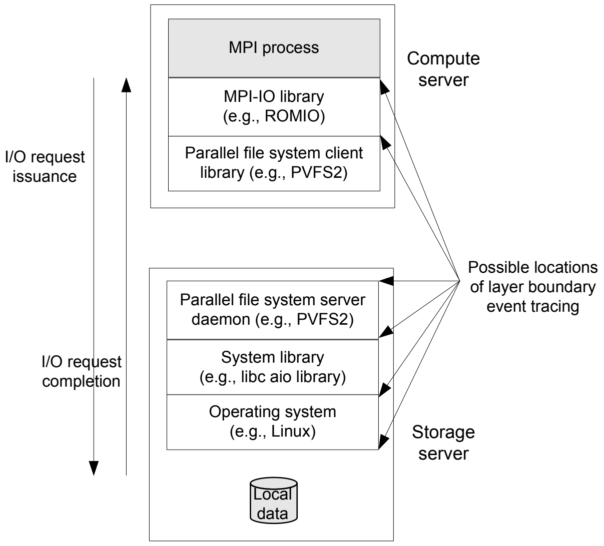

Figure 4. System components involved in an I/O operation and possible locations of layer boundary event tracing.

parallel file system. The parallel file system client library is usually well integrated with the host file system on each compute server and hidden from the user applications. All I/O requests made to the parallel file system are transparently handled by the client library, which takes care of the communication, connection establishment, metadata manipulation and data transfer with appropriate parallel file system server daemons on storage servers across the network. The server daemon is responsible for managing its storage domain and serving the I/O requests received from the client library. The server daemon interacts with system libraries and host OS at the storage server to perform actual I/O operations on the physical disk.

The entire system is multi-layered and all layers together form a deep I/O stack. Each read/write operation in user MPI programs takes a round-trip through all layers of the I/O stack. On the issuance path, each I/O request proceeds top-down from the MPI program all the way to the disk, where every layer admits and processes the request from its above layer and then issues it to the layer beneath itself. On the completion path, the notification travels back in a bottom-up fashion through the stack in the way that every layer receives the notification message from its lower layer and passes it to the upper layer. Layers often support I/O parallelism such that a layer allows the existence of simultaneous I/O streams (*i.e.*, concurrent issuance and completion notification) to increase the efficiency.

In such a system, each layer may transform and optimize I/O request characteristics and completed I/O results in ways that relate to functions at other layers. Performance problems may occur at a layer due to incorrect implementation of the layer itself or inefficient coupling among multiple layers. However, finding the sources of those problems is challenging due to I/O concurrency and the multilayering system structure. We propose to trace important I/O events at layer boundaries of the I/O stack. Characteristics extracted through these traces allow us to discover the transformation from normal to abnormal behaviors at specific layers, which consequently help pinpointing sources of performance problems. Specifically, we mark possible locations of layer boundary event tracing in Figure 4.

We follow two important principles in our approach:

- *Generic (layer-independent) events and I/O characteristics*. Although one can trace a variety of events and derive many I/O characteristics at multiple system layers, the subsequent analysis may be difficult given the semantic details of different layer functions. To facilitate automated cross-layer analysis, we only trace generic events and derive a set of generic I/O characteristics. Here by *generic*, we mean an event or an I/O characteristic that is not specific to any particular layer function and thus can apply to all system layers. It is much easier to track the evolution of a generic I/O characteristics (*e.g.*, I/O request concurrency) between two layers than to compare different I/O characteristics that are only meaningful to specific layer functions. We describe an example set of traced generic events and derived generic I/O characteristics later.

- *Bottom-up trace analysis*. Our trace analysis starts at a low system layer and proceeds upwards. We find this approach advantageous for the following reason. A lower system layer is closer to the raw storage device whose characteristics are relatively well understood, and therefore it is easier to pinpoint anomalous I/O characteristics that trigger lower-than-expected performance. The anomaly labeling can be either manual, or it can be performed automatically by comparing with a reasonable value range for each I/O characteristics collected from known anomaly-free systems. Based on the labeled anomalous I/O characteristics, we then perform a layer-by-layer comparison to uncover the highest layer where the anomalous I/O characteristics was first introduced.

Based on the two principles, we can systematically derive anomalous I/O characteristics and the range of system layers that exhibit such anomalies. Although the highest manifestation layer is typically the source of problem, other causes may also be possible and a human performance analyst will take over from this point on.

Traced events and derived I/O characteristics We trace the events of I/O request being passing through layer boundaries. The life span of each I/O request in a layer is marked

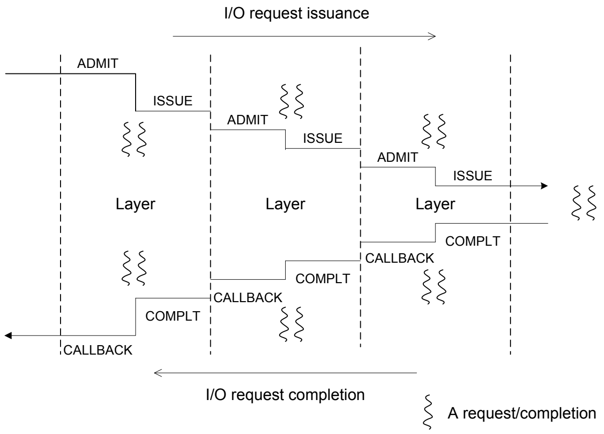

Figure 5. Traced layer boundary events.

by four types of generic boundary events. Admission event (or *ADMIT*) denotes the receipt of an I/O request from upper layer. Issuance event (or *ISSUE*) denotes the action of a layer passing an I/O request to lower layer. Completion event (or *COMPLT*) denotes that the acknowledgment of completion notification of an I/O request received from lower layer. Callback event (or *CALLBACK*) denotes that the layer notifies its upper layer about an I/O request's completion. Figure 5 provides an illustration of these events.

In the event trace, we record each I/O request event's timestamp, associated I/O file descriptor (`fd`), offset (`offset`) and byte count (`nbytes`). Based on the trace, we are able to derive a set of key characteristics that describe an I/O workload between two adjacent layers in the system.

- *Sequential Access Stream*: a group of spatially contiguous data items that are requested by a layer within an application execution. Note that a layer may not continuously access the stream at once. In other words, during an application execution, a layer may alternately access multiple streams in an interleaved fashion. This notion depicts the overall spatial contiguity of the layer's I/O footprint.
- *Sequential Access Run*: a portion of a sequential stream that is accessed contiguously without interleaved I/O. Sequential access run reflects the granularity of sequential accesses.
- *Maximum/Average I/O Request Size*: the maximum/average size of an I/O request issued by a layer. This metric indicates the request granularity.
- *Maximum/Average Concurrency*: the maximum/average number of simultaneous outstanding I/O requests in a layer during an application execution. We say that an I/O request is outstanding at a layer when its ADMIT in this layer has happened but CALLBACK has not. The I/O concurrency at an instant τ, called $\mathcal{C}(\tau)$, is defined as the number of outstanding I/O requests at the instant. Formally, the average I/O concurrency metric over a time period $[\tau_{start}, \tau_{end}]$ is defined as $\frac{\int_{\tau_{start}}^{\tau_{end}} \mathcal{C}(\tau)\,\mathrm{d}\tau}{\tau_{end}-\tau_{start}}$.
- *Throughput*: the total size of requested data completed in a layer divided by the time during which the layer is busy. We say that a layer is busy if in that particular moment there is at least one outstanding I/O request at this layer.
- *Timing breakdown*: the detailed and timed activities information of a layer and the detailed timing information of an I/O request in a layer. The timing breakdown can be utilized to monitor the layer activities and unexpectedly long waiting or blocking in any phase between two boundary events.

4 Trace Analysis Case Study

We provide a specific case study on trace analysis and anomaly root cause investigation. In our experimental system, the PVFS2 server daemon on storage server issues I/O operations through POSIX asynchronous I/O (`aio`) API. The asynchronous I/O is not directly supported by the Linux operating system. Instead, the runtime GNU C runtime library provides an emulation of asynchronous I/O.

Following the approach described in Section 3, we instrumented the MPI-IO library, the PVFS2 client, the PVFS2 server and the `aio` library to collect layer boundary event traces. The event traces are compact in size. They are aggregated in a separate memory area during the application executions and dumped to disk after the executions are completed. The overhead of our instrumentation is negligible since no extra I/O operations are incurred for the executions. In the rest of this paper, we use a layer prefix together with ADMIT, ISSUE, COMPLT and CALLBACK to denote the boundary events at a particular layer (*e.g.*, PVFS2_ISSUE represents the issuance of an I/O request from the PVFS2 server daemon to its lower layer — the `aio` emulation in the C runtime library).

4.1 Case I: Asynchronous I/O

We repeat the read tests that previously showed unsatisfactory performance on the instrumented system. We collect event traces and process them to derive I/O characteristics introduced in Section 3. Following the bottom-up trace analysis approach, the first anomaly that we observe is lack of I/O request concurrency at the OS level. The `aio` library never issues more than one I/O request at a time. This is confirmed by a detailed look at the collected traces — Figure 6 presents a trace segment with serialized AIO_ISSUE and AIO_COMPLT events.

Lack of concurrency at the OS level may seriously impair the effectiveness of elevator I/O scheduler in the OS.

```
Time:  2650212.242744   EVENT: AIO_ISSUE    OP: READ  FD: 15  OFFSET:712114176  NBYTES:65536
Time:  2650212.244758   EVENT: AIO_COMPLT   OP: READ  FD: 15  OFFSET:712114176  NBYTES:65536
Time:  2650212.244823   EVENT: AIO_ISSUE    OP: READ  FD: 15  OFFSET:712179712  NBYTES:65536
Time:  2650212.249684   EVENT: AIO_COMPLT   OP: READ  FD: 15  OFFSET:712179712  NBYTES:65536
Time:  2650212.249731   EVENT: AIO_ISSUE    OP: READ  FD: 15  OFFSET:712310784  NBYTES:65536
Time:  2650212.257953   EVENT: AIO_COMPLT   OP: READ  FD: 15  OFFSET:712310784  NBYTES:65536
Time:  2650212.258010   EVENT: AIO_ISSUE    OP: READ  FD: 15  OFFSET:712900608  NBYTES:65536
Time:  2650212.273527   EVENT: AIO_COMPLT   OP: READ  FD: 15  OFFSET:712900608  NBYTES:65536
........
```

Figure 6. I/O trace segment showing request serialization in the `aio` library.

This is because elevator I/O scheduler re-orders outstanding I/O requests to minimize the disk head movement and therefore improve I/O throughput. Elevator scheduler works better when there are more outstanding requests to choose from for the purpose of seek reduction. Moreover, serialization of I/O request issuance is more likely to create idle time between system layers and thus decrease the resource utilization efficiency.

A cross-layer analysis shows that the lack of I/O concurrency is limited to the OS layer. The timing breakdown further shows unexpectedly long average waiting time for an I/O request to be issued in the `aio` library (*i.e.*, the phase between AIO_ADMIT and AIO_ISSUE). This potentially suggests excessive I/O request blocking and queuing in the `aio` library. Both observations (the workload concurrency transition and the excessive I/O request blocking in the `aio` library) suggest that that I/O request issuance from the `aio` library is serialized.

We have investigated the `aio` library implementation (GNU libc 2.4) for the cause of I/O request issuance serialization. We find that the `aio` library employs a single thread to serve pending I/O requests on each file descriptor. And all requests are issued to the OS through blocking I/O system call `pread`/`pwrite`. Under this implementation, all I/O requests on the same file descriptor are strictly ordered without concurrency. At the same time, the PVFS2 server daemon employs a single file descriptor to issue all I/O operations on one file (regardless whether these operations come from multiple remote MPI processes). For parallel applications that operate on a single large file, effectively all I/O operations at a storage server are serialized.

There are multiple ways to address the request serialization problem. First, we can lift the `aio` library's one-thread-per-file-descriptor restriction so multiple I/O operations on the same file may be issued to the OS simultaneously. Second, we can augment the PVFS2 server by using different file descriptors for I/O operations from different remote MPI processes (even if these operations are on the same file).

4.2 Case II: Interleaved Read I/O

The collected event traces of `aio` library indicate another anomaly that most sequential access runs are small even when a large amount of spatially contiguous data is eventually accessed (indicating long sequential access stream). As a specific example, Figure 7 displays an I/O trace segment of ISSUE events occurring in the PVFS2 server daemon that illustrates such an interleaved execution. Table 1 shows the distribution of read I/O requests at various sequential access run lengths. The average length of a sequential access run is 73 KB. Considering that the average read I/O request size is 64 KB, only a small portion of adjacent read I/O requests is spatially contiguous.

Sequential access run length	*Percentage*
64 KB	92.4 %
128 KB	4.5 %
256 KB	1.9 %
320 KB	1.1 %
384 KB or larger	0.1 %

Table 1. Percentage of different lengths of sequential access runs in an I/O trace segment.

Upwards trace investigation in the PVFS2 client and server reveals that the concurrent I/O workload is a result of multiple MPI processes accessing the storage server simultaneously. The data accesses to one sequential access stream are therefore frequently interrupted by accesses to other streams under a concurrent I/O workload.

Modern disk drives have large seek and rotational delays. Operating systems (including Linux) employ two main optimization techniques to reduce disk seeks and rotations during concurrent operations: prefetching and anticipatory scheduling [4]. We briefly describe these two techniques and why they do not work well for MPI-IO and parallel file system under interleaved concurrent I/O requests.

- *Prefetching*. The OS reads ahead of the currently requested data in the hope that the upcoming I/O requests are sequential to previous ones. In such cases, data can be immediately returned without requiring any disk access. However, prefetching does not help much in our problematic cases for at least two reasons: *1)* Our previous work [6] found that the default OS prefetching is usually too conservative for I/O-intensive applications. For instance, the prefetching size in default Linux does not go beyond 128 KB. *2)* It is common for MPI programs that many concurrent I/O operations from MPI

```
Time:  2653111.280135   EVENT: PVFS2_ISSUE   OP: READ  FD: 15  OFFSET: 192151552  NBYTES: 65536
Time:  2653111.283790   EVENT: PVFS2_ISSUE   OP: READ  FD: 15  OFFSET: 192020480  NBYTES: 65536
........
Time:  2253111.360168   EVENT: PVFS2_ISSUE   OP: READ  FD: 15  OFFSET: 192937984  NBYTES: 65536
Time:  2253111.361018   EVENT: PVFS2_ISSUE   OP: READ  FD: 15  OFFSET: 192806912  NBYTES: 65536
Time:  2253111.362385   EVENT: PVFS2_ISSUE   OP: READ  FD: 15  OFFSET: 192217088  NBYTES: 65536
Time:  2253111.362916   EVENT: PVFS2_ISSUE   OP: READ  FD: 15  OFFSET: 192086016  NBYTES: 65536
Time:  2253111.363668   EVENT: PVFS2_ISSUE   OP: READ  FD: 15  OFFSET: 193200128  NBYTES: 65536
........
Time:  2653111.452228   EVENT: PVFS2_ISSUE   OP: READ  FD: 15  OFFSET: 193003520  NBYTES: 65536
Time:  2653111.452727   EVENT: PVFS2_ISSUE   OP: READ  FD: 15  OFFSET: 192872448  NBYTES: 65536
........
Time:  2653111.481709   EVENT: PVFS2_ISSUE   OP: READ  FD: 15  OFFSET: 193265664  NBYTES: 65536
```

Figure 7. I/O trace segment showing interleaved read I/O under concurrent operations.

processes are requesting data jointly from the same file. This might confuse the storage server OS to believe the access pattern for this file is non-sequential and thus curb or even disable prefetching.

- *Anticipatory Scheduling*. At the completion of an I/O request, anticipatory scheduling may keep the disk idle for a short period of time even if there are outstanding requests. The scheduler does so in anticipation of a new I/O request from the process that issued the just completed request, which often requires little or no seeking from the current disk head location. Though the anticipatory scheduling may significant improve the performance of concurrent I/O workloads, it is only effective when the I/O scheduler is aware of request-issuing process contexts. However, due to the inherent nature of remote I/O access in parallel I/O systems, the identities of remote MPI processes that issued I/O requests are not available at storage server host OS. Specifically in PVFS2, all I/O requests from remote MPI processes are handled by a single PVFS2 daemon process before entering the OS.

There are multiple ways to address the above performance issue. For instance, the PVFS2 server can employ aggressive I/O prefetching on its own. Additionally, we can also let the PVFS2 server to preserve the request-issuing remote MPI process contexts so the OS anticipatory I/O scheduling can be effective. More specifically, the PVFS2 server can use one unique local proxy process to issue read I/O requests on behalf of each remote MPI process.

4.3 Case III: Lack of Write Buffering

We also collect and analyze the traces for the anomalous write tests on the instrumented system. The timing breakdown at the PVFS2 server layer suggests that the PVFS2 server has low average I/O concurrency and spends excessively long time on idling. For `mpi-io-test` write test at block size 64 KB, the average I/O concurrency is well below 1.0, which suggests that the storage server is under-utilized

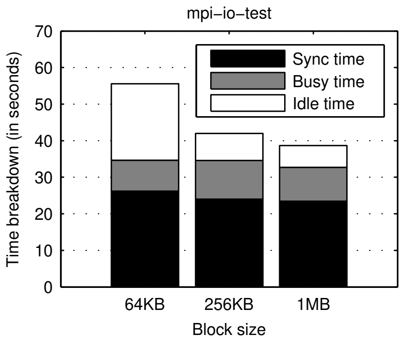

Figure 8. Timing breakdown in a PVFS2 server daemon. Sync time is the time that PVFS2 spends on perform `sync()` operation. Busy and idle time denote the time that the PVFS2 has or does not have outstanding request respectively.

for most of the time. The I/O concurrency improves at large block sizes. Figure 8 further illustrates the time spent in each activity for an individual storage server during the execution. As we can see, The PVFS2 server daemon spends a significant portion of time on waiting for requests when the block size is small, which reduces the I/O efficiency of the entire system. We label the excessive idling at the PVFS2 server as the anomaly in this case.

The bottom-up trace analysis reveals that the PVFS2 server inefficiency for the small writes results from lack of write buffering in the PVFS2 client. Each PVFS2 write call in the client side blocks until the data is synchronously transferred to and buffered in the storage server OSes. Therefore each write request incurs a significant amount of delay. When the application (`mpi-io-test`) employs small write request size, more write requests are needed and thus the write request delay is more pronounced.

Implementing write buffering at the PVFS2 client can help address the above performance problem. With the buffering capability, each MPI-IO write call from the MPI processes could return asynchronously once the data is

Affected scenario	Anomalous I/O characteristics	Identified cause
Read	Lack of concurrency at the OS level	One-thread-per-open-file limitation in `aio` library
Read	Interleaved small I/O	Concurrent accesses from multiple MPI processes; OS prefetching and anticipatory scheduling fail to help
Write	Low concurrency and long idling at PVFS2 server	Lack of write buffering in the PVFS2 client

Table 2. Case study summary.

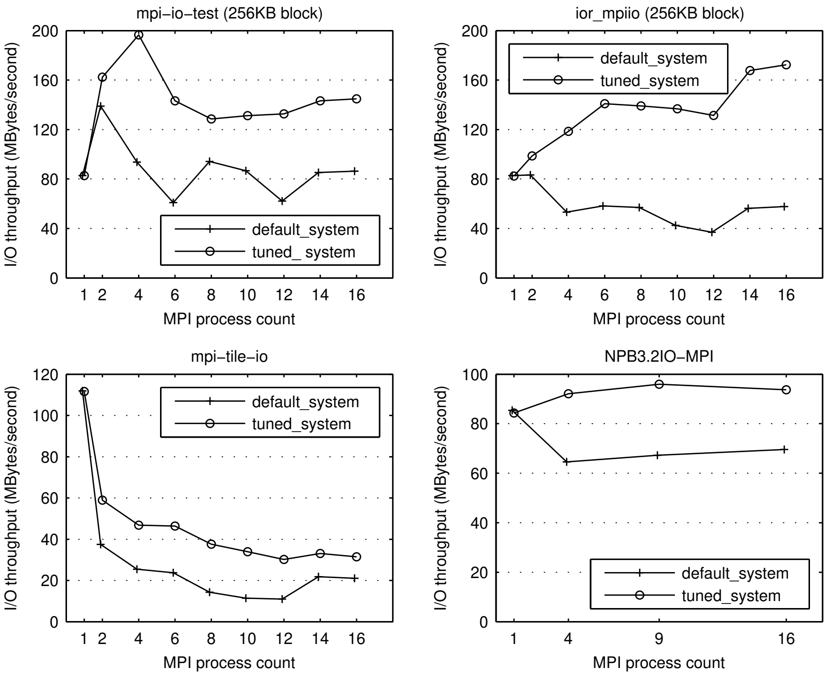

Figure 9. Read throughput of the original system and a tuned system.

handed off to the PVFS2 client. The PVFS2 client is able to further optimize small writes through aggregation and collective transfer.

4.4 Summary

Our proposed collection of generic I/O characteristics and bottom-up event trace analysis allow us discover root causes for three performance anomalies in the experimented system. Table 2 summarizes the trace analysis and root cause identification result. The asynchronous I/O implementation anomaly was also discovered independently by the PVFS2 authors [13]. More aggressive prefetching at the OS level has been investigated [6] to better support concurrent sequential I/O. The lack of write buffering at the PVFS2 client is still listed as an open issue in the PVFS2 development manuscript.

5 Evaluation of A Tuned System

We run experiments to validate that the performance problems discovered in the previous section are indeed causes of performance degradation. Note that our goal here is not to seek perfect corrections to these problems. We employ simple performance fixes to the previously discovered problems in cases I and II. We do not have a fix for case III since it requires more fundamental software changes.

First, we augment the PVFS2 server to prefetch 256 KB sequentially ahead the currently requested data. This would increase the sequential access granularity of I/O requests, addressing the first performance problem (Section 4.1). Second, we let PVFS2 server daemon bypass the `aio` library and issue parallel I/O requests to the operating system directly through multiple helper threads. This would allow concurrent requests at the OS layer, addressing the second performance problem (Section 4.2).

We repeated all application read tests in Section 2 with our tuned system. We use default_system and tuned_system respectively to denote the original and the corrected systems throughout. Results in Figure 9 show that tuned_system has significant throughput improvement over default_system in all problematic tests. On average, tuned_system boosts read I/O performance of the four applications with anomalies by 89%, 110%, 39%, and 156% respectively. The results confirm the soundness and capability of our methodology in findings and pinpointing I/O performance problems in MPI-IO on parallel file system.

6 Related work

Previous research has recognized the importance of providing cluster-based software infrastructure for I/O-intensive parallel applications. These efforts include continued development of parallel file systems [1, 7, 12] and the integration with parallel I/O standards like MPI-IO [2, 17]. Despite the success of these efforts, there is still a lack of comprehensive understanding on performance issues associated with concurrent I/O workloads, where I/O requests from multiple compute processes arrive simultaneously at the cluster-based parallel storage.

Tracing has been commonly employed as a general approach to debug programs and study system or application behaviors. In the high performance computing area, I/O tracing is used to study scientific workload characteristics [5, 11], to model and predict the system I/O behaviors [8], and to make access pattern adjustments to improve I/O efficiency [16]. Our work in this paper complements these previous studies by focusing on the discovery of performance problems and their causes in systems with deep I/O stacks (specifically MPI-IO on parallel file systems). In this paper, we demonstrate that the collection of a small number of generic I/O characteristics at multiple system layer boundaries are effective for this purpose.

There has been extensive studies on operating system (OS) support for I/O-intensive applications, in terms of I/O prefetching [6], scheduling [4], and performance anomaly analysis [15]. However, OS support targeting general-purpose workloads may not work well for high-end computing (HEC) applications with parallel I/O. In particular, many HEC applications exhibit large sequential I/O access pattern and thus more aggressive OS prefetching is desirable. Additionally, the anticipatory scheduling [4] may be ineffective due to the lack of information on remote process contexts in a parallel I/O architecture. Our performance analysis uncovered these problems which call for new OS I/O subsystem design on dedicated HEC platforms.

7 Conclusion

In this paper, through our multi-layer trace analysis, we discover that current OS and parallel file system may inefficiently manage I/O requests of concurrent MPI programs. Our approach follows two principles: 1) collect generic (layer-independent) events and I/O characteristics to ease the analysis on cross-layer I/O characteristics evolution; 2) perform bottom-up trace analysis to take advantage of the relatively easy anomaly identification at lower system layers. Our empirical study on a specific parallel I/O system uncovers the causes to three performance anomalies. Beyond the specific case study in this paper, we believe the multi-layer trace analysis technique is general and should be applicable to other systems with deep I/O stacks.

References

[1] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proc. of the 4th Annual Linux Showcase and Conf.*, pages 317–327, Oct. 2000.

[2] S. A. Fineberg, P. Wong, B. Nitzberg, and C. Kuszmaul. PMPIO - A Portable Implementation of MPI-IO. In *Proc. of the 6th Symp. on The Frontiers of Massively Parallel Computation*, pages 188–195, Oct. 1996.

[3] I/O Stress Benchmark Codes Web Page at LLNL. http://www.llnl.gov/asci/purple/benchmarks/limited/ior/.

[4] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, pages 117 – 130, Oct. 2001.

[5] D. Kotz and N. Nieuwejaar. File System Workload on a Scientific Multiprocessor. *IEEE Parallel and Distributed Technology*, 3(1):51–60, 1995.

[6] C. Li, K. Shen, and A. E. Papathanasiou. Competitive Prefetching for Concurrent Sequential I/O. In *Proc. of the Second EuroSys Conf.*, pages 189–202, Mar. 2007.

[7] Lustre Cluster File System. http://www.lustre.org.

[8] E. Miller and R. Katz. Input/Output Behavior of Supercomputing Applications. In *Proc. of Supercomputing*, pages 567–576, Nov. 1991.

[9] Parallel I/O Benchmarking Consortium at ANL. http://www-unix.mcs.anl.gov/ rross/pio-benchmark/.

[10] NAS Parallel Benchmarks Web Page. http://www.nas.nasa.gov/Software/NPB/.

[11] B. Pasquale and G. Polyzos. A Static Analysis of I/O characteristics of Scientific Applications in a Production Workload. In *Proc. of Supercomputing*, pages 388–397, 1993.

[12] The Parallel Virtual File System, Version 2. http://www.pvfs.org/pvfs2.

[13] R. B. Ross. Argonne National Lab, Aug. 2006. Personal Communication.

[14] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proc. of the USENIX Conf. on File and Storage Technologies*, pages 231–244, Jan. 2002.

[15] K. Shen, M. Zhong, and C. Li. I/O System Performance Debugging Using Model-driven Anomaly Characterization. In *Proc. of the 4th USENIX Conf. on File and Storage Technologies*, pages 309–322, Dec. 2005.

[16] E. Smirni, R. Aydt, A. Chien, and D. Reed. I/O Requirements of Scientific Applications: An Evolutionary View. In *Proc. of the 5th IEEE Int'l Symp. on High Performance Distributed Computing*, pages 49–59, 1996.

[17] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proc. of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32, May 1999.

[18] Y. Wang. Dept. of Electrical and Computer Engineering, Northeastern University, 2006. Personal Communication.