Disk I/O in Linux

Carl Staelin
HP Laboratories Israel[1]
HPL-2002-352
December 19th , 2002*

E-mail: staelin@hpl.hp.com

disk performance, RAID, Linux, I/O performance

The paper contains preliminary results from an investigation comparing Linux and WindowsXP disk I/O using kernel-based software RAID on identical hardware. WindowsXP performance is shown to be substantially superior to Linux performance, for reasons not currently understood.

One known problem with the Linux RAID driver is that in Linux 2.4 file I/O is synchronous and file operations are serialized by a kernel file lock. Since the RAID driver is exposed to user-level applications via a file interface, this means that all accesses to the RAID are serialized.

There are a number of possibilities to improve disk I/O performance: [1] Develop a user-level RAID layer built on `aio` and extend applications to allow asynchronous I/O. [2] Attempt to debug the Linux software RAID device so its synchronous performance is comparable to WindowsXP.

We developed a simple user-level RAID layer built on `aio`, and demonstrate its improved performance over the Linux software RAID implementation, but its performance is still lower than that of the WindowsXP software RAID implementation.

The alternative is to investigate various options for extending or improving the Linux kernel, which is likely to be a riskier and more difficult task.

* Internal Accession Date Only
Approved for External Publication
1 Technion City, Haifa, 32000, Israel

Disk I/O in Linux

Carl Staelin

Hewlett-Packard Laboratories Israel
Technion City
Haifa, Israel
staelin@hpl.hp.com
http://www.hpl.hp.com/personal/Carl_Staelin

ABSTRACT

The paper contains preliminary results from an investigation comparing Linux and WindowsXP disk I/O using kernel-based software RAID on identical hardware. WindowsXP performance is shown to be substantially superior to Linux performance, for reasons not currently understood.

One known problem with the Linux RAID driver is that in Linux 2.4 file I/O is synchronous and file operations are serialized by a kernel file lock. Since the RAID driver is exposed to user-level applications via a file interface, this means that all accesses to the RAID are serialized.

There are a number of possibilities to improve disk I/O performance:

[1] Develop a user-level RAID layer built on `aio` and extend applications to allow asynchronous I/O.

[2] Attempt to debug the Linux software RAID device so its synchronous performance is comparable to WindowsXP.

We developed a simple user-level RAID layer built on `aio`, and demonstrate its improved performance over the Linux software RAID implementation, but its performance is still lower than that of the WindowsXP software RAID implementation.

The alternative is to investigate various options for extending or improving the Linux kernel, which is likely to be a riskier and more difficult task.

1. Introduction

The goal of this research is to identify the parameters of Linux's I/O performance, and to try to find a software solution which provides sufficient I/O bandwidth with minimal development overhead and risk.

I/O patterns during real use will not be purely streaming sequential I/O. Instead they will typically be *chunked*, with randomly dispersed chunks of varying sized sequential I/O. A new benchmark was developed to measure this type of I/O activity in a controlled fashion.

1.1. Hardware

All the experiments were run on the same PC, an HP x4000 PC workstation with a 2.4GHz P4 Xeon processor and 512MB of RDRAM. There were two removable IDE boot disks, one containing WindowsXP and one containing Mandrake 8.2 Linux with kernel version 2.4.18-6mdk. In addition, there were three Fujitsu MAN3367MP 33GB 10,000RPM disks [3] on a single Ultra160MB/s SCSI string for testing. The SCSI adapter is an Adaptec aic7895 UltraSCSI adapter on a PCI 32/33 bus.

The Linux tests were run under Mandrake 8.2 with the stock Mandrake Linux 2.4.18-6mdk kernel. The tests on the kernel-based RAID implementation used the `md` driver configured as a RAID0 device.

The WindowsXP RAID implementation was the kernel-based `stripe set`, which appears to have a stripe size of 1MB.

2. Benchmark

We wrote a custom benchmark based on the `lmbench` timing harness which measures read and write bandwidth to both a single RAID device containing a variable number of disks, and direct access to a variable number of disks. The benchmark can measure both streaming, sequential I/O, and random I/O in fixed chunk sizes. The results reported here are for random I/O in various chunk sizes in order to estimate the performance of the I/O subsystem on various element sizes. When measuring RAID performance, it varies the RAID stripe size to evaluate its effect on performance.

The results reported here are for random, fixed size I/O requests. The reason is that sequential I/O has a significantly different character than random I/O since the system can pre-fetch data to improve performance. However, that same prefetching that improves sequential performance can reduce the performance of chunked or random I/O since the system spends resources reading data that is never accessed. The I/O request sizes vary from 4KB to 64MB, to mimic a range of element sizes. Before each I/O the system randomly chooses a segment to access. For large requests, the I/O behavior is largely sequential and will more closely mimic that of sequential I/O.

When measuring the performance of bundles of independent disks, sometimes known as Just a Bunch of Disks (JBOD), the system creates a process for each disk.

When measuring the performance of the software RAID disks, there is only a single thread which accesses the disk. The default RAID stripe size is 512KB under Linux and apparently 1MB under WindowsXP. On Linux, the RAID stripe size is varied from 4KB to 1MB to explore the affect of stripe size on performance.

3. Asynchronous I/O

I/O subsystems and devices generally perform better when given multiple requests in parallel. For example, a RAID device contains multiple disks, and any given I/O request is generally sent to a single disk, leaving the remaining devices idle. Giving the RAID device multiple requests can potentially allow the RAID device to send independent requests to different devices, achieving much better overall utilization and performance.

Since devices are represented by files in Linux, this means that it is potentially important to be able to send multiple outstanding requests to a single file.

The Linux I/O interfaces, such as `read` and `write`, are inherently synchronous. The calling thread is blocked until the I/O is complete. There is a standard POSIX interface, `aio`, which defines a suite of routines that can be used for asynchronous I/O. Currently, these routines are provided by a user-level library, `librt`, which uses threads to send multiple I/O requests to the kernel. Unfortunately, this implementation queues all I/O requests for a given file on a single thread, so all I/O for a single file is serialized by that thread.

In addition, the 2.4 Linux kernel currently serializes all requests for a single file since each I/O request locks the file datastructure for the duration of the operation. This means that requests to a given file will be serialized. In particular, applications which directly access devices will have all their requests serialized.

The Linux kernel developers are aware of this problem, and asynchronous block I/O is now the default in the Linux 2.5 (development) kernel. All intra-kernel file operations are now asynchronous operations, with an externally visible system call interface [1]. This would imply that even using the existing synchronous interfaces, independent I/O requests from various threads and processes could potentially operate on the same file in parallel. In addition, the POSIX `aio` interface would also allow multiple requests to the same file to proceed in parallel.

3.1. Intra-disk parallelism

Disk drives have two mechanisms to provide a level of parallelism in their activity: request queues and read-ahead buffering. Most SCSI disks, and increasingly IDE disks, provide a means for the host to queue up several I/O requests called *tagged-command-queueing* (TCQ), which allows the device to schedule them optimally given its detailed knowledge of the state of the device, such as its rotational position, head position, and seek-time costs. The Linux SCSI driver appears to support TCQ, but it is not clear if it is fully utilized due to serialization of requests via file locking.

When a disk head is sent to a region, or when a read request completes, many disks will read subsequent data from the disk and put it into a read-ahead buffer if there are no queued requests, in case the host asks for it in the near future. This

mechanism allows disks to stream data to the host at platter speeds.

Unfortunately, given the limitations of the current Linux kernel, it is not currently possible to explicitly send multiple requests to the same device when directly accessing the raw device.

4. User-level software RAID driver (CSRAID)

Due to the various limitations on available parallelism through the kernel software RAID driver, a user-level software RAID driver called `csraid` was written using `aio`. Since the user-level driver accesses each disk's device file directly, it can send requests to each device independently. This gives a maximal theoretic performance comparable to JBOD.

`csraid` has two sets of interfaces, one synchronous and the other asynchronous. The asynchronous interface returns as soon as the request(s) have been submitted to `aio`, while the synchronous interface waits for those requests to complete. Note that large I/O requests will generally result in multiple I/O requests being sent to the disks. In both cases these requests are handled asynchronously via `aio` so even the synchronous interface provides some level of parallel disk activity.

5. Results

The investigation is a multi-stage process. The first step is to identify the maximal attainable bandwidth by directly accessing the disks, without the software RAID layer, using chunk-wise sequential I/O. The next step is to bundle the disks into a software RAID device, and to measure strictly sequential I/O over various RAID stripe sizes. The next step is to measure performance using a chunked workload over a variety of chunk sizes.

The performance results were taken using a custom benchmark built using the `lmbench` version 3[2] timing harness which can measure performance under scalable load. At each point the system took 101 samples per load element. All RAID measurements were taken with a single process issuing requests serially, which is a single load element. The JBOD measurements used a process per disk, so the number of load elements is the number of disks. In this case the number of measurements taken per point is $101 \times ndisks$. The graphs show the Q1-Q3 confidence interval for each point, as well as the median value.

5.1. Just-a-bunch-of-disks (JBOD)

Since the various software RAID devices' performance is limited by the underlying hardware performance, we begin by analyzing the performance available when accessing the various disks directly.

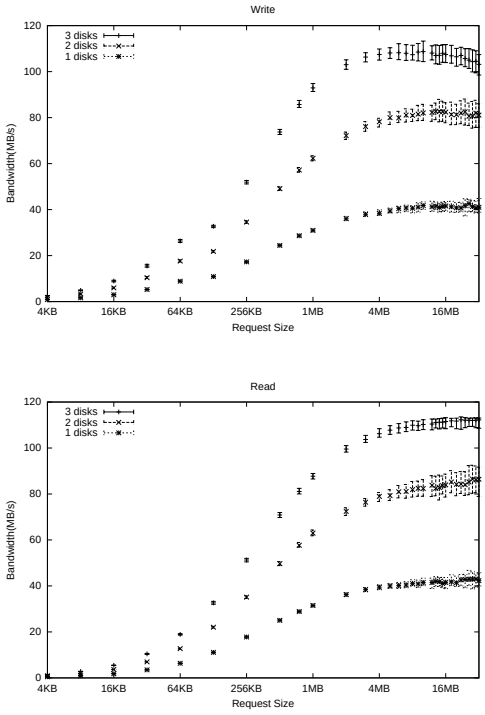

Figure 1. JBOD bandwidth vs request size

Figure 1 shows how aggregate disk bandwidth improves as the number of disks and the request size increase. Note that the peak available bandwidth per disk is roughly 40MB/s, and that this is obtained with I/O requests as small as 4MB. This verifies that under Linux, the maximal I/O bandwidth with this hardware configuration is nearly 120MB/s.

5.2. WindowsXP software RAID device

We measured the Windows software RAID device performance to obtain baseline comparison figures.

Figure 2 shows WindowsXP's RAID performance over a range of request sizes for one, two, and three disks. Note that WindowsXP's RAID performance for large requests is actually superior to the Linux JBOD performance, which is somewhat surprising.

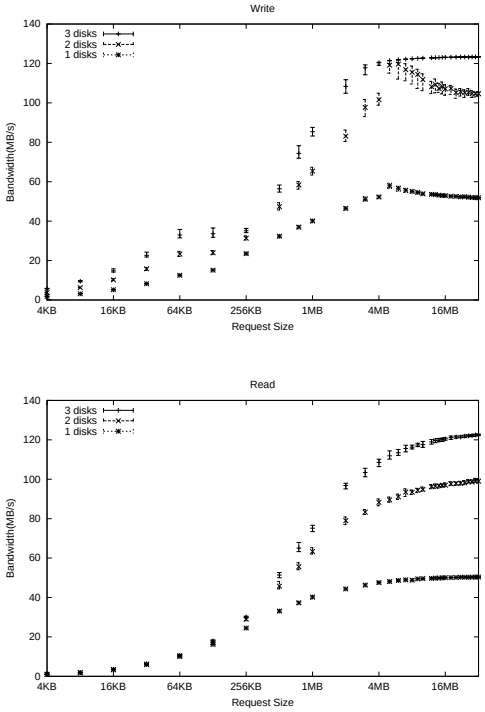

Figure 2. WindowsXP RAID bandwidth vs request size

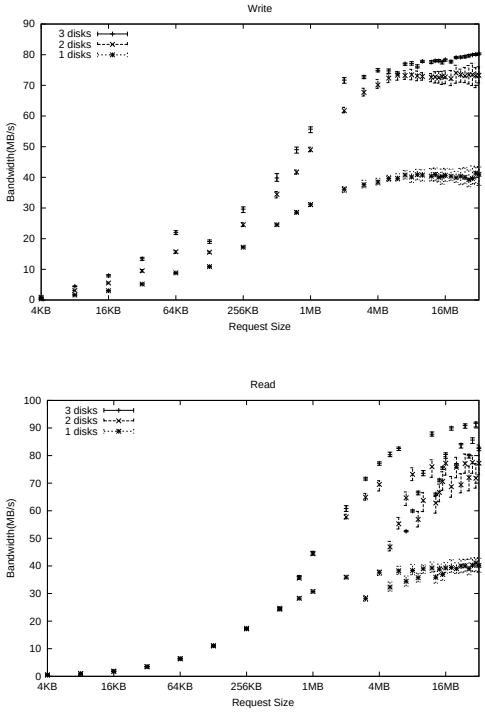

Figure 3. Linux RAID bandwidth vs request size

5.3. Linux software RAID device

The Linux software RAID device was evaluated on both sequential streaming I/O and random chunked I/O patterns. In addition, its performance with various RAID stripe sizes was evaluated.

To get a somewhat realistic assessment of expected performance under real workloads, we must use *chunked* I/O patterns, where each chunk is read sequentially, but chunks are placed randomly across the device. In this case, the implementation details of features such as read-ahead can have a substantial impact on the resulting performance. For example, if the read-ahead algorithm is overly optimistic, it can recognize the chunk read as sequential and schedule read-ahead operations for a substantial amount of data beyond the end of each chunk, resulting in gross waste of bandwidth. Alternatively, if the RAID does not send I/Os to other disks at the start of each chunk, then each of the first *N* reads will be strictly sequential, causing unnecessary serialization of the I/Os.

Figure 3 shows how aggregate disk bandwidth improves as the number of disks and the request size increase when the stripe size is 512KB. Unfortunately, the peak bandwidth is 3/4 the peak disk or sequential RAID bandwidths from Figures 1. This result is somewhat disturbing because it implies that the I/O subsystem is not making full use of the available parallelism when large I/Os are sent as a single request.

Figure 4 shows how aggregate disk bandwidth versus stripe size for three disks with two request sizes, 1MB and 16MB. The relatively flat curves indicate that between 64KB and 512KB there is not much difference between stripe sizes, although 128KB seems to be slightly preferable for smaller chunk sizes.

5.4. CSRaid

Given the atrocious performance of the kernel-based software RAID device, an alternative user-level RAID ‘‘device’’, `csraid`, was developed at HPL-Israel. The device was implemented using the `aio` interface, which allows it to asynchronously access multiple disks in parallel. However, the current limitations of the Linux `aio` system prevent the device from submitting multiple I/O requests to a single disk in parallel.

The device allows both asynchronous and synchronous I/O. The primary benefit of asynchronous I/O with the current `aio` implementation would be that independent RAID requests might access different disks, especially when the

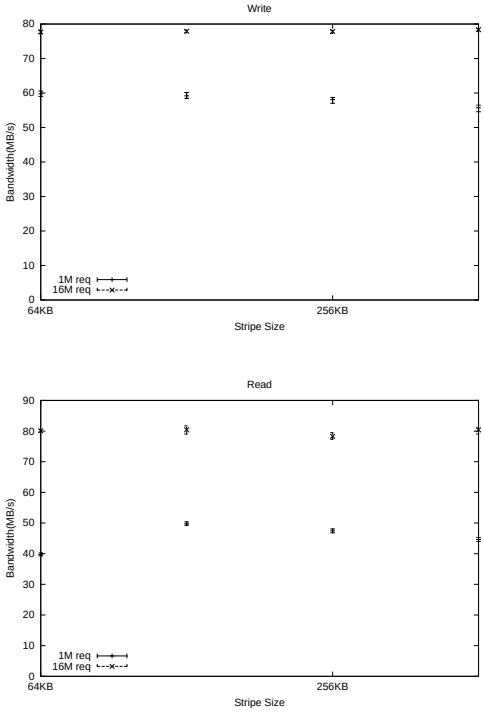

Figure 4. Linux RAID bandwidth vs stripe size (3 disk RAID)

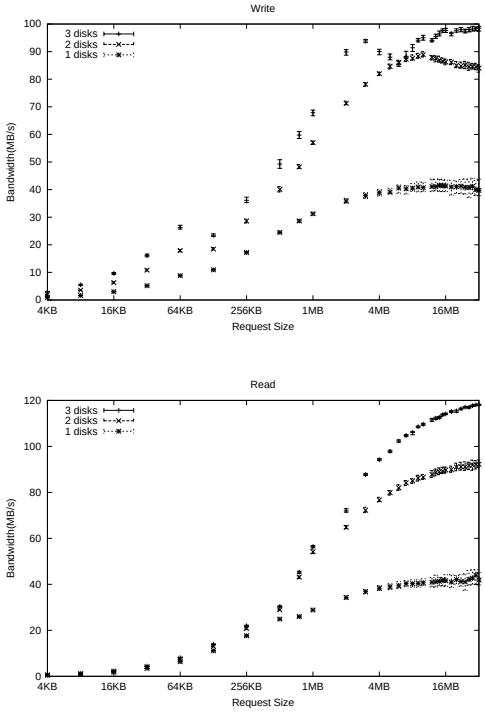

Figure 5. CSRAID synchronous bandwidth vs request size

request size is comparable to or smaller than the RAID stripe size.

Figure 5 shows the performance for synchronous I/O through the `csraid` interface.

Figure 6 also shows improved performance over the kernel-based RAID driver with asynchronous request handling.

Figure 7 shows that smaller stripe sizes can have performance benefits, and that stripe sizes of larger than the chunk size can harm performance. So smaller stripe sizes are recommended for `csraid`.

5.5. Direct comparison

Of crucial interest is the direct comparison between WindowsXP, the kernel-based Linux software RAID implementation, and the user-level software RAID implementation.

From Figure 8 it can be seen that `csraid` generally outperforms the kernel-based RAID driver, and its performance is far more stable, particularly for large requests. Also, the WindowsXP RAID driver is astonishingly good, even compared to JBOD, with JBOD's performing better for request sizes smaller than 4MB and WindowsXP performing better at the larger sizes.

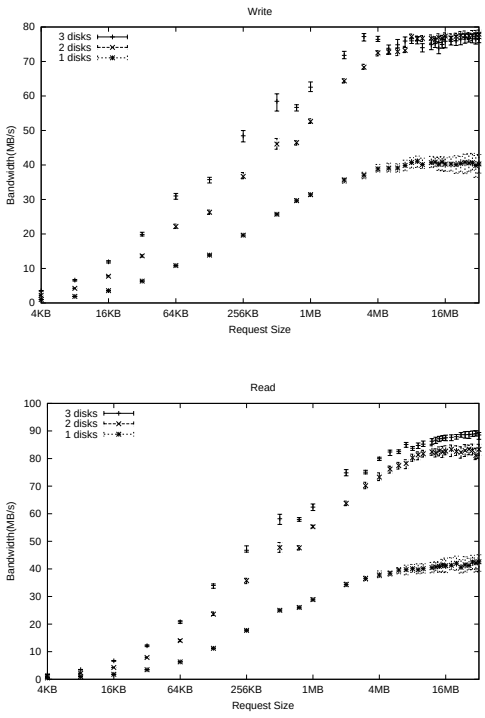

Figure 6. CSRAID asynchronous bandwidth vs request size

`csraid`'s performance is generally less than

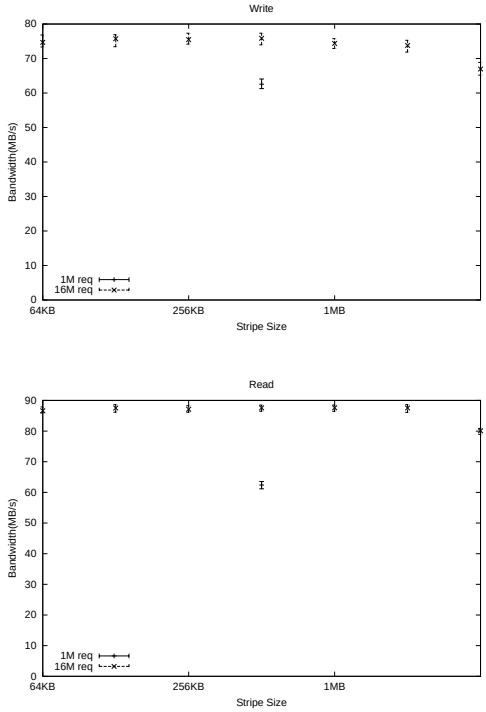

Figure 7. CSRAID asynchronous bandwidth vs stripe size (3 disk RAID)

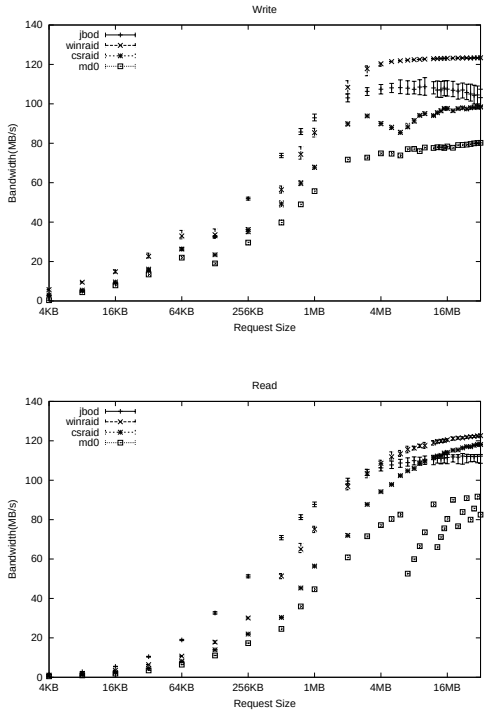

Figure 8. WindowsXP, RAID, and CSRAID bandwidth vs request size

WindowsXP, but overall the loss is not catastrophic, and is likely sufficient for the most purposes.

6. Conclusions and future work

Performance evaluation of the Linux kernel-based software RAID device under somewhat realistic conditions indicate substantially reduced performance compared to the equivalent WindowsXP software RAID implementation. This leaves two options: find a way to use the existing kernel so performance is acceptable, or fix the Linux kernel so performance is comparable.

The exact reasons for relatively poor performance of the kernel-based software RAID device, as shown in Figure 3 are currently unknown. A more detailed analysis of device utilization in the chunked I/O case would be required, including disk trace analysis, if possible. Depending on the level and quality of analysis tools available under Linux, this might be a relatively short project, or it might entail modifying the kernel to dump a trace of the necessary operations.

Tuning the software RAID implementation would require some effort to understand the Linux memory management and buffer management system, as well as the various read-ahead policies and implementation. It would also involve kernel hacking and could take as long as a few months to complete. Until that kernel survey is complete, it won't be clear whether changes to the software RAID driver are likely to result in performance increases, because the elements causing the problems might reside in other parts of the kernel.

References

1. LWN.net, *LWN: The Ottowa Kernel Summit, Day Two*, Ottowa, Canada (June 25, 2002). http://lwn.net/Articles/3467.

2. Larry McVoy and Carl Staelin, *lmbench: Portable tools for performance analysis*, pp. 279-284, Proceedings USENIX Winter Conference (January 1996).

3. Tom's Hardware Guide, *Fujitsu MAN3367MP - A gentle SCSI-runner* (January 2002). http://www.tomshardware.com/storage/02q1/020117.