Linux AIO Performance and Robustness for Enterprise Workloads

Suparna Bhattacharya, IBM (suparna@in.ibm.com)
John Tran, IBM (jbtran@ca.ibm.com)
Mike Sullivan, IBM (mksully@us.ibm.com)
Chris Mason, SUSE (mason@suse.com)

1 Abstract

In this paper we address some of the issues identified during the development and stabilization of Asynchronous I/O (AIO) on Linux 2.6.

We start by describing improvements made to optimize the throughput of streaming buffered filesystem AIO for microbenchmark runs. Next, we discuss certain tricky issues in ensuring data integrity between AIO Direct I/O (DIO) and buffered I/O, and take a deeper look at synchronized I/O guarantees, concurrent I/O, write-ordering issues and the improvements resulting from radix-tree based write-back changes in the Linux VFS.

We then investigate the results of using Linux 2.6 filesystem AIO on the performance metrics for certain enterprise database workloads which are expected to benefit from AIO, and mention a few tips on optimizing AIO for such workloads. Finally, we briefly discuss the issues around workloads that need to combine asynchronous disk I/O and network I/O.

2 Introduction

AIO enables a single application thread to overlap processing with I/O operations for better utilization of CPU and devices. AIO can improve the performance of certain kinds of I/O intensive applications like databases, web-servers and streaming-content servers. The use of AIO also tends to help such applications adapt and scale more smoothly to varying loads.

2.1 Overview of kernel AIO in Linux 2.6

The Linux 2.6 kernel implements in-kernel support for AIO. A low-level native AIO system call interface is provided that can be invoked directly by applications or used by library implementations to build POSIX/SUS semantics. All discussion hereafter in this paper pertains to the native kernel AIO interfaces.

Applications can submit one or more I/O requests asynchronously using the `io_submit()` system call, and obtain completion notification using the `io_getevents()` system call. Each I/O request specifies the operation (typically read/write), the file descriptor and the parameters for the operation (e.g., file offset, buffer). I/O requests are associated with the completion queue (ioctx) they were submitted against. The results of I/O are reported as completion events on this queue, and reaped using `io_getevents()`.

The design of AIO for the Linux 2.6 kernel has been discussed in [1], including the motivation

behind certain architectural choices, for example:

- Sharing a common code path for AIO and regular I/O
- A retry-based model for AIO continuations across blocking points in the case of buffered filesystem AIO (currently implemented as a set of patches to the Linux 2.6 kernel) where worker threads take on the caller's address space for executing retries involving access to user-space buffers.

2.2 Background on retry-based AIO

The retry-based model allows an AIO request to be executed as a series of non-blocking iterations. Each iteration retries the remaining part of the request from where the last iteration left off, re-issuing the corresponding AIO filesystem operation with modified arguments representing the remaining I/O. The retries are "kicked" via a special AIO waitqueue callback routine, `aio_wake_function()`, which replaces the default waitqueue entry used for blocking waits.

The high-level retry infrastructure is responsible for running the iterations in the address space context of the caller, and ensures that only one retry instance is active at a given time. This relieves the fops themselves from having to deal with potential races of that sort.

2.3 Overview of the rest of the paper

In subsequent sections of this paper, we describe our experiences in addressing several issues identified during the optimization and stabilization efforts related to the kernel AIO implementation for Linux 2.6, mainly in the area of disk- or filesystem-based AIO.

We observe, for example, how I/O patterns generated by the common VFS code paths used by regular and retry-based AIO could be non-optimal for streaming AIO requests, and we describe the modifications that address this finding. A different set of problems that has seen some development activity are the races, exposures and potential data-integrity concerns between direct and buffered I/O, which become especially tricky in the presence of AIO. Some of these issues motivated Andrew Morton's modified page-writeback design for the VFS using tagged radix-tree lookups, and we discuss the implications for the AIO `O_SYNC` write implementation. In general, disk-based filesystem AIO requirements for database workloads have been a guiding consideration in resolving some of the trade-offs encountered, and we present some initial performance results for such workloads. Lastly, we touch upon potential approaches to allow processing of disk-based AIO and communications I/O within a single event loop.

3 Streaming AIO reads

3.1 Basic retry pattern for single AIO read

The retry-based design for buffered filesystem AIO read works by converting each blocking wait for read completion on a page into a *retry exit*. The design queues an asynchronous notification callback and returns the number of bytes for which the read has completed so far without blocking. Then, when the page becomes up-to-date, the callback kicks off a retry continuation in task context. This retry continuation invokes the same filesystem read operation again using the caller's address space, but this time with arguments modified to reflect the remaining part of the read request.

For example, given a 16KB read request starting at offset 0, where the first 4KB is already in cache, one might see the following sequence of retries (in the absence of readahead):

```
first time:
   fop->aio_read(fd, 0, 16384) = 4096
and when read completes for the second page:
   fop->aio_read(fd, 4096, 12288) = 4096
and when read completes for the third page:
   fop->aio_read(fd, 8192, 8192) = 4096
and when read completes for the fourth page:
   fop->aio_read(fd, 12288, 4096) = 4096
```

3.2 Impact of readahead on single AIO read

Usually, however, the readahead logic attempts to batch read requests in advance. Hence, more I/O would be seen to have completed at each retry. The logic attempts to predict the optimal readahead window based on state it maintains about the sequentiality of past read requests on the same file descriptor. Thus, given a maximum readahead window size of 128KB, the sequence of retries would appear to be more like the following example, which results in significantly improved throughput:

```
first time:
   fop->aio_read(fd, 0, 16384) = 4096,
     after issuing readahead
     for 128KB/2 = 64KB
and when read completes for the above I/O:
   fop->aio_read(fd, 4096, 12288) = 12288
```

Notice that care is taken to ensure that readaheads are not repeated during retries.

3.3 Impact of readahead on streaming AIO reads

In the case of streaming AIO reads, a sequence of AIO read requests is issued on the same file descriptor, where subsequent reads are submitted without waiting for previous requests to complete (contrast this with a sequence of synchronous reads).

Interestingly, we encountered a significant throughput degradation as a result of the interplay of readahead and streaming AIO reads. To see why, consider the retry sequence for streaming random AIO read requests of 16KB, where `o1, o2, o3, ...` refer to the random offsets where these reads are issued:

```
first time:
  fop->aio_read(fd, o1, 16384) = -EIOCBRETRY,
    after issuing readahead for 64KB
    as the readahead logic sees the first page
    of the read
  fop->aio_read(fd, o2, 16384) = -EIOCBRETRY,
    after issuing readahead for 8KB (notice
    the shrinkage of the readahead window
    because of non-sequentiality seen by the
    readahead logic)
  fop->aio_read(fd, o3, 16384) = -EIOCBRETRY,
    after maximally shrinking the readahead
    window, turning off readahead and issuing
    4KB read in the slow path
  fop->aio_read(fd, o4, 16384) = -EIOCBRETRY,
    after issuing 4KB read in the slow path
.
.
and when read completes for o1
  fop->aio_read(fd, o1, 16384) = 16384
and when read completes for o2
  fop->aio_read(fd, o2, 16384) = 8192
and when read completes for o3
  fop->aio_read(fd, o3, 16384) = 4096
and when read completes for o4
  fop->aio_read(fd, o3, 16384) = 4096
.
.
```

In steady state, this amounts to a maximally-shrunk readahead window with 4KB reads at random offsets being issued serially one at a time on a slow path, causing seek storms and driving throughputs down severely.

3.4 Upfront readahead for improved streaming AIO read throughputs

To address this issue, we made the readahead logic aware of the sequentiality of all pages in a single read request upfront—before submitting the next read request. This resulted in a more desirable outcome as follows:

```
  fop->aio_read(fd, o1, 16384) = -EIOCBRETRY,
    after issuing readahead for 64KB
    as the readahead logic sees all the 4
    pages for the read
  fop->aio_read(fd, o2, 16384) = -EIOCBRETRY,
    after issuing readahead for 20KB, as the
    readahead logic sees all 4 pages of the
    read (the readahead window shrinks to
    4+1=5 pages)
```

```
  fop->aio_read(fd, o3, 16384) = -EIOCBRETRY,
    after issuing readahead for 20KB, as the
    readahead logic sees all 4 pages of the
    read (the readahead window is maintained
    at 4+1=5 pages)
        .
        .
and when read completes for o1
  fop->aio_read(fd, o1, 16384) = 16384
and when read completes for o2
  fop->aio_read(fd, o2, 16384) = 16384
and when read completes for o3
  fop->aio_read(fd, o3, 16384) = 16384
        .
        .
```

3.5 Upfront readahead and sendfile regressions

At first sight it appears that upfront readahead is a reasonable change for all situations, since it immediately passes to the readahead logic the entire size of the request. However, it has the unintended, potential side-effect of losing pipelining benefits for really large reads, or operations like sendfile which involve post processing I/O on the contents just read. One way to address this is to clip the maximum size of upfront readahead to the maximum readahead setting for the device. To see why even that may not suffice for certain situations, let us take a look at the following sequence for a webserver that uses non-blocking sendfile to serve a large (2GB) file.

```
  sendfile(fd, 0, 2GB, fd2) = 8192,
    tells readahead about up to 128KB
    of the read
  sendfile(fd, 8192, 2GB - 8192, fd2) = 8192,
    tells readahead about 8KB - 132KB
    of the read
  sendfile(fd, 16384, 2GB - 16384, fd2) = 8192,
    tells readahead about 16KB-140KB
    of the read
       ...
```

This confuses the readahead logic about the I/O pattern which appears to be 0–128K, 8K–132K, 16K–140K instead of clear sequentiality from 0–2GB that is really appropriate.

To avoid such unanticipated issues, upfront readahead required a special case for AIO alone, limited to the maximum readahead setting for the device.

3.6 Streaming AIO read microbenchmark comparisons

We explored streaming AIO throughput improvements with the retry-based AIO implementation and optimizations discussed above, using a custom microbenchmark called aio-stress [2]. aio-stress issues a stream of AIO requests to one or more files, where one can vary several parameters including I/O unit size, total I/O size, depth of iocbs submitted at a time, number of concurrent threads, and type and pattern of I/O operations, and reports the overall throughput attained.

The hardware included a 4-way 700MHz Pentium® III machine with 512MB of RAM and a 1MB L2 cache. The disk subsystem used for the I/O tests consisted of an Adaptec AIC7896/97 Ultra2 SCSI controller connected to a disk enclosure with six 9GB disks, one of which was configured as an ext3 filesystem with a block size of 4KB for testing.

The runs compared aio-stress throughputs for streaming random buffered I/O reads (i.e., without `O_DIRECT`), with and without the previously described changes. All the runs were for the case where the file was not already cached in memory. The above graph summarizes how the results varied across individual request sizes of 4KB to 64KB, where I/O was targeted to a single file of size 1GB, the depth of iocbs outstanding at a time being 64KB. A third run was performed to find out how the results compared with equivalent runs using AIO-DIO.

With the changes applied, the results showed an approximate 2x improvement across all block sizes, bringing throughputs to levels that match the corresponding results using AIO-DIO.

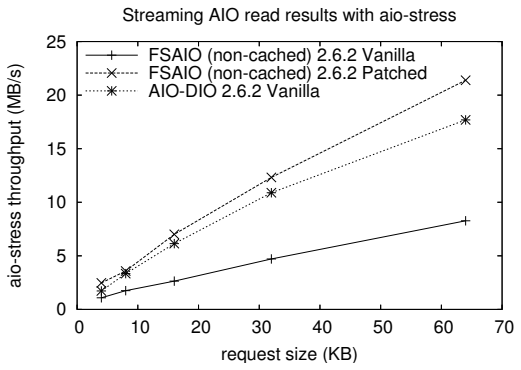

Figure 1: Comparisons of streaming random AIO read throughputs

4 AIO DIO vs cached I/O integrity issues

4.1 DIO vs buffered races

Stephen Tweedie discovered several races between DIO and buffered I/O to the same file [3]. These races could lead to potential stale-data exposures and even data-integrity issues. Most instances were related to situations when in-core meta-data updates were visible before actual instantiation or resetting of corresponding data blocks on disk. Problems could also arise when meta-data updates were not visible to other code paths that could simultaneously update meta-data as well. The races mainly affected sparse files due to the lack of atomicity between the file flush in the DIO paths and actual data block accesses.

The solution that Stephen Tweedie came up with, and which Badari Pulavarty reported to Linux 2.6, involved protecting block lookups and meta-data updates with the inode semaphore (`i_sem`) in DIO paths for both read and write, atomically with the file flush. Overwriting of sparse blocks in the DIO write path was modified to fall back to buffered writes. Finally, an additional semaphore (`i_alloc_sem`) was introduced to lock out deallocation of blocks by a truncate while DIO was in progress. The semaphore was implemented held in shared mode by DIO and in exclusive mode by truncate.

Note that handling the new locking rules (i.e., lock ordering of `i_sem` first and then `i_alloc_sem`) while allowing for filesystem-specific implementations of the DIO and file-write interfaces had to be handled with some care.

4.2 AIO-DIO specific races

The inclusion of AIO in Linux 2.6 added some tricky scenarios to the above-described problems because of the potential races inherent in returning without waiting for I/O completion. The interplay of AIO-DIO writes and truncate was a particular worry as it could lead to corruption of file data; for example, blocks could get deallocated and reallocated to a new file while an AIO-DIO write to the file was still in progress. To avoid this, AIO-DIO had to return with `i_alloc_sem` held, and only release it as part of I/O completion post-processing. Notice that this also had implications for AIO cancellation.

File size updates for AIO-DIO file extends could expose unwritten blocks if they happened before I/O completed asynchronously. The case involving fallback to buffered I/O was particularly non-trivial if a single request spanned allocated and sparse regions of a file. Specifically, part of the I/O could have been initiated via DIO then continued asynchronously, while the fallback to buffered I/O occurred and signaled I/O completion to the application. The application may thus have reused its I/O buffer, overwriting it with other data and potentially causing file data corruption if writeout to disk had still been pending.

It might appear that some of these problems

could be avoided if I/O schedulers guaranteed the ordering of I/O requests issued to the same disk block. However, this isn't a simple proposition in the current architecture, especially in generalizing the design to all possible cases, including network block devices. The use of I/O barriers would be necessary and the costs may not be justified for these special-case situations.

Instead, a pragmatic approach was taken in order to address this based on the assumptions that true asynchronous behaviour was really meaningful in practice, mainly when performing I/O to already-allocated file blocks. For example, databases typically preallocate files at the time of creation, so that AIO writes during normal operation and in performance-critical paths do not extend the file or encounter sparse regions. Thus, for the sake of correctness, synchronous behaviour may be tolerable for AIO writes involving sparse regions or file extends. This compromise simplified the handling of the scenarios described earlier. AIO-DIO file extends now wait for I/O to complete and update the file size. AIO-DIO writes spanning allocated and sparse regions now wait for previously- issued DIO for that request to complete before falling back to buffered I/O.

5 Concurrent I/O with synchronized write guarantees

An application opts for synchronized writes (by using the `O_SYNC` option on file open) when the I/O must be committed to disk before the write request completes. In the case of DIO, writes directly go to disk anyway. For buffered I/O, data is first copied into the page cache and later written out to disk; if synchronized I/O is specified then the request returns only after the writeout is complete.

An application might also choose to synchronize previously-issued writes to disk by invoking fsync(), which writes back data from the page cache to disk and waits for writeout to complete before returning.

5.1 Concurrent DIO writes

DIO writes formerly held the inode semaphore in exclusive mode until write completion. This helped ensure atomicity of DIO writes and protected against potential file data corruption races with truncate. However, it also meant that multiple threads or processes submitting parallel DIOs to different parts of the same file effectively became serialized synchronously. If the same behaviour were extended to AIO (i.e., having the `i_sem` held through I/O completion for AIO-DIO writes), it would significantly degrade throughput of streaming AIO writes as subsequent write submissions would block until completion of the previous request.

With the fixes described in the previous section, such synchronous serialization is avoidable without loss of correctness, as the inode semaphore needs to be held only when looking up the blocks to write, and not while actual I/O is in progress on the data blocks. This could allow concurrent DIO writes on different parts of a file to proceed simultaneously, and efficient throughputs for streaming AIO-DIO writes.

5.2 Concurrent `O_SYNC` buffered writes

In the original writeback design in the Linux VFS, per-address space lists were maintained for dirty pages and pages under writeback for a given file. Synchronized write was implemented by traversing these lists to issue writeouts for the dirty pages and waiting for writeback to complete on the pages on the writeback list. The inode semaphore had to be held all through to avoid possibilities of livelocking on these lists as further writes streamed into the same file. While this helped maintain atomicity

of writes, it meant that parallel O_SYNC writes to different parts of the file were effectively serialized synchronously. Further, dependence on i_sem-protected state in the address space lists across I/O waits made it difficult to retry-enable this code path for AIO support.

In order to allow concurrent O_SYNC writes to be active on a file, the range of pages to be written back and waited on could instead be obtained directly through a radix-tree lookup for the range of offsets in the file that was being written out by the request [4]. This would avoid traversal of the page lists and hence the need to hold i_sem across the I/O waits. Such an approach would also make it possible to complete O_SYNC writes as a sequence of non-blocking retry iterations across the range of bytes in a given request.

5.3 Data-integrity guarantees

Background writeout threads cannot block on the inode semaphore like O_SYNC/fsync writers. Hence, with the per-address space lists writeback model, some juggling involving movement across multiple lists was required to avoid livelocks. The implementation had to make sure that pages which by chance got picked up for processing by background writeouts didn't slip from consideration when waiting for writeback to complete for a synchronized write request. The latter would be particularly relevant for ensuring synchronized-write guarantees that impacted data integrity for applications. However, as Daniel McNeil's analysis would indicate [5], getting this right required the writeback code to write and wait upon I/O and dirty pages which were initiated by other processes, and that turned out to be fairly tricky.

One solution that was explored was per-address space serialization of writeback to ensure exclusivity to synchronous writers and shared mode for background writers. It involved navigating issues with busy-waits in background writers and the code was beginning to get complicated and potentially fragile.

This was one of the problems that finally prompted Andrew Morton to change the entire VFS writeback code to use radix-tree walks instead of the per-address space pagelists. The main advantage was that avoiding the need for movement across lists during state changes (e.g., when re-dirtying a page if its buffers were locked for I/O by another process) reduced the chances of pages getting missed from consideration without the added serialization of entire writebacks.

6 Tagged radix-tree based writeback

For the radix-tree walk writeback design to perform as well as the address space lists-based approach, an efficient way to get to the pages of interest in the radix trees is required. This is especially so when there are many pages in the pagecache but only a few are dirty or under writeback. Andrew Morton solved this problem by implementing tagged radix-tree lookup support to enable lookup of dirty or writeback pages in O(log64(n)) time [6].

This was achieved by adding tag bits for each slot to each radix-tree node. If a node is tagged, then the corresponding slots on all the nodes above it in the tree are tagged. Thus, to search for a particular tag, one would keep going down sub-trees under slots which have the tag bit set until the tagged leaf nodes are accessed. A tagged gang lookup function is used for in-order searches for dirty or writeback pages within a specified range. These lookups are used to replace the per-address-space page lists altogether.

To synchronize writes to disk, a tagged radix-tree gang lookup of dirty pages in the byte-range corresponding to the write request is performed and the resulting pages are written out. Next, pages under writeback in the byte-range are obtained through a tagged radix-tree gang lookup of writeback pages, and we wait for writeback to complete on these pages (without having to hold the inode semaphore across the waits). Observe how this logic lends itself to be broken up into a series of non-blocking retry iterations proceeding in-order through the range.

The same logic can also be used for a whole file sync, by specifying a byte-range that spans the entire file.

Background writers also use tagged radix-tree gang lookups of dirty pages. Instead of always scanning a file from its first dirty page, the index where the last batch of writeout terminated is tracked so the next batch of writeouts can be started after that point.

7 Streaming AIO writes

The tagged radix-tree walk writeback approach greatly simplifies the design of AIO support for synchronized writes, as mentioned in the previous section,

7.1 Basic retry pattern for synchronized AIO writes

The retry-based design for buffered AIO `O_SYNC` writes works by converting each blocking wait for writeback completion of a page into a *retry exit*. The conversion point queues an asynchronous notification callback and returns to the caller of the filesystem's AIO write operation the number of bytes for which writeback has completed so far without blocking. Then, when writeback completes for that page, the callback kicks off a retry continuation in task context which invokes the same AIO write operation again using the caller's address space, but this time with arguments modified to reflect the remaining part of the write request.

As writeouts for the range would have already been issued the first time before the loop to wait for writeback completion, the implementation takes care not to re-dirty pages or re-issue writeouts during subsequent retries of AIO write. Instead, when the code detects that it is being called in a retry context, it simply falls through directly to the step involving wait-on-writeback for the remaining range as specified by the modified arguments.

7.2 Filtered waitqueues to avoid retry storms with hashed wait queues

Code that is in a retry-exit path (i.e., the return path following a blocking point where a retry is queued) should in general take care not to call routines that could wakeup the newly-queued retry.

One thing that we had to watch for was calls to `unlock_page()` in the retry-exit path. This could cause a redundant wakeup if an async wait-on-page writeback was just queued for that page. The redundant wakeup would arise if the kernel used the same waitqueue on unlock as well as writeback completion for a page, with the expectation that the waiter would check for the condition it was waiting for and go back to sleep if it hadn't occurred. In the AIO case, however, a wakeup of the newly-queued callback in the same code path could potentially trigger a retry storm, as retries kept triggering themselves over and over again for the wrong condition.

The interplay of `unlock_page()` and `wait_on_page_writeback()` with hashed waitqueues can get quite tricky for retries. For example, consider what happens when the following sequence in retryable code is executed at the same time for 2 pages, *px*

and *py*, which happen to hash to the same waitqueue (Table 1).

```
lock_page(p)
check condition and process
unlock_page(p)
if (wait_on_page_writeback_wq(p)
  == -EIOCBQUEUED)
    return bytes_done
```

The above code could keep cycling between spurious retries on *px* and *py* until I/O is done, wasting precious CPU time!

If we can ensure specificity of the wakeup with hashed waitqueues then this problem can be avoided. William Lee Irwin's implementation of filtered wakeup support in the recent Linux 2.6 kernels [7] achieves just that. The wakeup routine specifies a key to match before invoking the wakeup function for an entry in the waitqueue, thereby limiting wakeups to those entries which have a matching key. For page waitqueues, the key is computed as a function of the page and the condition (unlock or writeback completion) for the wakeup.

7.3 Streaming AIO write microbenchmark comparisons

The following graph compares aio-stress throughputs for streaming random buffered I/O `O_SYNC` writes, with and without the previously-described changes. The comparison was performed on the same setup used for the streaming AIO read results discussed earlier. The graph summarizes how the results varied across individual request sizes of 4KB to 64KB, where I/O was targeted to a single file of size 1GB and the depth of iocbs outstanding at a time was 64KB. A third run was performed to determine how the results compared with equivalent runs using AIO-DIO.

With the changes applied, the results showed an approximate 2x improvement across all block sizes, bringing throughputs to levels that match the corresponding results using AIO-DIO.

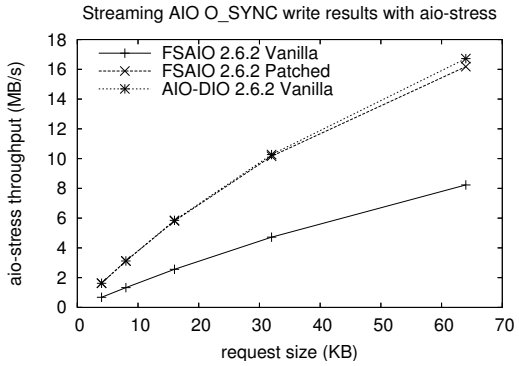

Figure 2: Comparisons of streaming random AIO write throughputs.

8 AIO performance analysis for database workloads

Large database systems leveraging AIO can show marked performance improvements compared to those systems that use synchronous I/O alone. We use IBM® DB2® Universal Database™ V8 running an online transaction processing (OLTP) workload to illustrate the performance improvement of AIO on raw devices and on filesystems.

8.1 DB2 page cleaners

A DB2 page cleaner is a process responsible for flushing dirty buffer pool pages to disk. It simulates AIO by executing asynchronously with respect to the agent processes. The number of page cleaners and their behavior can be tuned according to the demands of the system. The agents, freed from cleaning pages themselves, can dedicate their resources (e.g., processor cycles) towards processing transactions, thereby improving throughput.

```
CPU1                                    CPU2
lock_page(px)
...
unlock_page(px)
                                        lock_page(py)
wait_on_page_writeback_wq(px)           ...
                                        unlock_page(py) -> wakes up p1
                                                           triggering
                <-----------------------------------       a retry
lock_page(px)                           wait_on_page_writeback_wq(py)
...
unlock_page(py)  ---- wakes up py --- causes retry ---->
```

Table 1: Retry storm livelock with redundant wakeups on hashed wait queues

8.2 AIO performance analysis for raw devices

Two experiments were conducted to measure the performance benefits of AIO on raw devices for an update-intensive OLTP database workload. The workload used was derived from a TPC[8] benchmark, but is in no way comparable to any TPC results. For the first experiment, the database was configured with one page cleaner using the native Linux AIO interface. For the second experiment, the database was configured with 55 page cleaners all using the synchronous I/O interface. These experiments showed that a database, properly configured in terms of the number of page cleaners with AIO, can out-perform a properly configured database using synchronous I/O page cleaning.

For both experiments, the system configuration consisted of DB2 V8 running on a 2-way AMD Opteron system with Linux 2.6.1 installed. The disk subsystem consisted of two FAStT 700 storage servers, each with eight disk enclosures. The disks were configured as RAID-0 arrays with a stripe size of 256KB.

Table 2 shows the relative database performance with and without AIO. Higher numbers are better. The results show that the database performed 9% better when configured with one page cleaner using AIO, than when it was configured with 55 page cleaners using synchronous I/O.

Configuration	Relative Throughput
1 page cleaner with AIO	133
55 page cleaners without AIO	122

Table 2: Database performance with and without AIO.

Analyzing the I/O write patterns (see Table 3), we see that one page cleaner using AIO was sufficient to keep the buffer pools clean under a very heavy load, but that 55 page cleaners using synchronous I/O were not, as indicated by the 30% agent writes. This data suggests that more page cleaners should have been configured to improve the performance of the case with synchronous I/O. However, additional page cleaners consumed more memory, requiring a reduction in bufferpool size and thereby decreasing throughput. For the test configuration, 55 cleaners was the optimal number before memory constraints arose.

8.3 AIO performance analysis for filesystems

This section examines the performance improvements of AIO when used in conjunction with filesystems. This experiment was per-

Configuration	Page cleaner writes (%)	Agent writes (%)
1 page cleaner with AIO	100	0
55 page cleaners without AIO	70	30

Table 3: DB2 write patterns for raw device configurations.

formed using the same OLTP benchmark as in the previous section.

The test system consisted of two 1GHz AMD Opteron processors, 4GB of RAM and two QLogic 2310 FC controllers. Attached to the server was a single FAStT900 storage server and two disk enclosures with a total of 28 15K RPM 18GB drives. The Linux kernel used for the examination was 2.6.0+mm1, which includes the AIO filesystem support patches [9] discussed in this paper.

The database tables were spread across multiple ext2 filesystem partitions. Database logs were stored on a single raw partition.

Three separate tests were performed, utilizing different I/O methods for the database page cleaners.

Test 1. Synchronous (Buffered) I/O.

Test 2. Asynchronous (Buffered) I/O.

Test 3. Direct I/O.

The results are shown in Table 4 as relative commercial processing scores using synchronous I/O as the baseline (i.e., higher is better).

Looking at the efficiency of the page cleaners (see Table 5), we see that the use of AIO is more successful in keeping the buffer pools clean. In the synchronous I/O and DIO cases, the agents needed to spend more time cleaning

Configuration	Commercial Processing Scores
Synchronous I/O	100
AIO (Buffered)	113.7
DIO	111.9

Table 4: Database performance on filesystems with and without AIO.

buffer pool pages, resulting in less time processing transactions.

Configuration	Page cleaner writes (%)	Agent writes (%)
Synchronous I/O	37	63
AIO (buffered)	100	0
DIO	49	51

Table 5: DB2 write patterns for filesystem configurations.

8.4 Optimizing AIO for database workloads

Databases typically use AIO for streaming batches of random, synchronized write requests to disk (where the writes are directed to preallocated disk blocks). This has been found to improve the performance of OLTP workloads, as it helps bring down the number of dedicated threads or processes needed for flushing updated pages, and results in reduced memory footprint and better CPU utilization and scaling.

The size of individual write requests is determined by the page size used by the database. For example, a DB2 UDB installation might use a database page size of 8KB.

As observed in previous sections, the use of AIO helps reduce the number of database page cleaner processes required to keep the bufferpool clean. To keep the disk queues maximally utilized and limit contention, it may be preferable to have requests to a given disk streamed out from a single page cleaner. Typically a set of of disks could be serviced by each page

cleaner if and when multiple page cleaners need to be used.

Databases might also use AIO for reads, for example, for prefetching data to service queries. This usually helps improve the performance of decision support workloads. The I/O pattern generated in these cases is that of streaming batches of large AIO reads, with sizes typically determined by the file allocation extent size used by the database (e.g., a DB2 installation might use a database extent size of 256KB). For installations using buffered AIO reads, tuning the readahead setting for the corresponding devices to be more than the extent size would help improve performance of streaming AIO reads (recall the discussion in Section 3.5).

9 Addressing AIO workloads involving both disk and communications I/O

Certain applications need to handle both disk-based AIO and communications I/O. For communications I/O, the epoll interface—which provides support for efficient scalable event polling in Linux 2.6—could be used as appropriate, possibly in conjunction with `O_NONBLOCK` socket I/O. Disk-based AIO on the other hand, uses the native AIO API `io_getevents` for completion notification. This makes it difficult to combine both types of I/O processing within a single event loop, even when such a model is a natural way to program the application, as in implementations of the application on other operating systems.

How do we address this issue? One option is to extend epoll to enable it to poll for notification of AIO completion events, so that AIO completion status can then be reaped in a non-blocking manner. This involves mixing both epoll and AIO API programming models, which is not ideal.

9.1 AIO poll interface

Another alternative is to add support for polling an event on a given file descriptor through the AIO interfaces. This function, referred to as AIO poll, can be issued through `io_submit()` just like other AIO operations, and specifies the file descriptor and the eventset to wait for. When the event occurs, notification is reported through `io_getevents()`.

The retry-based design of AIO poll works by converting the blocking wait for the event into a *retry exit*.

The generic synchronous polling code fits nicely into the AIO retry design, so most of the original polling code can be used unchanged. The private data area of the iocb can be used to hold polling-specific data structures, and a few special cases can be added to the generic polling entry points. This allows the AIO poll case to proceed without additional memory allocations.

9.2 AIO operations for communications I/O

A third option is to add support for AIO operations for communications I/O. For example, AIO support for pipes has been implemented by converting the blocking wait for I/O on pipes to a *retry exit*. The generic pipe code was also structured such that conversion to AIO retries was quite simple, the only significant change was using the current `io_wait` context instead of a locally defined waitqueue, and returning early if no data was available.

However, AIO pipe testing did show significantly more context switches then the 2.4 AIO pipe implementation, and this was coupled with much lower performance. The AIO core functions were relying on workqueues to do most of the retries, and this resulted in constant

switching between the workqueue threads and user processes.

The solution was to change the AIO core to do retries in `io_submit()` and in `io_getevents()`. This allowed the process to do some of its own work while it is scheduled in. Also, retries were switched to a delayed workqueue, so that bursts of retries would trigger fewer context switches.

While delayed wakeups helped with pipe workloads, it also caused I/O stalls in filesystem AIO workloads. This was because a delayed wakeup was being used even when a user process was waiting in `io_getevents()`. When user processes are actively waiting for events, it proved best to trigger the worker thread immediately.

General AIO support for network operations has been considered but not implemented so far because of lack of supporting study that predicts a significant benefit over what epoll and non-blocking I/O can provide, except for the scope for enabling potential zero-copy implementations. This is a potential area for future research.

10 Conclusions

Our experience over the last year with AIO development, stabilization and performance improvements brought us to design and implementation issues that went far beyond the initial concern of converting key I/O blocking points to be asynchronous.

AIO uncovered scenarios and I/O patterns that were unlikely or less significant with synchronous I/O alone. For example, the issues we discussed around streaming AIO performance with readahead and concurrent synchronized writes, as well as DIO vs buffered I/O complexities in the presence of AIO. In retrospect, this was the hardest part of supporting AIO—modifiying code that was originally designed only for synchronous I/O.

Interestingly, this also meant that AIO appeared to magnify some problems early. For example, issues with hashed waitqueues that led to the filtered wakeup patches, and readahead window collapses with large random reads which precipitated improvements to the readahead code from Ramachandra Pai. Ultimately, many of the core improvements that helped AIO have had positive benefits in allowing improved concurrency for some of the synchronous I/O paths.

In terms of benchmarking and optimizing Linux AIO performance, there is room for more exhaustive work. Requirements for AIO fsync support are currently under consideration. There is also a need for more widely used AIO applications, especially those that take advantaged of AIO support for buffered I/O or bring out additional requirements like network I/O beyond epoll or AIO poll. Finally, investigations into API changes to help enable more efficient POSIX AIO implementations based on kernel AIO support may be a worthwhile endeavor.

11 Acknowledgements

We would like to thank the many people on the `linux-aio@kvack.org` and `linux-kernel@vger.kernel.org` mailing lists who provided us with valuable comments and suggestions during our development efforts.

We would especially like to acknowledge the important contributions of Andrew Morton, Daniel McNeil, Badari Pulavarty, Stephen Tweedie, and William Lee Irwin towards several pieces of work discussed in this paper.

This paper and the work it describes wouldn't have been possible without the efforts of Janet Morgan in many different ways, starting from review, test and debugging feedback to joining the midnight oil camp to help with modifications and improvements to the text during the final stages of the paper.

We also thank Brian Twitchell, Steve Pratt, Gerrit Huizenga, Wayne Young, and John Lumby from IBM for their help and discussions along the way.

This work was a part of the Linux Scalability Effort (LSE) on SourceForge, and further information about Linux 2.6 AIO is available at the LSE AIO web page [10]. All the external AIO patches including AIO support for buffered filesystem I/O, AIO poll and AIO support for pipes are available at [9].

12 Legal Statement

13 Disclaimer

The benchmarks discussed in this paper were conducted for research purposes only, under laboratory conditions. Results will not be realized in all computing environments.

References

[1] Suparna Bhattacharya, Badari Pulavarthy, Steven Pratt, and Janet Morgan. Asynchronous i/o support for linux 2.5. In *Proceedings of the Linux Symposium*. Linux Symposium, Ottawa, July 2003. `http://archive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Pulavarty-OLS2003.pdf`.

[2] Chris Mason. aio-stress microbenchmark. `ftp://ftp.suse.com/pub/people/mason/utils/aio-stress.c`.

[3] Stephen C. Tweedie. Posting on dio races in 2.4. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=105597840711609&w=2`.

[4] Andrew Morton. O_sync speedup patch. `http://www.kernel.org/pub/linux/kernel/people/akpm/patches/2.6/2.6.0/2.6.0-mm1/broken-out/O_SYNC-speedup-2.patch`.

[5] Daniel McNeil. Posting on synchronized writeback races. `http://marc.theaimsgroup.com/?l=linux-aio&m=107671729611002&w=2`.

[6] Andrew Morton. Posting on in-order tagged radix tree walk based vfs writeback. `http://marc.theaimsgroup.com/?l=bk-commits-head&m=108184544016117&w=2`.

[7] William Lee Irwin. Filtered wakeup patch. `http://marc.theaimsgroup.com/?l=bk-commits-head&m=108459430513660&w=2`.

[8] Transaction processing performance council. http://www.tpc.org.

[9] Suparna Bhattacharya (with contributions from Andrew Morton & Chris Mason). Additional 2.6 Linux Kernel Asynchronous I/O patches. http://www.kernel.org/pub/linux/kernel/people/suparna/aio.

[10] LSE team. Kernel Asynchronous I/O (AIO) Support for Linux. http://lse.sf.net/io/aio.html.

Proceedings of the Linux Symposium

Volume One

July 21st–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*