Datalogisk Institut
KØBENHAVNS UNIVERSITET

Do Linux Asynchronous I/0 Really Matters?

Christoffer Hall, Bjarke Mortensen, Philippe Bonnet, Heikki Tuuri, Peter Zaitsev

Technical Report no. 04/03
ISSN: 0107-8283

DIKU
University of Copenhagen • Universitetsparken 1
DK-2100 Copenhagen • Denmark

Do Linux Asynchronous I/O Really Matter?

Christoffer Hall[1], Bjarke Mortensen[1], Philippe Bonnet[1], Heikki Tuuri[2], Peter Zaitsev[3]

[1]University of Copenhagen, [2]Innobase Oy Inc., [3]MySQL AB
{*hall, rodaz, bonnet*}*@diku.dk*, *heikki.tuuri@innodb.com*, *peter@mysql.com*

Abstract

The newly released Linux 2.6 kernel supports asynchronous I/O. This evolution corresponds to the wishes of established database vendors. The question is though: What kind of impact do asynchronous I/O actually have on database performance? This depends on the performance characteristics of the Linux I/O subsystem, but also on the way the database server utilize it. In the context of the Badger project, a collaboration between MySQL AB and University of Copenhagen, we evaluated how MySQL/InnoDB can best take advantage of Linux asynchronous I/O. This paper documents our analysis and our proposals. We present a simple tool that we use to extract the key performance characteristics of Linux asynchronous I/O. We show that MySQL/InnoDB does not utilize their full potential (neither does Oracle 9.2), and we propose modifications of the InnoDB storage manager to remedy these limitations. Finally, we make the case that database performance could be further improved if Linux supported prioritized asynchronous I/O.

1 Introduction

1.1 Linux, Databases and I/O

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 [13, 10]. Oracle and IBM in particular have argued that kernel support for asynchronous I/O[1] was critical for database performance [19]. Their efforts have resulted in the adoption of asynchronous I/O in Linux 2.6.

In the context of the Badger project, a collaboration between University of Copenhagen and MySQL AB, we studied how MySQL equipped with the InnoDB[2] storage manager could best take advantage of Linux asynchronous I/O: What are the performance characteristics of Linux asynchronous I/O? Does the MySQL/InnoDB storage manager take best advantage of them? Can the Linux kernel be further enhanced to support database I/O? This paper documents our analysis and our proposals.

But why even bother with I/O? As all other database vendors, MySQL and InnoDB developers regularly interact with clients who want to get the best performance out of their I/O devices. Ideally, it is the hardware configuration that limits I/O performance, but *does 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.

1.2 The Promises of Asynchronous I/O

In the rest of the paper, we adopt a classic representation of the software layers on top of I/O devices (see Figure 1): The database server relies on OS kernel services to access the underlying I/O devices. Kernel services are typically organized in three layers [21]. 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 (e.g., layout and

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

[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 [14].

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

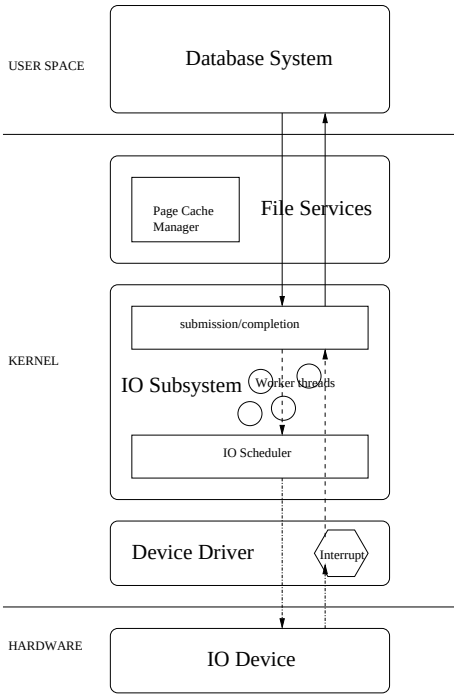

Figure 1: Software Layers (database and kernel services) on top of I/O Devices

metadata management, caching). We detail the internals of the Linux kernel services in Section 2.
Before we discuss the potential impact of asynchronous I/O on database performance, let us briefly review the I/O requests generated by a database system:

- Random reads to data files (as well as system or temporary files) at query processing time when a user transaction requests a page which is not found in the database buffer (these are called *physical I/O*).
- Sequential read requests to prefetch pages from a data file when scanning a table.
- Sequential writes to the log files when log records are forced to disk (on one or several log files).
- Random write requests to lazily write dirty pages to data files whenever the amount of free pages in the database buffer is too low or possibly whenever the checkpoint procedure requires it to do so.

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, in principle, 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. *Parallelize reads and write requests to multiple files.* This benefits concurrent requests on log and data files.
3. *Overlap I/O and CPU work.* This benefits physical I/O in particular when prefetching (or sorting) data.

Historically, Unix has implemented I/O requests as synchronous operations [12]. Kernel support for asynchronous I/O was introduced on raw devices in the mid-90s on Solaris 2.4 and other versions of Unix later on [24]. The problem with raw devices is that they are not practical to manage. To alleviate this problem, Veritas among others provide disk managers offering file system interfaces and fine grained backup capabilities on top of raw devices[26]. Linux Volume managers [22] also provide some management and backup capabilities when using raw devices. We use raw devices as a point of reference in our experiments.
Before kernel asynchronous I/O were available, Unix databases emulated non blocking I/O on top of the available synchronous primitives, with ancillary threads spawned to submit synchronous requests without directly blocking the threads managing user transactions (i.e., log writers, lazy writers and prefetchers). These ancillary threads provided an overlap between I/O and CPU work but still proceeded one I/O request at a time. As of version 7, Unix versions of Oracle could be configured to utilize asynchronous I/O. So far, MySQL/InnoDB still emulates asynchronous I/O on Linux. In the context of this study, we extended InnoDB to support native Linux asynchronous I/O. We describe our implementation in Section 3.
Interestingly, asynchronous I/O have always been available on Windows NT. In 2000, L.Chung and J.Gray studied the performance characteristics of the Windows NT file system from a database perspective [11]. Such a comprehensive study still has to be conducted for Linux.

1.3 Contribution

To study how InnoDB can best take advantage of Linux asynchronous I/O, we decided to tackle the following issues:

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

1. *What are the performance characteristics of Linux asynchronous I/O?* Our goal is to identify the regime where Linux asynchronous I/O provide best performance in terms of latency and throughput.

2. *Does MySQL/InnoDB (and as a point of comparison Oracle) utilize Linux asynchronous I/O to their full potential?* A database server typically submits a mix of read and write requests. Some of those block the threads executing user transactions (writes to log, physical I/O) and should complete as quickly as possible, others are executed in the background (prefetch reads, lazy writes) and should be executed in batches that maximize throughput. Does the actual I/O workload generated by MySQL/InnoDB correspond to the optimal regime for Linux asynchronous I/O performance? If not, how can InnoDB be modified to better utilize Linux asynchronous I/O performance?

3. *Can Linux asynchronous I/O be improved to support database workload?* More precisely: Can the OS kernel help the database server control the trade-off between I/O latency and throughput? For example, if we push the idea of lazy writing, the database server should be able to submit a batch of write requests and tell the OS to schedule them so that they do not block any other I/O request.

This paper documents our analysis and our proposals. More specifically, we make the following contributions:

- We evaluate the capacity of Linux to accumulate asynchronous I/O requests and schedule them efficiently, and we discuss the potential impact on database performance. The simple tool we used to run our experiments is available on-line[4] so that system administrators can evaluate the key performance characteristics of Linux asynchronous I/O on their own installation.

- We show that MySQL/InnoDB does not utilize the full potential of Linux asynchronous I/O (and that neither does Oracle). We propose modifications of InnoDB to remedy these problems.

- We argue that asynchronous I/O could be enhanced with priorities to control the latency/throughput trade-off for the mix of I/O requests that characterize a database workload.

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

We would like to note that the emergence of commercially viable open source systems empowers the database research community (and not just a few industrial labs) to impact the design and implementation of actual products. This paper is a first step in this direction.

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

Let us zoom in on the Linux kernel services [1] illustrated in Figure 1. The file system provides the file abstractions that the database storage manager relies on[5] and implements a set of services including the 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 for each file when it is opened (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 case of direct I/O, the I/O completion is notified from the

[5] We ignore memory-mapped operations as they are not an option for database systems, at least on 32 bits architectures.

[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.

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. Another way to minimize seeks is to 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 (0,5 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 in a Red-Black tree, that sorts requests on the first sector they access and 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 tree.

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. Based on the results of our experiments, we discuss how databases could take best advantage of Linux asynchronous I/O.

[7]Linux 2.6 actually 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.

2.2.1 Experimental Set-up

Our experiments focus on the capacity of Linux asynchronous I/O to optimize performance by accumulating read or write requests. We conduct two sets of experiments:

- *pure*: We consider pure workloads, consisting of either random read, random write, sequential read, or sequential write. We vary the number of outstanding requests. We expect that increasing the number of outstanding requests will improve throughput at the cost of increased latency. We conduct these experiments with buffered I/O, direct I/O, and raw I/O. The *pure* experiments are a baseline that characterize the performance of Linux asynchronous I/O.
- *mix*: We consider workloads where random and sequential I/O requests are mixed. We vary the amount of random requests to find out how much perturbation mix workloads introduce with respect to the *pure* baseline.

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 1 GB file. Each request manipulates 16KB (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 (1 GB) over the total time for all requests. We measure CPU usage using mpstat. We remounted the file system (ext3) between runs to enforce a cold cache.
The I/O devices in our experiments are just a bunch of disks (IBM Ultrastar 36LZX), directly connected to a dual 1GHz Pentium III server via a SCSI bus on two different channels (Dual chanal Adaptec AHA-3960D controller). The disks are configured with read ahead and write back enabled. The server has 1GB of RAM. This simple hardware configuration (no RAID) 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/sec which matches the disk specification [2]. This is the sequential throughput we can hope for.

2.2.2 Results

Pure Workload
We expect that increasing the number of outstanding requests will improve throughput and increase latency.
Figure 2 shows the throughput of random requests for buffered and direct I/O as we increase the number of outstanding requests. We should note that

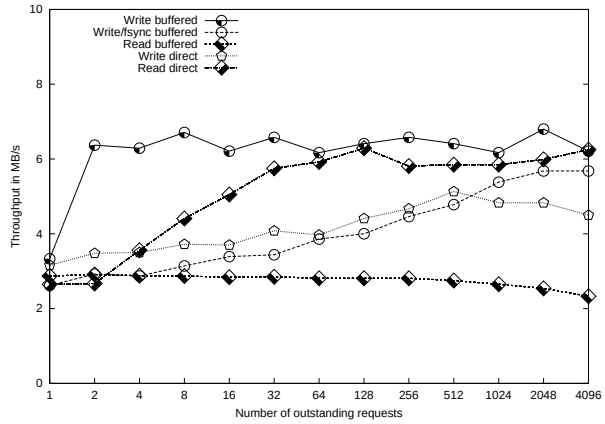

Figure 2: Throughput of random requests (buffered vs. direct I/O)

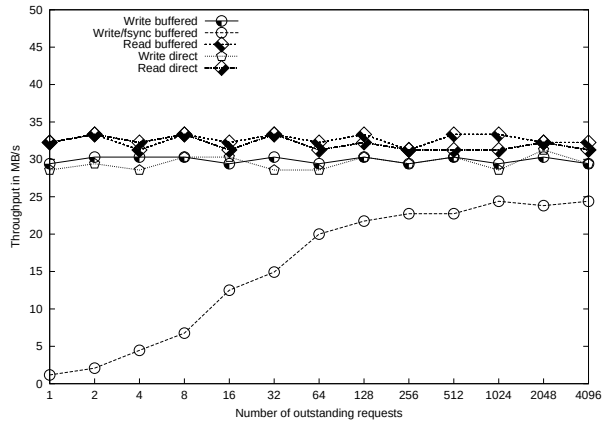

Figure 3: Throughput of sequential requests (buffered vs. direct I/O)

we issue a single fsync for *write buffered*, while for *write/fsync buffered* we issue an fsync command every N request we submit (e.g., when we maintain 64 outstanding requests, we issue a fsync every 64 requests).

Throughput remains constant at around 2,5 MB/sec for buffered reads and at around 6 MB/sec for buffered writes (with a single fsync). The frequency of fsync has a significant impact on performance. Throughput remains lower than 4 MB/sec until fsync commands are issued every 1024 requests or more[8]. The good performance of buffered write is not surprising as requests are submitted to disk in large chunks by the lazy writer. The poor performance of buffered reads is mainly due the unnecessary read ahead. The throughput of direct reads increases quickly with the number of outstanding requests: from 3 MB/sec for 1 outstanding request to 6 MB/sec for 64 outstanding requests. In comparison, the throughput of direct writes increases slowly: it reaches 5 MB/sec for 512 outstanding requests. As expected, the increased throughput is due to the ordering of requests in the disk scheduler. Requests spread over a larger file (or a fragmented file) would result in a more modest throughput increase.

[8]The disk controller maintains a number of outstanding requests and schedules them to minimize access time, i.e., the combination of seek and rotational latencies required to bring the desired physical block under the head.

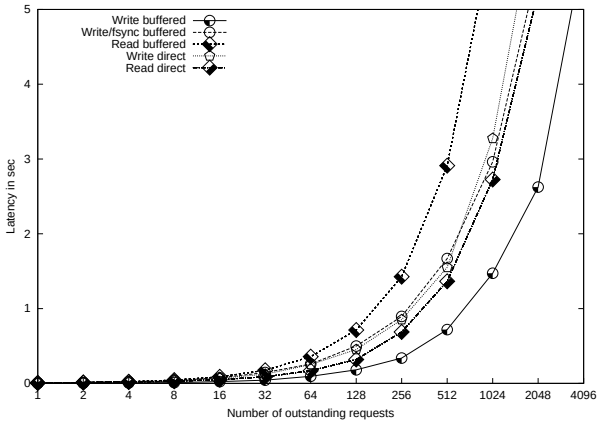

Figure 4: Latency of random requests (buffered vs. direct I/O)

Figure 3 traces the throughput of sequential requests as a function of the number of outstanding requests. We could expect that increasing the number of outstanding requests would lead the scheduler to merge requests into large blocks and thus favour 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 read compared to writes. This is because the disk controllers implement a form of read ahead [2] that benefits read requests. Those results show that for sequential requests, the number of outstanding requests is irrelevant. The key parameter is the rate at which requests are submitted. The frequency of fsync has a very significant impact on performance: throughput improves from 1 to 25 MB/sec when the frequency of fsync increases from one per request to one per 4096 requests.

Figure 4 traces the average latency of random requests as a function of the number of outstanding requests. The latency increases linearly with the number of outstanding requests. 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.

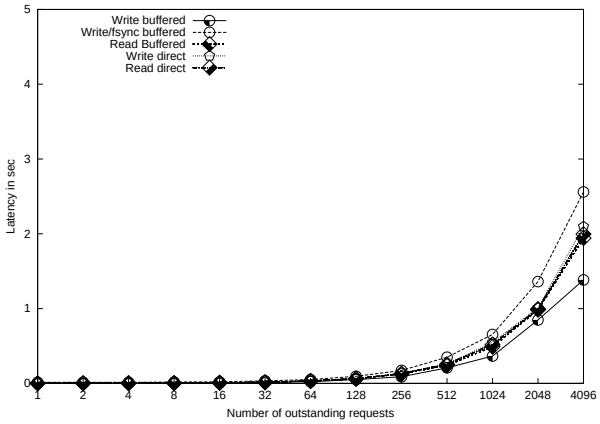

Figure 5: Latency of sequential requests (buffered vs. direct I/O)

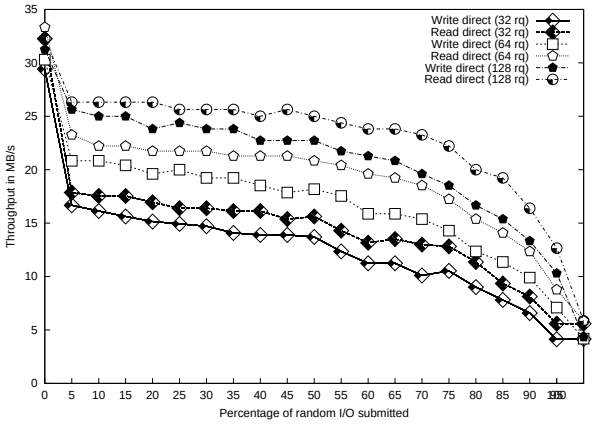

Figure 7: Throughput of a mix of sequential and random requests

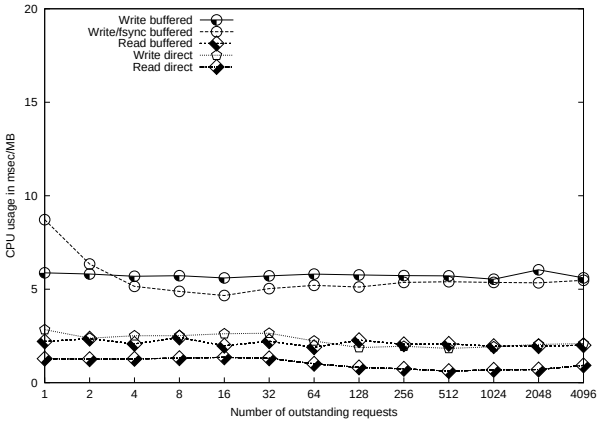

Figure 6: CPU usage for sequential requests (buffered vs. direct I/O)

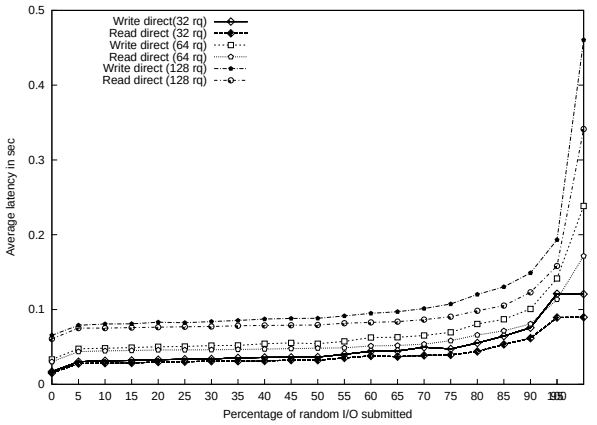

Figure 8: Latency of a mix of sequential and random requests

Figure 5 traces the latency of sequential requests as a function of the number of outstanding requests. 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.

Figure 6 traces CPU overhead as a function of the number of outstanding requests. CPU usage is high for buffered writes, because of contention between our benchmark tool and the lazy writer. For the other requests, CPU usage is lower than 3 msec/MB (this is comparable to the CPU overhead measured by Chung et al. for direct I/O on Windows 2000 [11]).

Additional experiments showed that 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.

Mix Workload

Figure 7 (resp. 8) traces the throughput (resp. latency) of a mix of sequential and random requests. The x-axis represents the percentage of random requests in the mix. As expected, throughput decreases sharply as soon as random I/O are mixed with sequential I/O. However, troughput remains almost constant until 50% of the requests are random. Throughput falls sharply when the pecentage of random requests is higher than 75%. As our pure experiments indicated, throughput increases with the number of outstanding requests. Latency follows a similar pattern: it increases significantly as soon as random requests are submitted in addition to sequential requests, it then remains stable until the percentage of random requests reaches 70% after which latency increases very sharply.

2.2.3 Discussion

Our experiments focused on the capacity of Linux asynchronous I/O to accumulate I/O requests and schedule them efficiently. Our results show that: Near optimum random read throughput is achieved with only 32 outstanding requests, near optimum random write throughput is achieved with 512 outstanding requests, latency of random requests is higher than 0,5 seconds if the number of outstanding requests is greater than 128 outstanding requests. Our benchmarking tool can be used by system administrators to gather key characteristics of the Linux asynchronous I/O on their own installation.
The experiments we ran do not represent an extensive study of the performance characteristics of Linux asynchronous I/O. We did not evaluate the performance impact of varying the data size of the I/O requests[9]. Also, our experiments are run with a single thread issuing requests on a single file: We did not conduct scalability experiments. IBM Linux Technology Center is conducting such experiments. Preliminary results are presented in [9, 7].
We draw the following conclusions from our experiments:

1. *Linux databases should use direct I/O* Direct I/O offer best performance for the mix of sequential and random request that characterize database workloads. Compared to buffered I/O, direct I/O provide better random read, better CPU utilization, and equivalent scan performance. And if the number of outstanding requests is high enough, direct I/O approaches the performance of buffered I/O for random writes. Direct I/O provides equivalent performance compared to a raw device while providing a file system abstraction.

2. *The rate of submission is key for sequential operations.* This impacts prefetching (sequential reads) and log writing (sequential writes). Both operations should result in a sustained flow of request submissions. These operations will be most efficient if they succeed in maintaining at least one outstanding I/O request. Throughput drops significantly as soon as random requests are mixed with the sequential requests. They should be kept separated as much as possible (separate disks for log and data that are scanned extensively).

3. *The throughput of random reads increases sharply with the number of outstanding requests.* In a database workload, random reads correspond to physical I/O directly issued by the query thread, which means that latency is of the essence. As a consequence, it does not seem reasonable to consider more than 64 or 128 outstanding read requests in our set-up. Fortunately, 32 outstanding requests already achieve close to optimal throughput. Also, the mix experiments show that throughput and latency remain high if few random requests are mixed with sequential requests. As a consequence, the database system should control the number of outstanding random read requests.

4. *The throughput of random writes increases slowly with the number of outstanding requests.* In a database workload, random writes correspond to lazy writes of dirty pages. It should be possible for the database server to collect large number of dirty pages and submit them in large batches of random writes in the background. The high latency is not a problem as long as random writes do not interfere with other I/O requests. Ideally, the random writes are scheduled when no other I/O request is pending. This would require that the disk scheduler integrates the notion of priority. We discuss this issue in Section 4.

[9] InnoDB performs I/O using 16KB pages

3 InnoDB and Asynchronous I/O

InnoDB is the most classical of MySQL storage managers as its model is comparable to Oracle's: Support for multi-read consistency, separation of redo and undo log, utilization of tablespaces as abstraction for data files. We focus on I/O, see [16] for more details.

3.1 InnoDB and I/O

Before we present how InnoDB issues I/O requests, let us review how files are organized. InnoDB manipulates 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[10]. 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 stores log records in memory is 50% full. In

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

addition, the background server thread forces log records to disk every second.

- *Random writes of dirty pages.* Dirty pages are flushed to disk in batches of tunable size whenever there is not enough free pages in the database buffer. In addition, the background server collects up to 100 dirty pages and submits them as a batch if the observed I/O activity is low.

- *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, InnoDB uses `WaitForMultipleObjects` to wait for events. This funtion returns when a given event is ready for processing. This is not the case in Linux as `get_event` returns with a list of events. Instead an `eventprocessing` thread is created, that processes completion events and signals any thread that is waiting for the completion of a given I/O request.

3.2 Utilization of Asynchronous I/O

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

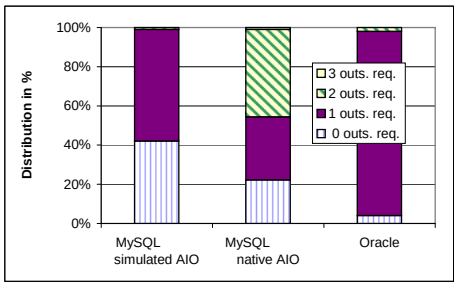

Figure 9: Distribution of outstanding requests when performing a scan

Using asynchronous I/O, a scan of 800 MB takes 29 seconds as opposed to 33 seconds with simulated asynchronous I/O. The gain can be explained as follows. Using simulated asynchronous I/O, when a request completes there is no outstanding request in the kernel, it is InnoDB's I/O thread that submits the next request. Using native asynchronous I/O, 64 requests are submitted at once so each extent is prefetched efficiently. But could InnoDB do even better? The answer is yes.
Figure 9 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 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 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 due to the fact that each extent is prefetched independently. InnoDB would improve performance by adopting a prefetching strategy similar to Oracle's.
The rationale for prefetching an extent at a time is that there is no guarantee that extents of a same relation are contiguous on disk. It is true that if extents are not contiguous, a seek has to be performed. But prefetching one extent at a time, introduces significant processing time that could overlap with I/O processing.
Physical I/O
In order to study how InnoDB (and Oracle) submit 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

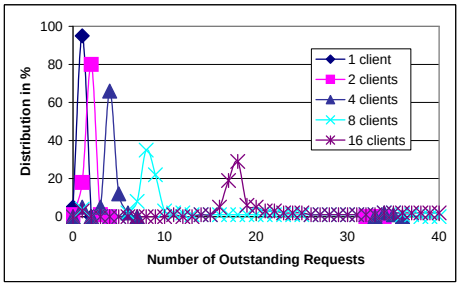

Figure 10: Distribution of outstanding requests when performing range queries on MySQL/InnoDB

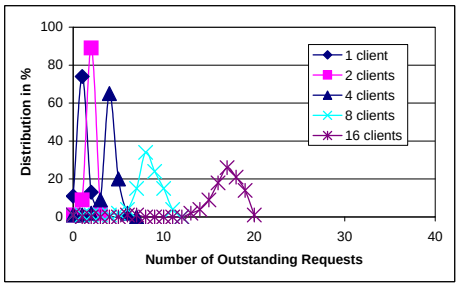

Figure 11: Distribution of outstanding requests when performing range queries on Oracle

these queries. Again, we measure the distribution of outstanding requests in the kernel by sampling the kernel data structure used by the disk scheduler.
Figures 10 and 11 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 (one per client thread): Each query thread traverses the secondary index and accesses one data page at a time.
In the case of InnoDB, we observe a heavy tail distribution with a peak around 34 outstanding requests. This is due to the extent read ahead mechanism that prefetches extents on which random requests are concentrating. On average however, the number of outstanding requests is slightly higher than the number of client threads.
We saw in the previous section that the trade-off between latency and throughput should be controlled by the database server. However, fixing one outstanding random read per query thread is definitely not the most efficient way to control this trade-off: (1) A single query thread could issue multiple outstanding random reads and (2) the number of outstanding requests should be limited per I/O device not per query thread. Both InnoDB and Oracle could improve the performance of multipoint and range queries by modifying the way physical I/O is submitted.

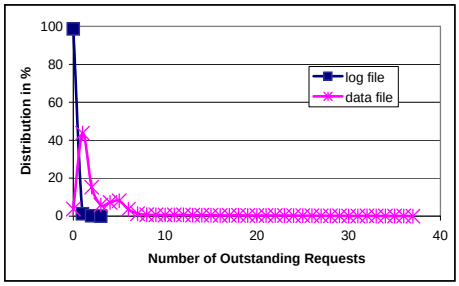

Figure 12: Distribution of Outstanding Requests when performing an update larger than the database buffer on InnoDB (Oracle displays a similar behavior)

Database Writes

Figure 12 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.

When running a large update, we expect the database system (a) to write frequently to the log (as soon as it is full enough), (b) to read data from the data file and (c) to write once in a while large batches of dirty pages to the data file. We observe a very different phenomenon. During 98% of the update there are no outstanding requests to the log file. It means that read and write requests to the data file constitute a bottleneck. More precisely, at some point during the execution of the update statement, the amount of free pages reaches the threshold where dirty pages are flushed to disk in batches of tunable size (around 35 pages in our case, as the actual number of free pages is taken into account when submitting the batch of writes).

This is typical of a situation where there is memory pressure: There are steady flows of (sequential or random) read and (random) write requests. In order to achieve a trade-off between throughput and latency, the storage manager 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. As we saw in Section 2 though, a mix of sequential and random requests achieves a satifactory throughput even if 50% of the submitted requests are random. The storage manager should thus give the kernel a chance to intertwine read and write requests.

3.3 Design Proposals

InnoDB does not utilize asynchronous I/O as well as it could. Oracle does a better job for scan and writes, but is still limited for random reads. Unix databases both systems were initially designed for synchronous I/O. By contrast, SQLServer was designed for asynchronous I/O and offers solutions that match the performance problems we pointed out [23].
We propose that InnoDB implements the following mechanisms (including tunable parameters) to improve its utilization of asynchronous I/O:

- *Prefetching.* 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).

- *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 [23]).

- *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. The problem is that submitting large batches of random writes might interfere with other I/O requests. 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. This underlines the need for prioritized I/O in the disk scheduler.

4 The Case for Prioritized I/O

Asynchronous I/O empowers the kernel disk scheduler to optimize its utilization of the available disk bandwidth. The disk scheduler takes the decision of submitting requests based on (a) the placement of data on disk and on (b) the deadline associated to each request.
The only possibility for the database server to control the scheduling of requests is to control their submission. As we saw in Section 3, this mode of control is sub-optimal. The database server should be able to provide the disk scheduler with information so that it can take the best decision. Ideally, I/O that block the query thread should be scheduled as fast as possible, and in any case before I/O performed in the background. This could be achieved if the database server could associate priorities to the I/O requests it submits. We consider the following notion of priority:

- Each request is associated with an *absolute deadline*. This would be useful to control the trade-off between throughput and latency when prefetching pages during a scan. This is implemented by defining a set of priority levels. Each priority level is associated to an absolute deadline.

- Each request is associated with a *relative priority*, i.e., requests blocking the query thread have a higher priority than requests that do not. This is implemented by introducing *priority classes* and a scheduling mechanism ensuring that (i) requests from a given priority class are scheduled before requests from a lower priority class and that (ii) requests from low priority classes do not starve.

The Linux disk scheduler can be modified to account for priorities as follows. Instead of assigning deadline based on fixed time offsets, the scheduler could 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.
Deadlines based on variable time offsets require some changes to the deadline scheduler. Since time offsets were fixed, the scheduler maintainted read and write requests using FIFO queues. First, the fact that time offsets vary with request priority, invalidates the use of FIFO queues. This means that the deadline queues need to be kept explicitly sorted. This can be achieved with an insertion sort on the deadline queues. However, insertion sorts require scans of

one of the deadline queues and are thus CPU intensive. CPU overhead may be reduced by using other data structures than queues. B+-trees are good candidates. Second, we need to separate requests from different priority class. This way, the device driver can schedule requests with high priority before requests with lower priority. This principle guarantees that requests from the highest priority queue are always scheduled first and that requests from the lowest priority queue are scheduled when requests in no other priority class are submitted. However, it introduces a risk of starvation for low priority queues. An approach to the starvation problem is actually to reduce the number of priority classes to three: Requests in the upper class are serviced as soon as possible, requests in the middle class are serviced as soon as there is no upper class request, requests in the lower class are submitted whenever the scheduler has nothing else to do. Starvation might still take place if there is a constant flow of requests in the upper class, but we saw that it was unlikely in the context of a database workload.

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 low priority task is able to get all the preallocated requests. It reduces the number of requests that the disk scheduler can work with and may reduce quality of the scheduling. This is why priorities should also be considered when allocating requests. A simple solution is to define allocation groups associated to priority classes. That effectively means that higher priority task can get requests preallocated for lower priority tasks. This also reduces contention on *popular* allocation groups (eg. the allocation group with the default priority). Incorporating support for priorities in the Linux disk scheduler and evaluating its benefits for database workloads is a topic for future research.

5 Related Work

Asynchronous I/O, and even prioritized I/O, are commonplace on mainframes. For example, IBM supports I/O request priorities in the context of its Enteprise Storage Server [18]. 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 [12].

Recently, McWerther et al. [17] 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 [9, 8]. Jens Axboe posted a version of the Linux anticipatory disk scheduler that supports priorities [6]. 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 expressed in Section 4.

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, 15, 3], or on Jim Gray's home page (e.g., [11, 12]).

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. [4] 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) [5]. 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. [20] 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 study how MySQL/InnoDB could take best advantage of Linux asynchronous I/O. We first established the capacity of the deadline scheduler to accumulate I/O requests and schedule them efficiently. We showed that (a) direct I/O achieve better performance than buffered I/O, that (b) the rate of submission is key for sequential I/O and that (c) for random I/O, the trade-off between throughput and latency depends on the number of outstanding requests. We added support for native Linux asynchronous I/O to InnoDB. We demonstrated that MySQL/InnoDB performs better with native asynchronous I/O than with simulated asynchronous I/O. However, our experiments show that MySQL/InnoDB does not utilize the full potential of asynchronous I/O when scanning a table, accessing a set of pages through a secondary index, or performing large amounts of writes. We proposed modifications of InnoDB to remedy these limitations. We also proposed to extend Linux asynchronous I/O with priorities in order to improve the collaboration between the database server and the disk scheduler. There is nothing revolutionary about these results. Our proposals are inspired from existing systems. We believe however that this study is an interesting contribution to the evolution of MySQL/InnoDB and Linux.

Support for native Linux asynchronous I/O is being transferred from a prototype implementation to the next release of InnoDB. Prototype implementation of our design proposal is under way at University of Copenhagen. We are also finishing the implementation of a priority based deadline scheduler for Linux 2.6.

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] Linux Kernel 2.6.3. Home Page. http://www.kernel.org/.

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

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

[4] 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.

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

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

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

[8] Suparna Bhattacharya. Personal Communication.

[9] 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.

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

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

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

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

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

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

[16] MySQL Reference Manual. InnoDB Storage Manager. http://www.mysql.com/doc/en/InnoDB.html.

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

[18] 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.

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

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

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

[22] Sistina. Logical Volume Manager. http://www.sistina.com/.

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

[24] Gaja Krishna Vaidyanatha, Kirtikumar Deshpande, and John A. Kostelac. *Oracle Performance Tuning 101*. McGraw-Hill Osborne Media, 4th edition, 2001.

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

[26] Veritas. Storage Foundation. http://www.veritas.com/.