Getting Priorities Straight: Improving Linux Support for Database I/O

Christoffer Hall, Philippe Bonnet

University of Copenhagen
Universitetsparken 1
2100 Copenhagen
Denmark
{hall,bonnet}@diku.dk

Abstract

The Linux 2.6 kernel supports asynchronous I/O as a result of propositions from the database industry. This is a positive evolution but is it a panacea? In the context of the Badger project, a collaboration between MySQL AB and University of Copenhagen, we evaluate how MySQL/InnoDB can best take advantage of Linux asynchronous I/O and how Linux can help MySQL/InnoDB best take advantage of the underlying I/O bandwidth. This is a crucial problem for the increasing number of MySQL servers deployed for very large database applications. In this paper, we first show that the conservative I/O submission policy used by InnoDB (as well as Oracle 9.2) leads to an under-utilization of the available I/O bandwidth. We then show that introducing prioritized asynchronous I/O in Linux will allow MySQL/InnoDB and the other Linux databases to fully utilize the available I/O bandwith using a more aggressive I/O submission policy.

1 Introduction

Established database vendors are promoting Linux as a platform of choice for commodity servers. In addition to crafting marketing messages, they are involved in the evolution of the Linux kernel [15, 10]. Oracle and IBM in particular have argued that kernel support for asynchronous I/O[1] was critical for database performance [20]. Their efforts have resulted in the adoption of asynchronous I/O in Linux 2.6. But is that good enough? Can the Linux kernel be further enhanced to support database I/O?

The collaboration between the database industry and the Linux community is a positive development. We believe that this collaboration should be extended to the research community. Indeed, the emergence of commercially viable open source database systems empowers the data management research community to impact the design and implementation of actual products.

In the context of the Badger project, a collaboration between University of Copenhagen and MySQL AB, we study how MySQL equipped with the InnoDB[2] storage manager can best take advantage of the underlying I/O bandwidth. As all other database vendors, MySQL and InnoDB regularly interact with clients who want to get the best performance out of their I/O devices. More generally, the widening gap between CPU speed and I/O bandwith defines the need to utilize the existing bandwith as efficiently as possible [13]. Ideally, it is the hardware configuration that limits I/O performance, but *do the software layers, MySQL/InnoDB (or as a comparison point Oracle) on top of Linux, utilize the underlying I/O devices as efficiently as possible*? This is the question that underlies this study.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

[1] According to Open Group definition: An asynchronous I/O operation is an I/O operation that does not of itself cause the thread requesting the I/O to be blocked from further use of the processor [16].

[2] MySQL can accomodate various storage managers. In this paper we focus on InnoDB, whose structure is similar to Oracle's storage manager.

1.1 Throughput/Latency Trade-off

To efficiently support database I/O, the OS kernel must efficiently support concurrent I/O requests on multiple files[3]. This is why asynchronous I/O are very attractive for database servers. They allow to:

1. *Accumulate reads or write requests so that the I/O subsystem can optimize performance.* The file system can group and reorder I/O requests in order (i) to favor sequential access to disk and (ii) to favor larger requests. This benefits all I/O issued by the database server.

2. *Overlap I/O and CPU work.* This benefits background operations such as data prefetching or preflushing (by the lazy writer).

3. *Parallelize reads and write requests to multiple files.* This benefits concurrent requests on log and data files.

Intuitively, asynchronous I/O are most effective if the number of outstanding request is constant and sufficiently high to allow the file system to schedule them efficiently. We verify Linux ability to accumulate and schedule asynchronous I/O in Section 2.
The database storage manager is submitting I/O requests both synchronously (physical reads and log writes that are blocking the query thread) and asynchronously (prefetching and preflushing that are non-blocking). For synchronous requests, latency is of the essence. For asynchronous requests, throughput is key. There is thus a trade-off: The database should submit as many asynchronous requests as necessary to increase throughput while making sure the file system is ready to process synchronous requests whenever they are submitted.
The only way for the database to ensure that synchronous requests are processed efficiently is to keep the number of outstanding asynchronous requests low at all time[4]. This way, the file system will not be too busy whenever a synchronous request is submitted. InnoDB for example limits the number of asynchronous requests submitted when prefetching or preflushing. Prefetching is done 1MB at-a-time, and preflushing is done at various intervals depending on memory pressure and I/O activity (up to once a second if I/O activity is low).
Such a *conservative policy* is currently the best way for databases to handle the throughput/latency trade-off. Ideally though, it is not the database that should handle this trade-off but the file system. Indeed the database is not actually scheduling IO requests, it is the role of the file system[5]. Instead of controlling the latency/throughput trade-off by limiting the number of submitted I/O requests, the database should rely on the I/O subsystem to schedule latency critical I/Os in a timely manner even if there are many outstanding requests. In other words, the I/O subsystems should distinguish between synchronous and asynchronous requests, thus providing the database system with a way to schedule latency critical requests without sacrificing throughput.
On some mainframes (e.g., DB2 on z/OS [19]), the operating system supports I/O priorities. The file system schedules I/O requests based on their priority thus trading throughput for latency in case a high priority request needs to be scheduled while a large number of consecutive low priority requests are being queued.
If the database system associates a high priority to synchronous requests and a low priority to asynchronous requests, it can implement an *aggressive I/O policy* that maintains a high number of outstanding asynchronous requests and let the file system handle the latency/throughput trade-off.

1.2 Contribution

Following the downsizing trend from mainframes to commodity servers [14], we propose to introduce prioritized I/O in Linux so that MySQL/InnoDB (and other Linux databases) can implement aggressive I/O policies. More specifically, our contribution is the following:

- We evaluate the capacity of Linux to accumulate asynchronous I/O requests and schedule them efficiently.

- We show that MySQL/InnoDB does not utilize the full potential of Linux asynchronous I/O (and that neither does Oracle) due to its conservative I/O policy. Note that we extended InnoDB to support native Linux asynchronous I/O.

- We introduce prioritized I/O in Linux so that the file system can control the latency/throughput trade-off. We detail our design and implementation within the Linux kernel and illustrate its potential benefit using a version of InnoDB modified to associate priorities to the I/O it submits.

- We summarize the lessons we learnt and discuss the design of an aggressive I/O policy.

[3]In addition, the file system should provide backup capabilities (specially for the data and index files).

[4]We focus on the I/O generated by the database and we consider that the database tuner has made sure that the amount of reads and writes is not higher than necessary [22]

[5]In this study we consider that the file I/O subsystem is responsible for scheduling I/O requests. Studying the utilization of I/O bandwidth on servers connected to a SAN is a topic for future work and a key challenge for the data management research community.

2 Linux Asynchronous I/O

Before we focus on the performance of Linux asynchronous I/O, let us describe the internals of the Linux kernel that are relevant for our study.

2.1 Linux Kernel Internals

In the rest of the paper, we adopt a classic representation of the software layers on top of I/O devices. The database server relies on OS kernel services to access the underlying I/O devices. Kernel services are typically organized in three layers [23]: At the bottom, device drivers abstract the actual communication with the hardware devices. On top of these drivers, the I/O subsystem is responsible for the execution of I/O requests. The top layer consists of the file services including layout and metadata management as well as caching.

Let us zoom in on the Linux file system cache. This cache stores pages that have previously been read from files and it stores pages that have just been written. The *fsync* system call flushes all buffered write requests to disk. In addition, a lazy writer (called `pdflush`) forces dirty pages to disk regularly or as a result of memory pressure. When a sequential pattern is detected among read requests, a read ahead mechanism is activated to prefetch pages (the number of prefetched pages depends on memory pressure). Because the file system cache is in kernel space, buffered pages are copied between kernel and user space whenever they are read or written.

Linux supports *direct* I/O that bypass the file system cache when reading or writing to a file. Direct I/O requests manipulate pages allocated in user space. Whether read and write operations are buffered or direct is specified when opening a file (e.g., using the `O_DIRECT` parameter for direct I/O). Note that I/O devices can also be opened as *raw* devices, in which case the application directly interacts with the I/O subsystem[6].

As of version 2.6, the Linux I/O subsystem fully relies on asynchronous I/O. Synchronous I/O are implemented on top of asynchronous I/O. When an I/O request is submitted, it is associated to a completion queue. A worker thread then progresses through an asynchronous state machine, and ends up sending a page request to the disk scheduler. When the page request completes, an interrupt is raised (within the I/O device driver). In the case of direct I/O, the I/O completion is notified from the interrupt context. In the case of buffered read, the I/O completion is notified after the page is added to the file system cache.

A request to the disk scheduler logically consist of a number of contiguous sectors and a flag that states if the blocks should be read or written. These requests are typically sorted in order to minimize disk seeks. The scheduler will also merge smaller requests into a larger one to exploit disk throughput and to minimize the number of DMA transfers. Note that Linux uses one disk scheduler per I/O device. As a consequence, I/O requests submitted to different disks are actually treated in parallel.

Linux 2.6 implements a deadline-based scheduler[7]. When a request enters the disk scheduler it is assigned a deadline using a fixed time offset. Different time offsets are used for reads (1 second) and writes (5 seconds). This deadline describes a point in time by which the request should be submitted to disk. This is done to avoid starvation (requests that are never submitted if the scheduler only focuses on minimizing seeks). The disk scheduler arranges the requests it receives both (a) in two Red-Black tree (one for read, the other for write) sorted by timestamp (or deadline) – that sort requests on the first sector they access –, and (b) in two lists (one for read, the other for write) sorted by timestamp (or deadline) associated to each request. Because the deadlines are simply defined using time offset, those sorted lists are simply FIFO queues. A dispatch queue is used to store the requests scheduled for submission.

The device driver always accesses the dispatch queue first. If there are requests on it they will be sent to the I/O device. If there are no requests on the dispatch queue, then the driver will look to the deadline list. If the earliest deadline is reached then the associated requests are moved to the dispatch queue. If no deadline is reached, then requests are moved to the dispatch queue from the sector-sorted Red-Black trees.

2.2 Performance Characteristics

We ran a set of experiments to find out (a) whether Linux asynchronous I/O did a good job at utilizing the capacity of the underlying I/O device in terms of latency and throughput, and (b) what kind of overhead they incurred in terms of CPU usage. The results of these experiments are presented in [11][8]. In

[6] By default, Linux supports buffered access to raw devices. As of Linux 2.6, direct access to raw devices is possible using the `O_DIRECT` parameter. Previously, a specific raw driver needed to be used to avoid buffering accesses to a raw device. In our experiments we used direct access to raw devices using the deadline scheduler.

[7] Linux 2.6 provides several disk schedulers. The default scheduler is called *anticipatory*, as it waits for some predefined I/O patterns, e.g., when a page is read the disk scheduler waits for a contiguous page before scheduling other requests. Those patterns do not fit the needs of database systems. There is also a *no-op* disk scheduler that basically serializes the incoming requests and hands over the responsability of scheduling to an underlying RAID or SAN controller.

[8] Note that we did not conduct scalability experiments: our experiments are run with a single thread issuing requests on a single file. IBM Linux Technology Center is conducting

this section we focus on the throughput and latency of random asynchronous requests.
We run our experiments using a simple tool that submits I/O requests, so that the number of outstanding requests remains constant over the duration of each experiment. Requests are issued against a 10 GB file. Each request manipulates 16 KB (i.e., the size of an InnoDB page). Sequential requests scan the file, while random requests cover the whole file randomly. We measure latency for each request and throughput as the ratio of the total amount of data transferred (10 GB) over the total time for all requests. We remounted the file system (reiserfs) between runs to enforce a cold cache. Our benchmarking tool can be used by system administrators to gather key characteristics of the Linux asynchronous I/O on their own installation [9].
The I/O devices in our experiments are just a bunch of disks (IBM Ultrastar 36LZX), directly connected to a dual 1 GHz Pentium III server via a SCSI bus on two different channels (Dual channel Adaptec AHA-3960D controller). The disks are configured with read ahead and write back enabled. The server has 1 GB of RAM. This simple hardware configuration (no RAID, no SAN) allows us to reduce the number of parameters as we focus on the Linux kernel services. Running hdparm, we measured a sustained data rate of 34,5 MB/s which matches the disk specification [1]. This is the sequential throughput we can hope for.

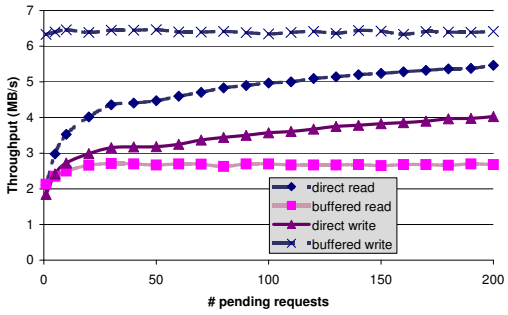


Figure 1: Throughput of Random Requests (buffered vs. direct I/O)

We expect that increasing the number of outstanding requests submitted by the benchmarking tool will improve throughput and increase latency. Figure 1 shows the throughput of random requests for buffered and direct I/O as we increase the number of outstanding requests. Throughput remains constant at around 2,7 MB/sec for buffered reads and at around 6,5 MB/sec for buffered writes (with a single fsync[10]). The throughput of direct reads increases quickly with the number of outstanding requests: from 2 MB/sec for 1 outstanding request to 4,5 MB/sec for 64 outstanding requests. In comparison, the throughput of direct writes increases slowly: it reaches 4,1 MB/sec for 200 outstanding requests. As expected, the increased throughput is due to the ordering of requests in the disk scheduler. The poor performance of buffered read is due to a bug in the file system that serializes the submission of I/O requests.

scalability experiments. Preliminary results were presented in [8, 6].

[9] http://www.distlab.dk/badger/

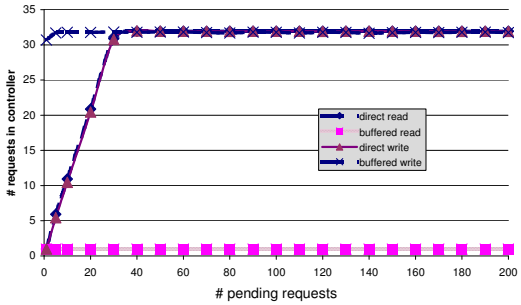


Figure 2: Number of requests in SCSI controller as a function of the number of pending requests

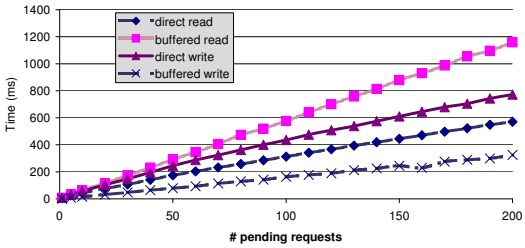


Figure 3: Latency of random requests (buffered vs. direct I/O)

Figure 2 traces the number of requests actually placed in the SCSI controller as a function of the number of outstanding requests (submitted by the benchmarking tool). The graph shows that the SCSI controller can handle up to 32 requests and no more. The lazy writer does a good job at submitting write requests (for buffered writes). For buffered reads however, the file system serializes the submission of requests to the SCSI controller. This explains the advantage of direct reads in terms of throughput. Indeed for direct I/O the number of requests in the SCSI controller rapidly increases to the maximum as the number of I/O requests submitted by the benchmarking tool increases.
Figure 3 traces the average latency of random re-

[10] We dicuss the impact of fsync on performance in our technical report

quests as a function of the number of outstanding requests. The latency increases linearly with the number of outstanding requests. As exected, the latency of buffered writes is significantly lower than the latency of buffered reads. The latency of direct requests lies in between. For direct requests, average latency becomes noticeably high (above 0,5 second) when the number of outstanding requests reaches 128.
Because of space limitation, we do not show all the graphs we obtained (see [11] for details). Here is a summary of our results:

- Sequential requests: We could expect that increasing the number of outstanding sequential requests would lead the scheduler to merge requests into large blocks thus increasing throughput. However, we observe that throughput remains constant for buffered and direct requests (at around 30MB/sec for writes and 33 MB/sec for reads). In general throughput is slightly higher for sequential reads compared to sequential writes because the disk implements a form of read ahead that benefits read requests [1]. The latency of sequential requests is lower compared to the latency of random requests but has a similar pattern. A latency of 0,5 seconds is reached when the number of outstanding requests is higher than 1024. Those results show that the number of outstanding requests is irrelevant for sequential requests. The key parameter is the rate at which requests are submitted.

- CPU utilization: We observed that CPU usage is lower than 3 msec/MB for direct I/O requests (this is comparable to the CPU overhead measured by Chung et al. for direct I/O on Windows 2000 [12]).

- Raw I/O: Direct I/O and raw I/O exhibit similar characteristics in terms of throughput and latency. The only minor difference we observed was a slightly higher CPU overhead for direct writes compared to raw writes.

Our experiments focused on the capacity of Linux asynchronous I/O to accumulate I/O requests and schedule them efficiently. We can draw the following conclusions:

1. *Linux asynchronous I/O implementation is efficient.* For sequential requests, the throughput we obtain (33 MB/s) is close to the maximum (34,5 MB/s). For random requests, we obtain a throughput of up to 320 pages/s (5 MB/s). This is much higher than the 200 pages/s we would obtain with the documented average seek time of 4,9 msec. The good performance for random requests is due to the ordering of requests in the disk scheduler that results in a reduction of the seek distance. There is however a clear tradeoff between throughput and latency for random requests. Issuing a large number of requests improves throughput but hurts latency.

2. *Linux databases should use direct I/O.* Compared to buffered I/O, direct I/O provide better performance for random requests (provided enough requests are submitted), better CPU utilization, and equivalent scan performance. Direct I/O provides equivalent performance compared to a raw device while providing a file system abstraction.

3. *The rate of submission is key for sequential operations.* This impacts prefetching (sequential reads) and log writing (sequential writes). Both operations will be most efficient if they succeed in issuing a sustained flow of request submissions.

We have established the potential of Linux asynchronous I/O. In the next Section, we focus on how InnoDB (and Oracle 9.2) utilize the available I/O bandwidth.

3 InnoDB Conservative I/O Policy

The InnoDB storage manager is comparable to Oracle's with its support for multi-read consistency, the separation of redo and undo log, and the utilization of tablespaces as abstraction for data files. In this Section, we describe InnoDB's I/O submission policy and we evaluate how this policy impacts I/O bandwidth utilization.

3.1 InnoDB Internals

Before we focus on how I/O are submitted, let us review the organization of the files on which I/O requests are submitted. InnoDB separates log and data files (temporary files are organized as data files). Log files are managed as a circular structure to which redo log records are appended[11]. Data files are organized in tablespaces. A tablespace consists of one or several operating system files. Each tablespace is structured in segments. Segments are associated to tables. For example, a table with a primary index is stored using a data segment and an index segment. Each segment is organized in extents of 64 pages. Page size is fixed at 16KB.
InnoDB issues following the I/O requests:

- *Sequential writes of log records.* A write request containing log records is submitted by a query thread at commit time, or if the cache that

[11] As Oracle, InnoDB manages the circular log over several files.

stores log records in memory is 50% full. In addition, the background server thread forces log records to disk every second.

- *Random writes of dirty pages.* If there are dirty pages in the buffer pool and if the I/O activity is low then preflusing takes places and InnoDB issues asynchronous write requests to write committed dirty pages to disk. Now, if there is memory pressure InnoDB issues synchronous write requests – possibly dirty pages are stolen to free buffer space.

- *Random reads for physical I/O.* Random reads are submitted by query threads if the page they access is not in the database cache.

- *Sequential reads during prefetching.* The query thread performs prefetching as follows. If it accesses pages with a sequential pattern then it prefetches extents, one at a time. Otherwise, if a query thread accesses more than a tunable number of pages from a same extent, then the whole extent is prefetched. Pages are allocated in the database cache as soon as I/O requests are submitted. A query thread might access a page for which the I/O request has not yet completed. In that case the query thread waits on a latch and is notified when the I/O completes.

InnoDB uses native asynchronous I/O on Windows, while it uses simulated AI/O on Linux and other Unix systems. Simulated I/O rely on dedicated threads (I/O handler threads) that accumulate I/O requests and process them while the query thread is running. The I/O threads merge I/O requests on consecutive pages and submit them in sequence using synchronous I/O.

We modified MySQL/InnoDB v4.1 to utilize Linux asynchronous I/O. Support for Linux asynchronous I/O is modeled on InnoDB's support for Windows asynchronous I/O. That means that asynchronous I/O are submitted from the query threads directly. The completion mechanism in Linux is a bit different than on Windows. On Windows, a query thread uses `WaitForMultipleObjects` to wait for the completion of the I/O requests it has issued. There is no such function in Linux: `get_event` returns with a list of completed events – there is no possibility to filter events within the kernel. We thus introduced the `eventprocessing` thread that filters completion events and signals any thread that is waiting for the completion of a given I/O request. This support for native asynchronous I/O on Linux has been transferred to InnoDB Oy.

3.2 Performance Characteristics

We conducted a set of experiments to establish how efficiently MySQL/InnoDB v4.1 uses asynchronous I/O. We used Oracle 9.2 as a comparison because it has similar characteristics. We configured both systems to use native direct asynchronous I/O.

Prefetching

Using asynchronous I/O, a scan of 800 MB takes 29 seconds as opposed to 33 seconds with simulated asynchronous I/O. This slight improvement is due to a more sustained flow of sequential requests: Using simulated asynchronous I/O, I/O threads only submit a new requests when the current request has completed. There is thus no sustained flow of sequential request. Using native asynchronous I/O, requests are submitted in batches in order to prefetch entire extents. But could InnoDB do even better? The answer is yes.

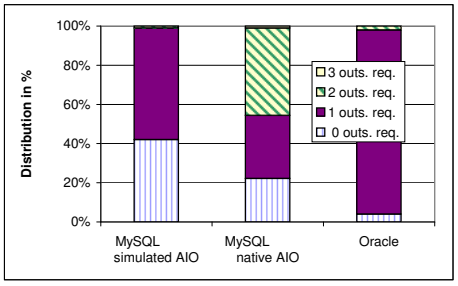


Figure 4: Distribution of outstanding requests when performing a scan

Figure 4 shows the distribution of outstanding requests when scanning a 800 MB table with MySQL/InnoDB using simulated asynchronous I/O, MySQL/InnoDB using native asynchronous I/O, and with Oracle 9.2. We measured the number of outstanding requests by sampling the kernel data structure managed by the disk scheduler. We calibrated the sampling rate using our benchmarking tool from Section 2.

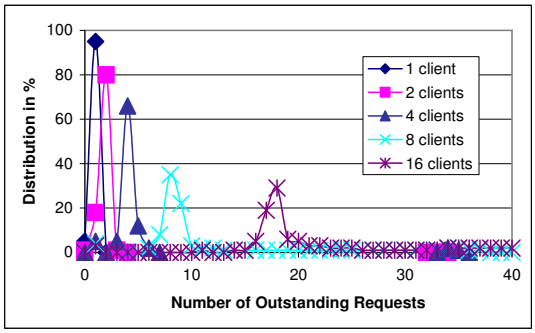
We saw in the previous section that the key to good sequential read performance is to maintain a sustained flow of requests, i.e., at least one outstanding request. Oracle does a good job: There is at least one outstanding request for 96% of the scan duration. InnoDB is much less efficient: There is no outstanding request for 22% of the scan duration using native asynchronous I/O (and for 42% of the scan duration using simulated asynchronous I/O). This is because InnoDB prefetches one extent at a time, while Oracle prefetches the whole table (see [9] for a complete analysis of InnoDB scan performance).

InnoDB chose to limit the number of outstanding requests to guarantee that latency critical I/O requests could be scheduled in a timely manner. Or-

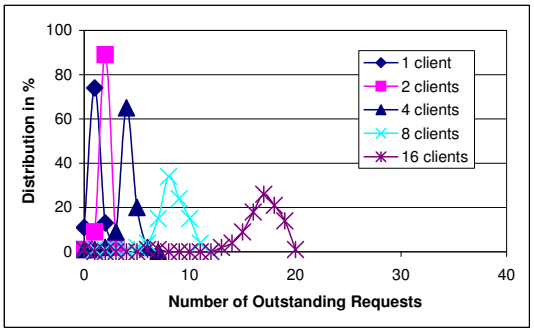
acle is more aggressive in its utlization of the I/O bandwidth for prefetching purposes.

Physical I/O

In order to study how efficiently InnoDB (and Oracle) submits physical I/O, we submitted range queries that select 1000 out of 3 millions tuples using a secondary index. We vary the number of client threads submitting these queries. Again, we measure the distribution of outstanding requests in the kernel by sampling the kernel data structure used by the disk scheduler.



(a) MySQL/InnoDB



(b) Oracle

Figure 5: Distribution of outstanding requests when performing range queries on MySQL/InnoDB and Oracle

Figures 5(a) and 5(b) trace the distribution of outstanding requests. They reveal similar behaviours. For both systems, the number of outstanding requests follows the number of query threads issuing random reads. Each query thread traverses the secondary index and accesses one data page at a time. There is thus one outstanding request per client thread.

In the case of InnoDB, we observe a heavy tail distribution. This is due to the random read ahead mechanism that prefetches extents on which random requests are concentrating. There is thus up to 64 oustanding requests, i.e., the number of I/O requests submitted to prefetch an entire extent.

We discussed in the Introduction the trade-off between throughput and latency of random requests. Submitting one I/O request at a time is a pretty extreme way to control this trade-off. It should be possible to achieve a better throughput by submitting batches of I/O requests without sacrificing latency.

Database Writes

Figure 6 traces the number of outstanding requests on the log file and on the data file (located on two different disks) when performing an update of a table (800 MB) larger than the database cache (600 MB) using InnoDB. We ran this experiment with Oracle and observed similar results.

Such a large update is interesting as it focuses on InnoDB behaviour under memory pressure. As there is constant read activity, there is no preflushing. When the buffer cache is full with uncommitted dirty pages, they need to be stolen to make room for free buffers.

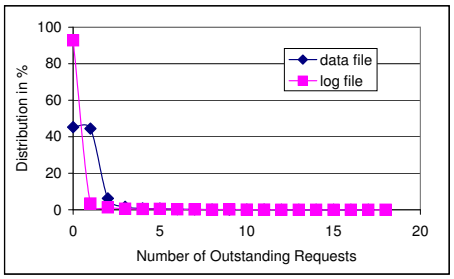


Figure 6: Distribution of Outstanding Requests when performing an update larger than the database buffer on InnoDB (Oracle displays a similar behavior)

We observe a low activitiy to the log file: there are no outstanding requests to the log file 98% of the time. The distribution of outstanding requests to the data file is characteristic of a mix of sequential reads (with a peak around 40% for 2, 3 outstanding requests and a long tail distribution due to prefetching) and random writes in relatively small batches (with a peak around 10% for 5 outstanding requests).

This is another illustration of InnoDB conservative policy. InnoDB partitions read and write requests in time. First, the storage manager only preflushes if there is a low I/O activity. Second, under memory pressure, InnoDB submits alternately read and write requests. Writes are submitted to disk in batches of limited size in order to reduce latency. In the meantime, the query thread is blocked on free pages. As soon as a page is freed a new page is read in. Note

that there is no avoiding a mix of read and write requests during such a large update. However, it is not the database that should arbitrarily alternate between the submission of reads and writes. The database should subit requests and let the file system organize the mix of read and writes as efficiently as possible.

3.3 Discussion: Towards an Aggressive I/O Policy

Our experiments show that a database system such as MySQL/InnoDB (as well as Oracle) sacrifices throughput to provision for latency. InnoDB submits few random requests in order not to block potential critical requests. This *conservative* approach is definitely sub-optimal.
As we showed in Section 2, the best performance is achieved with a sustained rate of sequential requests and with a reasonably high number of outstanding random requests. The number of outstanding random requests should be fixed by device to obtain a reasonable latency/throughput trade-off. As a consequence, we propose that InnoDB implements the following mechanisms to improve its utilization of the underlying I/O bandwidth:

1. *Prefetching.* The key requirement for sequential I/O performance is that requests are submitted at a sustained rate so that there is always at least one outstanding request. Instead of prefetching one extent at a time, InnoDB should prefetch data so that there is always at least one outstanding request. Whenever MySQL indicates that a scan is to be performed, the query thread could initially prefetch a couple of extents and then keep on prefetching extents as soon as one extent has been accessed. Such a design requires that the query thread keeps track of the extent boundaries (as it does currently) and keeps track of the next extent to be prefetched (this can be accessed from the primary index used to structure each table in InnoDB).

2. *Index read ahead.* Instead of accessing one page at a time when traversing an index, InnoDB could implement an index read ahead similar to SQLServer's. The idea is first to traverse the index and collect the page ids to be accessed, and then to submit read requests for these page ids in batches of tunable size (on our hardware platform, our experiments from Section 2 show that 16 outstanding request would be the ideal size for these batches). If the number of concurrent threads increases then the size of the batch should be reduced to maintain the number of outstanding requests per I/O device under a tunable maximum (the equivalent of `max_async_io` in SQLServer [24]).

3. *Database writes.* InnoDB manages the trade-off between latency and throughput by submitting random writes in large batches when I/O activity is low and batches of limited size when there is pressure on the database cache. Instead, InnoDB should maintain a steady stream of outstanding write requests to give the disk scheduler a chance to optimize throughput (again respecting the maximum number of outstanding requests per device). When there is no pressure on the database cache, writes should be performed when there is no other request to submit. When there is pressure on the database cache, writes should be agressively intertwined with read requests.

These propositions define an *aggressive* I/O policy where InnoDB submits I/O requests agressively to utilize the I/O bandwidth. The throughput/tradeoff latency is controlled at the storage manager level by a tunable parameter that fixes the maximum number of outstanding requests per device. Now this requires that the file system makes sure that outstanding asynchronous requests do not get in the way of the synchronous requests. In the next Section, we show that introducing prioritized I/O is an appropriate solution to this problem.

4 Linux Prioritized I/O

In this Section, we present the design and implementation of I/O priorities inside the Linux kernel. We evaluate the performance of our implementation and describe its impact on the InnoDB storage manager.

4.1 Design

We modified the Linux disk scheduler to account for priorities as follows. Instead of assigning deadline based on *fixed time offsets* (1 sec for reads and 5 sec for writes), the scheduler assign to each I/O request a deadline based on its priority: Requests submitted with high priority will get deadlines that expire within a short time interval and requests with low priority will get deadlines that will expire within a longer time interval.
Let us first define our notion of priority. We use *absolute deadlines*, i.e., to each priority level is associated a different deadline (0,25 sec for the highest priority level and 12 sec for the lowest priority level). We use 5 levels of priority.
Deadlines based on *variable time offsets* required some changes to the deadline scheduler. Since time offsets were fixed in the original design, the scheduler maintainted read and write requests using FIFO

queues. The most natural extension is to define one FIFO queue per priority level. This is what we implemented [12]. Whenever a deadline expires in one of the priority queues, the corresponding request is moved to the dispatch queue. If no deadline has expired, requests are moved from the sector-ordered red-black trees to the dispatch queue as before.

Another key aspect of the disk scheduler concerns the allocation of requests. Linux preallocates a number of requests available for normal disk I/O (special commands for eg. disk flushes do not use the preallocated requests). There is not much point in setting different deadlines, if a task is able to get all the preallocated requests for low priority requests. This is why we also consider priorities when allocating requests. Our solution consists in defining allocation groups associated to priority levels. A task that submits a low priority requests gets preallocated requests from the low priority allocation group and does not interfere with high priority requests.

4.2 Implementation

Our implementation did not require to change the file system interface. Indeed the data structure passed as an argument to the `io_submit` system call(see below) already accounts for priorities (as required by the POSIX standard [16]). No modification was needed for the event completion mechanism. Completion events are fetched through the `io_getevents`

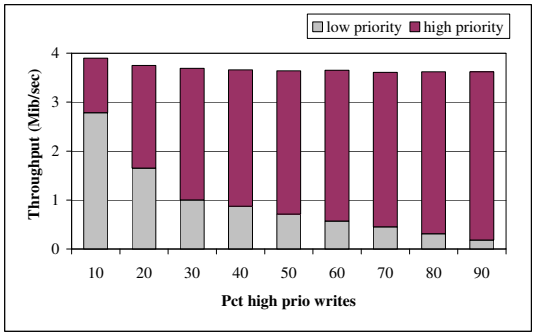
```
long io_submit (aio_context_t ctx_id,
                long nr, struct iocb **iocbpp);

struct iocb {
// Pointer returned on completion
        void            *data;
// Internal key
        unsigned        key;
// read, write etc.
        short           aio_lio_opcode;
// Request priority
        short           aio_reqprio;
// Filedescriptor
        int             aio_fildes;
// Buffer, buffer size and offset
        void            *buf;
        unsigned long   nbytes;
        long long       offset;
};
```

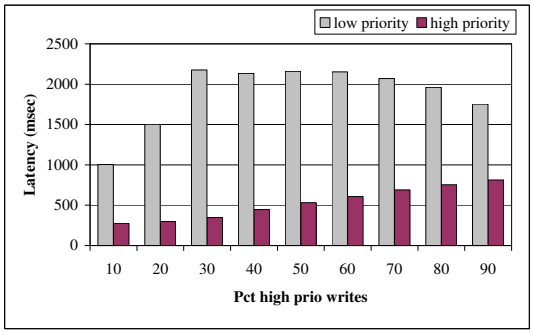
[12] An alternative would have been to maintain the requests explicitely sorted in one data structure, i.e., either an insertion sort on the deadline queues – which would be CPU intensive – or insertions into a B+-tree storing the requests based on their deadline – which introduce non negligible complexity into the kernel.

The patch to the Linux kernel contains a new disk scheduler (basically a modified version of the deadline scheduler). In addition, only a few changes were needed to carry the priority information all the way from the asyncronous state machine and down to the disk scheduler. The patch is publically available[13].

4.3 Performance Evaluation



(a) Throughput



(b) Latency

Figure 7: Average throughput and latency for a mix of 200 outstanding high and low priority requests. Note that in our pure experiments (Section 2), throughput for 200 requests was around 4 Mib/sec while average latency was around 750 msec.

Figure 7 traces throughput and latency for a mix of high and low priority random requests. We maintain 200 outstanding requests throughout the experiment and we vary the percentage of high priority requests. The requests are random writes over a 10 Gb file. We measure average latency and throughput for high and low priority requests. High priority requests have a deadline of 3 sec and low priority requests have a deadline of 9 sec (recall that the

[13] http://www.distlab.dk/badger/deadline_prio.patch.gz

native deadline deadline for writes is 5 sec). This experiment is run on the server described in Section 2.2.

Figure 7(a) shows that the total throughput for low and high priority requests is around 4 Mb/sec which corresponds to the numbers from Figure 1. When we increase the percentage of high priority requests, their throughput increases sharply until they constitute 30% of the mix. Thereafter throughput still increases but slowly.

In terms of latency, we observed around 800 msec in the pure experiment for 200 outstanding write requests (see Figure 3). Figure 7(b) shows that the latency is around 280 msec for high priority requests when they constitute 10% of the mix. At the same time the latency of low priority requests is around 1000 msec. When we increase the percentage of high priority requests, their latency remains low (below 350 msec) until they constitute 30% of the mix. At the same time the latency of low priority requests increases up to 2000 msec. When the percentage of high priority requests is higher than 30% the latency of low priority requests remains constant while the latency of high priority requests increases.

In the context of database I/O the most interesting mix consists of low priority sequential I/O (prefetching or preflushing) and high priority random I/O (physical reads). We ran an experiment where a thread submits random reads while three threads maintaining a sustained rate of sequential read requests.

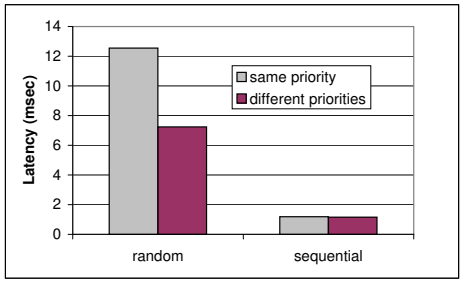


Figure 8: Latency of mixed sequential and random requests.

Figure 8 shows the result when sequential and read requests have the same priorities and when a high priority (with a deadline of 1 sec) is associated to random reads and a low priority (with a deadline of 3 sec) is associated to sequential reads (recall that the native deadline associated to reads is 1 sec). As expected, the graph shows that when sequential and random read get the same level of priority, the latency of random requests suffers: We observe a latency of 13 msec for the random request. With a high priority, the latency of random read is back to 7 msec which is the latency we observed for one outstanding random read requests in Figure 3. Interestingly the average latency of sequential requests does not suffer when it is associated a low priority. This is because sector sorting is still active. The performance of sequential requests would suffer significantly if their deadline were to expire. This is obviously not the case in our experiment.

These experiments show that our implementation lives up to the promises of prioritized I/O. The prioritized disk scheduler achieves a low latency for high priority requests while maintaining an overall high throughput. Our experiments show that we achieve these good performances if high priority requests constitute up to 30% of the submitted requests.

4.4 Impact on InnoDB

The re-implementation of InnoDB to incorporate an aggressive I/O policy is out of the scope of this paper. In order to validate our approach, we focused on InnoDB preflushing policy.

We modified InnoDB in order to associate priorities to the I/O requests that it submits. We associate a low priority to asynchronous read requests (prefetching) and a high priority to synchronous requests (physical reads). We modified the preflushing mechanism as follows. Once per second a background thread checks the amount of repleacable pages in the database buffer. If less than 50% of all pages are free then (up to) 64 pages are flushed with a low priority. We increase the number of pages flushed with low priority to $5 * 64$ if the percentage of free pages falls to 40%. and to $10 * 64$ if the percentage of free pages falls below 40%. Whenever the percentage of free pages reaches 10%, we flush 100 pages with high priority and 100 pages with low priority.

This preflushing mechanism submits database writes more and more aggressively as the amount of free pages diminishes. Interestingly, the pririty associated to write requests evolves adaptively depending on memory pressure. This is possible because priorities are associated to individual I/O requests (and not to processes or transactions).

When running the large update from Section 3, we observe a 5% improvement in terms of response time. Figure 9 shows the distribution of outstanding requests for the conservative approach (from Section 3) and for the aggressive approach. The aggressive policy reduces the amount of idle time for the disk controller and increases the likelihood of 2 or 3 outstanding requests.

This experiment shows the potential of an aggressive policy. However, our implementation still submits I/O requests in batches (write requests are submitted every second). What is needed to significantly improve performance is to modify the InnoDB stor-

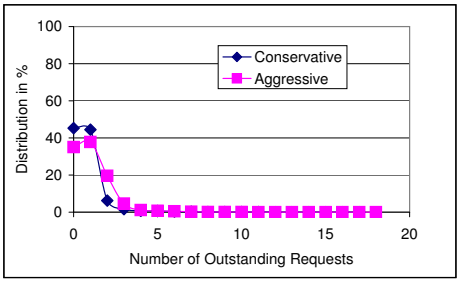


Figure 9: Distribution of outstanding requests when performing an update larger than the database buffer with conservative and aggressive policies

age manager so that it submits a constant flow of I/O and thus fully utilizes the underlying I/O bandwidth. This is a topic for future work.

5 Related Work

Prioritized I/O, are commonplace on mainframes. For example, IBM supports I/O request priorities in the context of its Enteprise Storage Server [19]. A priority is associated to each I/O request. Within the fibre channel adapter, requests are dispatched into different queues depending on their level of priority. The scheduled requests are taken from the highest priority non empty queue. Once the active queue is empty, requests from lower priorities are promoted one priority level and the new highest priority non empty queue becomes active. This queue promotion process guarantees that low priority requests do not starve. Our case for prioritized I/O in Linux follows the downsizing trend from mainframes to commodity servers analyzed by Gray and Nyberg [14].

Even if the POSIX standard defines priorities, we do not know of any UNIX implementation of prioritized disk I/O. Jens Axboe posted a version of the Linux anticipatory disk scheduler that supports priorities [5]. The priority level of I/O requests is fixed by the priority of the task that submits them. This does not correspond to the database needs (in particular the preflushing needs expressed in Section 4).

Recently, McWerther et al. [18] argued that priority mechanisms were needed inside the database systems to efficiently support OLTP and transactional web applications. Their study show that PostgresSQL with its multi-read consistency model (similar to Oracle and InnoDB) exhibits an I/O bottleneck when running the TPC-C and TPC-W benchmarks. The I/O priorities we argue for are a natural complement to their CPU and lock scheduling policies.

There are few studies of Linux asynchronous I/O. The most complete description so far was led at IBM Linux Center [8, 7].

More generally, I/O have not received a lot of attention in the database research community lately. Most results are published in measurements (CMG, SIGMETRICS) or high performance computing conferences (HPCA), in white papers (e.g., [25, 17, 2], or on Jim Gray's home page (e.g., [12, 14]).

We chose to consider a simple hardware configuration involving a server connected to a couple of disks because our point concerned the way the database utilize the underlying kernel services. Now, it will be interesting to study the behaviour of Linux databases on hardware set-up including large SMP, and clusters as well as storage area networks (SAN). Arpaci-Dusseau et al. [3] studied the impact of different architectures (server, SMP and cluster) on the performance of streaming I/O. They concluded that none of the architectures were well-balanced and that CPU was becoming a bottleneck before any other ressources as they increased the amount of I/O. Their data processing benchmarks (scan and insert) issued I/O requests as efficiently as possible. We showed in Section 3 that this was not the case for the asynchronous I/O submitted by InnoDB (and to a lesser extent Oracle).

SAN raise a set of interesting challenges as they encapsulate a significant portion of I/O processing (including cache management and request scheduling) [4]. Our goal with this paper was to study the collaboration between a database server and the kernel disk scheduler. When using a SAN, the scheduling of I/O requests does not take place at the OS level but within the SAN controller. Improving the collaboration between a database server and the SAN controller raises great challenges. Schindler et al. [21] already proposed to communicate performance characteristics from the storage device (e.g., preferred access patterns) to the storage manager so that it can take take informed decisions when submitting I/O requests. This area should definitely be investigated further.

6 Conclusion

Our goal was to find out whether MySQL/InnoDB on top of Linux took best advantage of the available I/O bandwith. We showed that the conservative I/O submission policy implemented by InnoDB (and Oracle) constitutes a barrier to I/O performance. In order to remove that barrier, the database storage manager needs to rely on the operating system's ability to process large amounts of asynchronous I/O while guaranteeing the latency of synchronous I/O. We designed and implemented priorities in the Linux kernel for that purpose. We showed that our implementation is flexible and efficient, and that it is now

possible to define a more aggressive I/O submission policy for InnoDB and for Linux databases in general.

The support for native Linux asynchronous I/O we implemented in InnoDB has been transferred to InnoDB Oy. The patch of the Linux kernel is publically available, making it possible for the data management research community to experiment with various aggressive I/O submission policies. This is hopefully a first step towards a more effective collaboration between this research community, the database industry and the Linux community.

Improving the collaboration between a database system and the underlying storage system presents plenty of interesting challenges, e.g., how to leverage the storage cache hierarchy? How to control the throughput-latency trade-off when submitting I/O requests to a SAN? These are topics for future research.

References

[1] IBM Ultrastar 36LZX. Documentation. http://www.hgst.com/tech/techlib.nsf/products/Ultrastar_36LZX.

[2] Steve Adams. The Mysteries of DBWR Tuning, 1997. http://www.ixora.com.au/tips/mystery.doc.

[3] Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and Dave Patterson. The Architectural Costs of Streaming I/O: A Comparison of Workstations, Clusters, and SMPs. In *Symposium on High-Performance Computer Architecture (HPCA '98)*, February 1998.

[4] Mark Cohen Austrowiek and Pierluigi Grassi. UNIX IO Performance Measurement Methodologies Applied to Old and New Storage Technologies. In *EuroCMG*, 2002.

[5] Jens Axboe. Cfq + IO Priorities. http://www.kerneltrap.org/comment/reply/1596.

[6] Suparna Bhattacharya. Linux Asynchronous IO. http://www.kernel.org/pub/linux/kernel/people/suparna/aio/.

[7] Suparna Bhattacharya. Personal Communication.

[8] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Ottawa Linux Symposium*, 2003.

[9] Philippe Bonnet Bjarke Buur Mortensen. Beyond Response Time: Analyzing MySQL Performance. Submitted to publication.

[10] IBM Linux Technology Center. Home Page. http://www.ibm.com/linux/tlc/.

[11] Do Linux Asynchronous I/O Really Matter? DIKU Technical Report 04/03.

[12] Leonard Chung, Jim Gray, Bruce Worthington, and Robert Horst. Windows 2000 Disk IO Performance. Technical Report MS-TR-2000-55, Microsoft Research, 2000.

[13] Garth Gibson, Jeffrey Scott Vitter, and John Wilkes. Report of the Working Group on Storage I/O for Large-Scale Computing. *ACM Computing Surveys*, December 1996.

[14] Jim Gray and Chris Nyberg. Desktop batch processing. In *Proceedings of COMPCON 94*, 1994.

[15] Oracle's Linux Project Development Group. Home Page. http://oss.oracle.com/.

[16] The Open Group. Base Specifications Issue 6, 2003. http://www.opengroup.org/onlinepubs/007904975/.

[17] Oracle Performance Tuning Tips: Use asynchronous I/O. http://www.ixora.com.au/tips/use_asynchronous_io.htm.

[18] David McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority Mechanisms for OLTP and Transactional Web Applications. In *ICDE 2004*.

[19] A.S. Meritt, J.A. Staubi, K.M. Trowell, G. Whistance, and H.M. Yudenfriend. z/OS support for the IBM TotalStorage Enterprise Storage Server. *IBM Systems Journal*, July 2003.

[20] Oracle Technical White Paper. Oracle 9iR2 on Linux: Performance, Reliability and Enhancements on Red Hat Linux Advanced Server 2.1, 2002.

[21] Jiri Schindler, Anastassia Ailamaki, and Gregory Ganger. Matching Database Access Patterns to Storage Characteristics. In *VLDB 2003 PhD Workshop*, 2003.

[22] Dennis Shasha and Philippe Bonnet. *Database Tuning: Principles, Experiments and Troubleshooting Techniques*. Morgan Kaufmann, 2002.

[23] Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin. *Operating System Concepts*. Wiley Text Books, 6th edition, 2002.

[24] Ron Soukup and Kalen Delaney. *Inside Microsoft SQL Server 7.0*. Microsoft Press, 1999.

[25] Nitin Vengurlekar. Oracle Disk Manager. White paper, Oracle Solutions Support Center, 2002. http://otn.oracle.com/deploy/availability/pdf/nitin_ODM.pdf.