Exploring High Bandwidth Filesystems on Large Systems

Dave Chinner and Jeremy Higdon
Silicon Graphics, Inc.
dgc@sgi.com jeremy@sgi.com

Abstract

In this paper we present the results of an investigation conducted by SGI into streaming filesystem throughput on the Altix platform with a high bandwidth disk subsystem.

We start by describing some of the background that led to this project and our goals for the project. Next, we describe the benchmark methodology and hardware used in the project. We follow this up with a set of baseline results and observations using XFS on a patched 2.6.5 kernel from a major distribution.

We then present the results obtained from XFS, JFS, Reiser3, Ext2 and Ext3 on a recent 2.6 kernel. We discuss the common issues that we found to adversely affect throughput and reproducibility and suggest methods to avoid these problems in the future.

Finally, we discuss improvements and optimisations that we have made and present the final results we achieved using XFS. From these results we reflect on the original goals of the project, what we have learnt from the project and what the future might hold.

1 Background and Goals

In the past, there have been many comparisons of the different filesystems suppported by Linux. Most of these comparisons focus on activities typically performed by a kernel developer or use well known benchmark programs. Typically these tests are run on an average desktop machine with a single disk or, more rarely, a system with two or four CPUs with a RAID configuration of a few disks.

However, this really doesn't tell us anything about the maximum capabilities of the filesystems; these machine configurations don't push the boundaries of the filesystems and hence these observations have little relevance to those who are trying to use Linux in large configurations that require substantial amounts of I/O.

Over the past two years, we have seen a dramatic increase in the bandwidth customers require new machines to support. On older, modified 2.4.21 kernels, we could not achieve much more than 300MiB/s on parallel buffered write loads. Now, on patched 2.6.5 kernels, customers are seeing higher than 1GiB/s under the same loads. And, of course, there are customers who simply want all the I/O bandwidth we can provide.

The trend is unmistakable. A coarse correlation is that required I/O bandwidth matches the

amount of memory in a large machine. Memory capacity is increasing faster than physical disk transfer rates are increasing, and this means that systems are being attached to larger numbers of disks in the hope that this provides higher throughput to populate and drain memory faster. Unfortunately, what we currently lack is any data on whether Linux can make use of the increased bandwidth that larger disk farms provide.

Some of the questions we need to answer include:

- How close to physical hardware limits can we push a filesystem?
- How stable is Linux under these loads?
- How does the Linux VM stand up to this sort of load?
- Do the Device Mapper (DM) and/or Multiple Device (MD) drivers limit performance or configurations?
- Are there NUMA issues we need to address?
- Do we have file fragmentation problems under these loads?
- How easily reproducible are the results we achieved and can we expect customers to be able to achieve them?
- What other bottlenecks limit the performance of a system?

To answer these questions, as they are important to SGI's customers, we put together a modestly sized machine to explore the limits of high-bandwidth I/O on Linux.

2 Test Hardware and Methodology

2.1 Hardware

The test machine was an Altix A3700 containing 24 Itanium2 CPUs running at 1.5GHz in 12 nodes in a single cache-coherent NUMA domain. Each node is configured with 2GiB of RAM for a system total of 24GiB. Each node has 6.4GB/s peak full duplex external interconnect bandwidth provided by SGI's NUMALink interconnect. A total of 12 I/O nodes, each with three 133MHz PCI-X slots on two busses, were connected to the NUMALink fabric supplying 6.4GB/s peak full duplex bandwidth per I/O node. The CPU and I/O nodes were connected via crossbar routers in a symmetric topology.

The I/O nodes were populated with a mix of U320 SCSI and Fibre Channel HBAs (64 SCSI controllers in total) and distributed 256 disks amongst the controllers in JBOD configuration. This provided an infrastructure that allowed each disk run at close to its maximum read or write bandwidth independently of any other disk in the machine.

The result is a machine with a disk subsystem theoretically capable of just over 11.5GiB/s of throughput evenly distributed throughout the NUMALink fabric. Hence the hardware should be able to sustain maximum disk rates if the software is able to drive it that fast.

2.2 Methodology

The main focus of our investigation was on XFS performance. In particular, parallel sequential I/O patterns were of most interest as these are the most common patterns we see our customers using on their large machines. We also assessed how XFS compares with other mainstream filesystems on Linux on these workloads.

The main metrics we used to compare performance were aggregate disk throughput and CPU usage. We used multiple programs and independent test harnesses to validate the results against each other so we had confidence in the

results of individual test runs that weren't replicated.

To be able to easily compare different configurations and kernels, we present normalised I/O efficiency results along with the aggregate throughput achieved. This gives an indication of the amount of CPU time being expended for each unit of throughput achieved. The unit of efficiency reported is `% CPU/MiB/s`, or the percentage of a CPU consumed per mebibyte per second throughput. The lower the calculated number, the better the efficiency of the I/O executed.

The tests were run with file sizes large enough to make run times long enough to ensure that measurement was accurate to at least 1%. This, combined with running the tests in a consistent (scripted) manner, enabled us to draw conclusions about the reproducibility of the results obtained.

For most of the tests run, we used SGI's Performance Co-Pilot infrastructure [PCP] to capture high resolution archives of the system's behaviour during tests. This included disk utilisation and throughput, filesystem and volume manager behaviour, memory usage, CPU usage, and much more. We were able to analyse these archives after the fact which gave us great insight into system wide behaviour during the testing.

To find the best throughput under different conditions, we varied many parameters during testing. These included:

- different volume configurations
- the effect of I/O size on throughput and CPU usage
- buffered I/O and direct I/O
- different allocation methods for writes
- block device readahead
- filesystem block size
- pdflush tunables
- NUMA allocation methods

We tested several different kernels so we could chart improvements or regressions over time that our customers would see as they upgraded. Hence we tested XFS on SLES9 SP2, SLES9 SP3, and 2.6.15-rc5.

We also ran a subset of the above tests on other Linux filesystems including Ext2, Ext3, ReiserFS v3, and JFS. We kept as many configuration parameters as possible constant across these tests. Where supported, we used mkfs and mount parameters that were supposed to optimise data transfer rates and large filesystem performance.

The volume size for Ext2, Ext3 and ReiserFS V3 was halved to approximately 4.2TiB because they don't support sizes of greater than 8TiB. We took the outer portion of each disk for this smaller volume, hence maintaining the same stripe configuration. Compared to the larger volume used by XFS and JFS, the smaller volume has lower average seek times and higher minimum transfer rates and hence should be able to maintain higher average throughputs than the larger volume as the filesystems fill up during testing.

The comparison tests were scripted to:

1. Run mkfs with relevant large filesystem optimisations.
2. Make a read file set with dd by writing out the files to be read back with increasing levels of parallelism.
3. Perform buffered read tests using one file per thread across a range of I/O sizes and thread count measuring throughput, CPU usage, average process run time and other metrics required for analysis.

The filesystem was unmounted and remounted between each test to ensure that all tests started without any cached filesystem data and memory approximately 99% empty.

4. Repeat Step 3 using buffered write tests, including truncating the file to be written in the overall test runtime.

Parallel writes were used to lay down the files for reading back to demonstrate the level of file fragmentation the filesystem suffered. The greater the fragmentation, the more seeking the disks will do and the lower the subsequent read rate achieved will be. Hence the read rate directly reflects on the fragmentation resistance of the filesystem. This is also a best case result because the tests are being run on an empty filesystem.

Finally, after we fixed several of the worst problems we uncovered, we re-ran various tests to determine the effect of the changes on the system.

2.3 Volume Layout and Constraints

Achieving maximum throughput from a single filesystem required a volume layout that enabled us to keep every disk busy at the same time. In other words, we needed to distribute the I/O as evenly as possible.

Building a wide stripe was the easiest way to achieve even distribution since we were mostly interested in sequential I/O performance. This exposed a configuration limitation of DM; `dmsetup` was limited to a line length of 1024 characters which meant we could only build a stripe approximately 90 disks wide.

Hence we ended up using a two level volume configuration where we had an MD stripe of 4 DM volumes each with 64 disks. We used an MD stripe of DM volumes because it was unclear whether DM and `dmsetup` supported multi-level volume configurations.

Using SGI's XVM volume manager, we were able to construct both a flat 256 disk stripe and a 4x64 disk multi-level stripe. Hence we were able to confirm that there was no measurable performance or disk utilisation difference between the two configurations.

Therefore we ran all the tests on the multi-level, MD-DM stripe volume layout. The only parameter that was varied in the layout was the stripe unit (and therefore stripe width) and most of the testing was done with stripe units of 512KiB or 1MiB.

3 Baseline XFS Results

Baseline XFS performance numbers were obtained from SuSE Linux Enterprise Server 9 Service Pack 2 (SLES9 SP2). We ran tests on XFS filesystem with both 4KiB and 16KiB block sizes. Performance varied little with I/O size, so the results presented used 128KiB, which is in the middle of the test range.

Looking at read throughput, we can see from Figure 1 that there was very little difference between the different XFS filesystem configurations. In some cases the 16KiB block size filesystem was faster, in other cases the 4KiB block size filesystem was faster. Overall, they both averaged out at around 3.5GiB/s across all block sizes.

In contrast, the 16KiB block size filesystem is substantially faster than the 4KiB filesystem when writing. The 4KiB filesystem appeared to be I/O bound as it was issuing much smaller I/Os than the 16KiB filesystem and the disks were seeking significantly more.

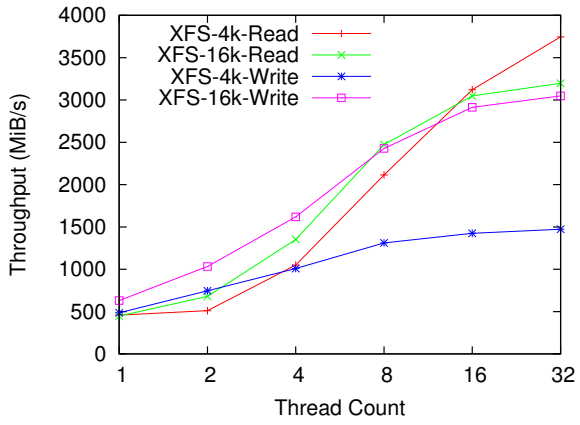

Figure 1: Baseline XFS Throughput.

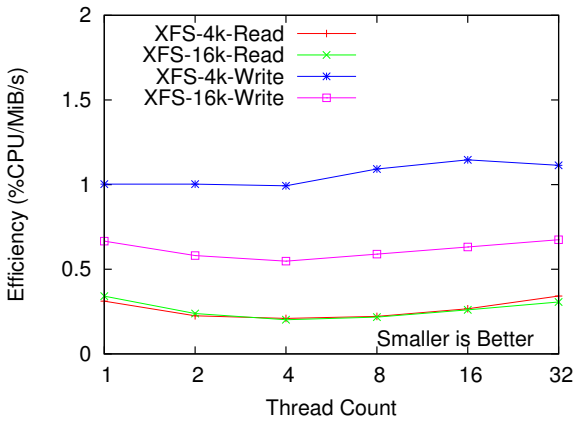

Figure 2: Baseline XFS Efficiency.

From the CPU efficiency graph in Figure 2, we can see that there is no difference in CPU time expended by the filesystem for different block sizes on read. This was expected from the throughput results.

Both the read and write tests show that CPU usage is scaling linearly with throughput; increasing the number of threads doing I/O does not decrease the efficiency of the filesystem. In other words, we are limited by either the rate at which we can issue I/Os or by something else outside the filesystem. Also, the write efficiency is substantially worse than for reads, it would seem that there is room for substantial improvement here.

4 Filesystem Comparison Results

The first thing to note about the results is that some of the filesystems were tested to higher numbers of threads and larger block sizes. The reasons for this were that some configurations were not stable enough to complete the whole test matrix and we had to truncate some of the longer test runs that would have prevented us from completing a full test cycle in our available time window. Consequently some of the results presented represent best-case performance rather than a mean of repeated test runs.

The kernel used for all these tests was 2.6.15-rc5.

4.1 Buffered Read Results

The maximum read rates achieved by each filesystem can be seen in Figure 3. The read rate changed very little with varying I/O block size, we saw the same maximum throughput using 4KiB I/Os as using 1MiB I/Os. The only real difference was the amount of CPU consumed.

It is worth noting that XFS read throughput is substantially higher on 2.6.15-rc5 compared to the baseline results on SLES9 SP2. A discussion of this improvement canbe found in Section 6.2.

The performance of Ext2 and Ext3 is also quite different despite their common heritage. However, the results presented for Ext2 and Ext3 (as well as JFS) are the best of several test executions due to the extreme variability of the filesystem performance under these tests. The

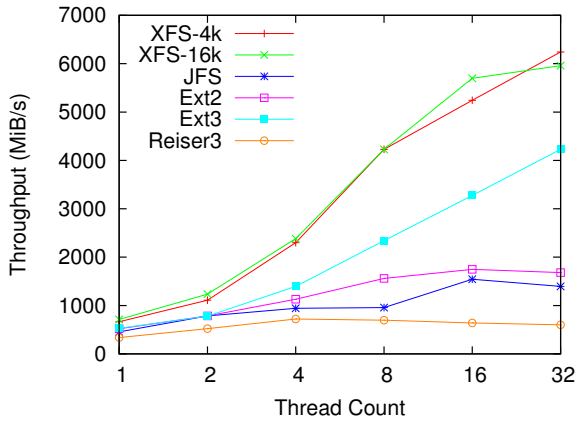

Figure 3: Buffered Read Throughput Comparison.

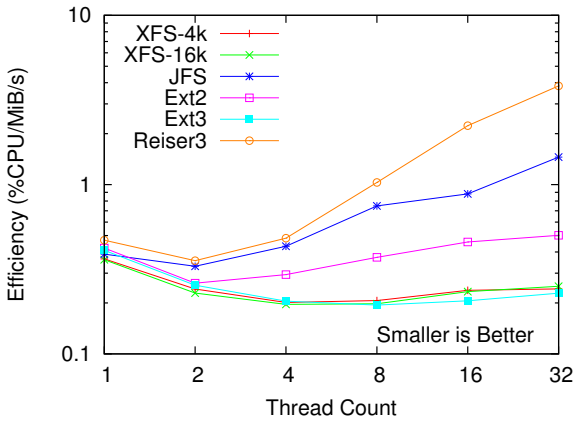

Figure 4: Buffered Read Efficiency Comparison.

reasons for this variability are discussed in Section 5.2.

It is clear that XFS and Ext3 give substantially better throughput, and this is reflected in the efficiency plots in Figure 4, where these are the most efficient filesystems. Both ReiserFS and JFS show substantial decreases in efficiency as thread count increases. This behaviour is discussed in Section 5.1.

4.2 Buffered Write Results

Figure 5 shows some very clear trends in buffered write throughput. Firstly, XFS is substantially slower than the SLES9 SP2 baseline results. Secondly, throughput is peaking at four to eight concurrent writers for all filesystems except for Ext2. XFS, using a 16KiB filesystem block size, was still faster than Ext2 until high thread counts were reached.

The poor write throughput of Ext3 and JFS is worth noting. JFS was unable to exceed an average of 80MiB/s write speed in all but two of the many test points executed, and Ext3 did not score above 250MiB/s and decreased to less than 100MiB/s at sixteen or more threads. We used the `data=writeback` mode for Ext3 as it was consistently 10% faster than the `data=ordered` mode.

The ReiserFS results are truncated due to problems running at higher thread counts. Writes would terminate without error unexpectedly, and sometimes the machine would hang. Due to time constraints this was not investigated further, but it is suspected that buffer initialisation problems which manifested on machines with both XFS and ReiserFS filesystems were the cause. The fixes did not reach the upstream kernel until well after testing had been completed[Scott][Mason].

JFS demonstrated low write throughput. We discovered that this was partially due to truncating a multi-gigabyte file taking several minutes to execute. However, the truncate time made up only half the elapsed time of each test. Hence, even if we disregarded the truncate time, JFS would still have had the lowest sustained write rate of all the filesystems.

Looking at the efficiency graph in Figure 6, we can see that only JFS and Ext2 had relatively flat profiles as the number of threads increased. However, the profile for JFS is relatively meaningless due to the low throughput. All the other filesystems show decreas-

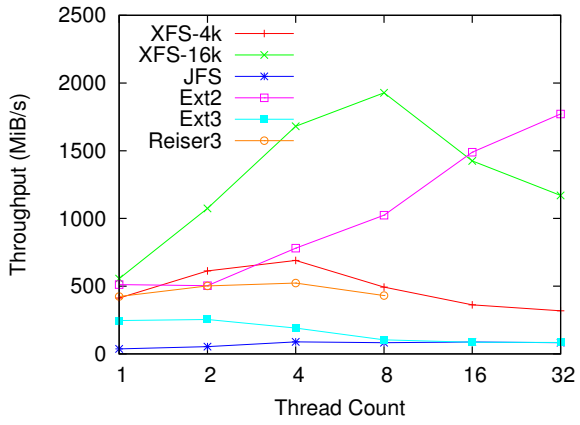

Figure 5: Buffered Write Throughput Comparison.

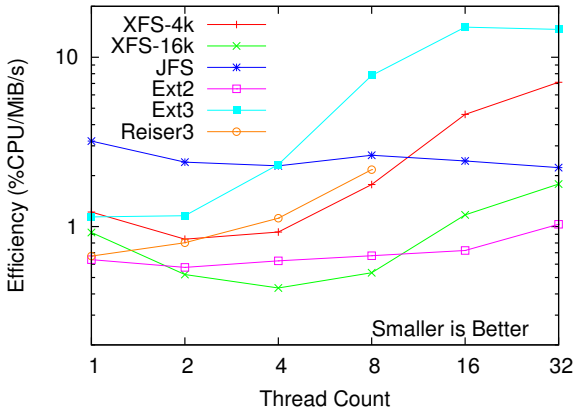

Figure 6: Buffered Write Efficiency Comparison.

ing efficiency (increasing CPU time per MiB transferred to disk every second) at the same load points that they also showed decreasing throughput. This is discussed further in Section 5.1.

4.3 Direct I/O Results

Only XFS and Ext3 were compared for direct I/O due to time constraints. The tests were run over different block sizes and thread counts, and involved first writing a file per thread, then overwriting the file, and finally reading the file back again. A 512KiB stripe unit was used for these tests.

Table 1 documents the maximum sustained throughput we achieved with these tests. Ext3 was fastest with only a single thread, but writes still fell a long way behind XFS. As the number of threads increased, Ext3 got slower and slower as it fragmented the files it was writing. At 18 threads, Ext3 direct I/O performance was between 10 and 20 times lower than for a single thread.

Threads	FS	Read	Write	Overwrite
1	XFS	5.5	4.0	7.5
1	Ext3	4.2	0.6	2.5
18	XFS	10.0	7.7	7.7
18	Ext3	0.58	0.06	0.12

Table 1: Sequential Direct I/O Throughput (GiB/s)

In contrast, from 1 to 18 threads, XFS doubled its read and write throughput, and overwrite increased marginally from its already high single thread result. It is worth noting that the XFS numbers peaked substantially higher than the sustained throughput - reads peaked at above 10.7GiB/s, while writes and overwrites peaked at over 8.9GiB/s.

5 Issues Affecting Throughput

5.1 Spinlocks in Hot Paths

One thing that is clear from the buffered I/O results is that global spinlocks in hot paths of a filesystem do not scale. Every journalled filesystem except JFS was limited by spinlock contention during parallel writes. In the case of JFS, it appeared to be some kind of sleeping contention that limited performance, and so the impact of contention on CPU usage was not immediately measurable. Both ReiserFS and JFS

displayed symptoms of contention in their read paths as well.

From analysis of the contention on the XFS buffered write path, we found that the contended lock was not actually being held for very long. The fundamental problem is the number of calls being made. For every page we write on a 4KiB filesystem, we are allocating four filesystem blocks. We do this in four separate calls to `->prepare_write()`. Hence at the peak throughput of approximately 700MiB/s, we are making roughly 180,000 calls per second that execute the critical section.

That gives us less than 5.6 microseconds to obtain the spinlock and execute our critical section to avoid contention. The code that XFS executes inside this critical section involves a function call, a memory read, two likely branches, a subtraction and a memory write. That is not a lot of code, but with enough CPUs trying to execute it in parallel it quickly becomes a bottleneck.

Of all the journalling filesystems, XFS appears to have the smallest global critical section in its write path. Filesystems that do allocation in the write path (instead of delaying it until later like XFS does) can't help but have larger critical sections here, and this shows in the throughput being achieved.

Looking to the future, we need to move away from allocating or mapping a block at a time in the generic write path to reduce the load on critical sections in the filesystems. While work is being done to reduce the number of block mapping calls on the read path, we need to do the same work for the write path. In the meantime, we have solved XFS's problem in a different way. (see Section 6.1.2)

5.2 File Fragmentation and Reproducibility

From observation, the main obstacle in obtaining reproducible results across multiple test runs on each filesystem was file fragmentation. XFS was the only filesystem that almost completely avoided fragmentation of its working files. ReiserFS also seemed to be somewhat resistant to fragmentation but the results are not conclusive due to the problems ReiserFS had writing files in parallel.

Ext2, Ext3 and JFS did not resist fragmentation at all well. From truncated test results, we know that the variation was extreme. A comparison of the best case results versus the worst case results for ext2 can be seen in Table 2. Both Ext3 and JFS demonstrated very similar performance variation due to the different amounts of fragmentation of the files being read in each test run. While we present the best numbers we achieved for these filesystems, you should keep in mind that these are not consistently reproducible under real world conditions.

At the other end of the scale, the XFS results were consistently reproducible to within $\pm$3%. This is due to the fact that we rarely saw fragmentation on the XFS filesystems and the disk allocation for each file was almost identical on every test run. Even when we did see fragmentation, the contiguous chunks of file data were never smaller than several gigabytes in size.

A further measure of fragmentation we used was the number of physical disk I/Os required to provide the measured throughput. In the case of XFS, we were observing stripe unit sized I/Os being sent to each disk (512KiB) while sustaining roughly 13,000 disk I/Os per second to achieve 6.3GiB/s.

In contrast, Ext2 and Ext3 were issuing approximately 60-70,000 disk I/Os per second

Threads	Best Run	Worst Run
1	522.2	348.5
2	780.2	74.8
4	1130.3	105.0
8	1542.1	176.8

Table 2: Example of Ext2 Read Throughput Variability (MiB/s)

to achieve 1.7GiB/s and 4.5GiB/s respectively. That equates to average I/O sizes of approximately 24KiB and 56KiB and each disk executing more than 250 I/Os per second each. The disks were seek bound rather than bandwidth bound. Sustained read throughput of less than 300MiB/s at 60-70,000 disk I/Os per second with an average size of 4KiB was not uncommon to see. This indicates worst case (single block) fragmentation in the filesystem. The same behaviour was seen with JFS as well.

The source of the fragmentation on Ext2 and Ext3 would appear to be interleaved disk allocation when multiple files are written in parallel from multiple CPUs. This also occurred when running parallel direct I/O writes on Ext3 (see Table 1) so it would appear to be a general issue with the way Ext3 handles parallel allocation streams.

XFS solves this problem by decoupling disk block allocation from disk space accounting and then using well known algorithmic techniques to avoid lock contention to achieve write scaling.

The message being conveyed here is that most Linux filesystems do not resist fragmentation under parallel write loads. With parallelism hitting the mainstream now via multicore CPUs, we need to recognise that filesystems may not be as resistant to fragmentation under normal usage patterns as they were once recognised to be. This used to be a problem that only supercomputer vendors had to worry about...

PID	State	% CPU	Name
23589	R	97	dd
345	R	88	kswapd7
344	R	83	kswapd6
23556	R	81	dd
348	R	80	kswapd10
346	R	79	kswapd8
347	R	77	kswapd9
339	R	76	kswapd1
349	R	74	kswapd11
343	R	72	kswapd5
23517	R	71	dd
23573	R	64	dd
338	R	64	kswapd0
23552	R	64	dd
23502	R	63	dd
340	S	63	kswapd2
23570	R	61	dd
23592	R	60	dd
341	R	57	kswapd3

Table 3: kswapd CPU usage during buffered writes.

5.3 kswapd and pdflush

While running single threaded tests, it was clear that there was something running in the background that was using more CPU time than the writer process and pdflush combined. A single threaded read from disk consuming a single CPU was consuming 10-15% of a CPU on each node running memory reclaim via kswapd. For a single threaded write, this was closer to 30% of a CPU per node. On our twelve node machine, this meant that we were using between 1.5 and 3.5 CPUs to reclaim memory being allocated by a single CPU.

On buffered write tests, pdflush also appeared to be struggling to write out the dirty data. With a single write thread, pdflush would consume very little CPU; maybe 10% of a single CPU every five seconds. As the number of threads increased, however, pdflush quickly became overwhelmed. At four threads writing

Threads	Average I/O Size
1	1000KiB
2	450KiB
4	400KiB
8	250KiB
16	200KiB
32	220KiB

Table 4: I/O size during buffered writes.

at approximately 1.5GiB/s, pdflush ran permanently consuming an entire CPU.

At eight or more write threads, pdflush consumed CPU time only sporadically; instead the kswapd CPU usage jumped from 30% of a CPU to 70-80% of a CPU per node. This can be seen in Table 3.

Monitoring of the disk level I/O patterns indicated that writeback was occuring from the LRU lists rather than in file offset order from pdflush. This could also be seen in the I/O sizes that were being issued to disk as seen in Table 4 as the thread count increased.

This is clearly not scalable writeback and memory reclaim behaviour; we need reclaim to consume less CPU time and for all writeback to occur in file offset order to maximise throughput. For XFS, this will also minimise fragmentation during block allocation. See Section 6.2.2 for details on how we improved this behaviour.

6 Improvements and Optimisations

6.1 XFS Modifications

6.1.1 Buffered Write I/O Path

In 2.6.15, a new buffered write I/O path implementation was introduced. This was written by Christoph Hellwig and Nathan Scott[Hellwig]. The main change this introduced was XFS clustering pages directly into a `bio` instead of by buffer heads and `submit_bh` calls. Using buffer heads limited the size of an I/O to the number of buffer heads a `bio` could hold. In other words, the larger the block size of the filesystem, the larger the I/Os that could be formed in the write cluster path. This is the primary reason for the difference in throughput we see for the XFS filesystems with different block sizes.

By adding complete pages to a `bio` rather than buffer heads, we were able to make XFS write clustering independent of the filesystem block size. This means that any XFS filesystem can issue I/Os only limited in size by the number of pages that can be held by the `bio` vector.

Unfortunately, due to the locking issue described earlier in Section 5.1, XFS with the modified write path was actually slower on our test machine than without it. Clearly, the spinlock problem needed to be solved before we would see any benefit from the new I/O path.

6.1.2 Per-CPU Superblock Counters

Kernel profiles taken during parallel buffered write tests indicated contention within XFS on the in-core superblock lock. This lock protects the current in-core (in-memory) state of the filesystem.

In the case of delayed allocation, XFS uses the in-core superblock to track both disk space that is actually allocated on disk as well as the space that has not yet been allocated but is dirty in memory. That means during a `write(2)` system call we allocate the space needed for the data being written but we don't allocate disk blocks. Hence the "allocation" is very fast whilst maintaining an accurate representation

of how much space there is remaining in the filesystem.

This makes contention on this structure a difficult problem to solve. We need global accuracy, but we now need to avoid global contention. The in-core superblock is a write-mostly structure, so we can't use atomic operations or RCU to scale it. The only commonly used method remaining is to make the counters per-CPU, but we still need to have some method of being accurate when necessary that performs in an acceptable manner.

Hence for the free space counter we decided to trade off performance for accuracy when we are close to ENOSPC. The algorithm that was implemented is essentially a distributed counter that gets slower and more accurate as the aggregated total of the counter approaches zero.

When an individual per-CPU counter reaches zero, we execute a balance operation. This operation locks out all the per-CPU counters before aggregating and redistributing the aggregated value evenly over all the counters before re-enabling the counters again. This requires a per-CPU atomic exclusion mechanism. The balance operation must lock every CPU fast path out and so can be an expensive operation on a large machine.

However, on that same large machine, the fast path cost of the per-CPU counters is orders of magnitude lower than a global spinlock. Hence we are amortising the cost of an expensive rebalance very quickly compared to using a global spinlock on every operation. Also, when the filesystem has lots of free space we rarely see a rebalance operation as the distributed counters can sink hundreds of gigabytes of allocation on a single CPU before running dry.

If a counter rebalance results in a very small amount being distributed to each CPU, the counter is considered to be near zero and we fall back to a slow, global, single threaded counter for the aggregated total. That is, we prefer accuracy over blazing speed. It should also be noted that using a global lock in this case tends to be more efficient than constant rebalancing on large machines.

The results (see Figure 7 and Figure 8) speak for themselves and the code is to be released with 2.6.17[Chinner].

6.2 VM and NUMA Issues

6.2.1 SN2 Specific TLB Purging

When first running tests on 2.6.15-rc5, it was noticed that XFS buffered read speeds were much higher than we saw on SLES9 SP2, SLES9 SP3 and 2.6.14. On these kernels we were only achieving a maximum of 4GiB/s. Using 2.6.15-rc5 we achieved 6.4GiB/s, and monitoring showed all the disks at greater than 90% utilisation so we were now getting near to being disk bound.

Further study revealed that the memory reclaim rate limited XFS buffered read throughput. In this particular case, the global TLB flushing speed was found to make a large difference to the reclaim speed.

We found this when we reverted a platform specific optimisation that was included in 2.6.15-rc1 to speed up TLB flushing[Roe]. Reverting this optimisation reduced buffered read throughput by approximately 30% on the same filesystem and files. Simply put, this improvement was an unexpected but welcome side effect of an optimisation made for different reasons.

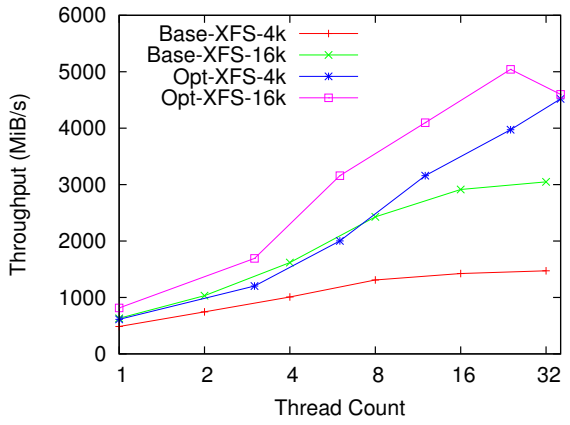

Figure 7: Improved XFS Buffered Write Throughput.

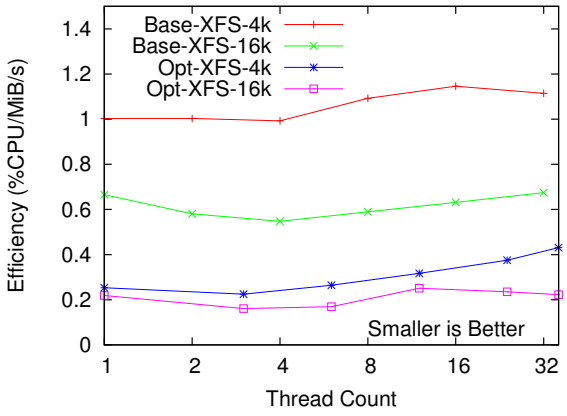

Figure 8: Improved XFS Buffered Write Efficiency.

6.2.2 Node Local Memory Reclaim

In a stroke of good fortune, Christoph Lameter completed a set of modifications to the memory reclaim subsystem[Lameter] while we were running tests. The modifications were included in Linux 2.6.16, and they modified the reclaim behaviour to reclaim clean pages on a given node before trying to allocate from a remote node.

The first major difference in behaviour was that kswapd never ran during either buffered read or write tests. Buffered reads were now quite obviously I/O bound with approximately half the disks showing 100% utilisation. Using a different volume layout with a 1MiB stripe unit, sustained buffered read throughput increased to over 7.6GiB/s.

The second most obvious thing was that pdflush was now able to flush more than 5GiB/s of data whilst consuming less than half a CPU. Without the node local reclaim, it was only able to push approximately 500MiB/s when it consumed an equivalent amount of CPU time. Writeback, especially at low thread counts, became far more efficient.

6.2.3 Memory Interleaving

While doing initial bandwidth characterisations using direct I/O, we found that it was necessary to ensure that buffer memory was allocated evenly from every node in the machine. This was achieved using the `numactl -i all` command prefix to the test commands being run.

Without memory interleaving, direct I/O (read or write) struggled to achieve much more than 6GiB/s due to the allocation patterns limiting the buffers to only a few nodes in the machine. Hence we were limited by the per-node NUMALink bandwidth. Interleaving the buffer memory across all the nodes solved this problem.

With buffered I/O, however, we saw very different behaviours. In initial testing we saw little difference in throughput because the page cache ended up spread across all nodes of the machine due to memory reclaim behaviour.

However, when testing the node local memory reclaim patches we found that interleaving did make a big difference to performance as the local reclaim reduced the number of nodes that

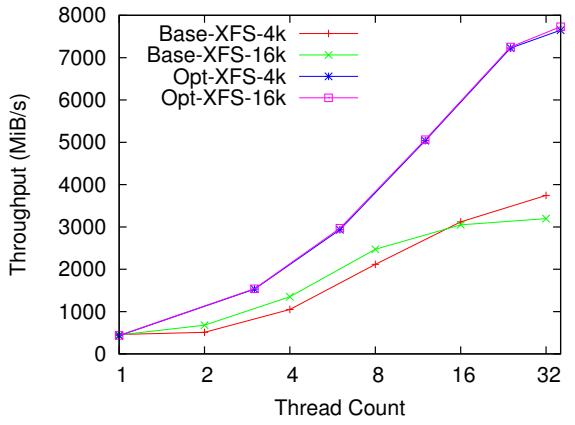

Figure 9: Improved XFS Buffered Read Throughput.

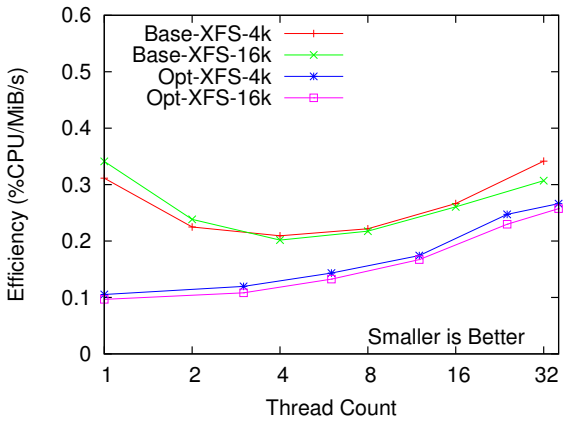

Figure 10: Improved XFS Buffered Read Efficiency.

the page cache ended up spread over. Interestingly, the improvement in memory reclaim speed that the local reclaim gave us meant that there was no performance degradation despite not spreading the pages all over the machine. Once we spread the pages using the `numactl` command we saw the substantial performance increases.

6.2.4 Results

We've compared the baseline buffered I/O results from Section 3 with the best results we achieved with our optimised kernel.

From Figure 7 it is clear that we achieved a substantial gain in write throughput. The outstanding result is the improvement of 4KiB block size filesystems and is a direct result of the I/O path rewrite. The improved write clustering resulted in consistently larger I/Os being sent to disk, and this has translated into improved throughput. Local memory reclaim has also prevented I/O sizes from decreasing as the number of threads writing increases which has also contributed to higher throughputs as well.

On top of improved throughput, Figure 8 indicates that the buffered write efficiency has improved by factor of between three and four. It can been seen that the efficiency decreases somewhat as throughput and thread count goes up, so there is still room for improvement here.

Buffered read throughput has roughly doubled as shown in Figure 9. This improvement can be almost entirely attributed to the VM improvements as the XFS read path is almost identical in the baseline and optimised kernels.

Once again, the improvement in throughput corresponds directly to an improvement in efficiency. Figure 10 indicates that we saw much greater improvements in efficiency at low thread counts than at high thread counts. The source of this decrease in efficiency is unknown and more investigation is required to understand it.

One potential reason for the decrease in efficiency of the buffered read test as throughput increases is that the NUMALink interfaces may be getting close to saturation. With the tests being run, the typical memory access patterns are a DMA write from the HBA to memory, which due to the interleaved nature of the page cache is distributed across the NUMALink fab-

ric. The data is then read by a CPU, which gathers the data spread across every node, and is then written back out into a user buffer which is spread across every node.

With both bulk data and control logic memory references included, each node node in the system is receiving at least 2GiB/s and transmitting more than 1.2GiB/s. With per-node receive throughput this high, remote memory read and write latencies can increase compared to an idle interconnect. Hence the increase in CPU usage may simply be an artifact of sustained high NUMALink utilisation.

7 Futures

The investigation that we undertook has provided us with enough information about the behaviour of these large systems for us to predict issues that SGI customers will see over the next year or two. It has also demonstrated that there are issues that mainstream Linux users are likely to start to see over this same timeframe. With technologies like SAS, PCI express, multicore CPUs and NUMA moving into the mainstream, issues that used to affect only high end machines are rapidly moving down to the average user. We need to make sure that our filesystems behave well on the average machine of the day.

At the high end, while we are on top of filesystem scaling issues with XFS, we are starting to see interactions between high bandwidth I/O and independent cpuset constrained jobs on large machines. These interactions are complex and are hinting that for effective deployment on large machines at high I/O bandwidths the filesystem needs to be NUMA and I/O path topology aware so that filesystem placement and I/O bandwidth locality to the running job can be maximised. That is, we need to be able to control placement in the filesystem to minimise the NUMALink bandwidth that a job's I/O uses.

This means that filesystems are likely to need allocation hints provided to them to enable this sort of functionality. We already have policy information controlling how a job uses CPU and memory in large machines, so extending this concept to how the filesystem does allocation is not as far-fetched as it seems.

Improving performance in filesystems is all about minimising disk seeking, and this comes down to the way the filesystem allocates its disk space. We have new issues at the high end to deal with, while the issues that have been solved at the high end are now becoming issues for mainstream. As the intrinsic parallelism of the average computer increases, algorithms need to be able to resist fragmentation when allocations occur simultaneously so that filesystem performance can grow with machine capability.

8 Conclusion

The investigation we undertook has provided us with valuable information on the behaviour of Linux in high bandwidth I/O loads. We identified several areas which limited our performance and scalability and fixed the worst during the investigation.

We improved the efficiency of buffered I/O under these loads and significantly increased the throughput we could achieve from XFS. We discovered interesting NUMA scalability issues and either fixed them or developed effective strategies to negate the issues.

We proved that we could achieve close to the physical throughput limits of the disk subsystem with direct I/O. From analysis, we found

that even buffered I/O was approaching physical NUMALink bandwidth limits. We proved that Linux and XFS in combination could do this whilst maintaining reproducible and stable operation.

We also uncovered a set of generic filesystem issues that affected every filesystem we tested. We solved these problems on XFS, and provided recommendations on why we think they also need to be solved.

Finally, we proved that XFS is the best choice for our customers; both on the machines they use and for the common workloads they run.

In conclusion, our investigation fulfilled all the goals we set at the beginning of the task. We gained insight into future issues we are likely to see, and we raised a new set of questions that need further research. Now all we need is a bigger machine and more disks.

References

[PCP] Silicon Graphics Inc,
Performance Co-Pilot,
`http://oss.sgi.com/projects/pcp/`

[Scott] Nathan Scott
Make alloc_page_buffers() initialise buffer_heads using init_buffer()
Git commit key:
`01ffe339e3a0ba5ecbeb2b3b5abac7b3ef90f374`

[Mason] Chris Mason,
[PATCH] reiserfs: zero b_private when allocating buffer heads
Git commit key:
`fc5cd582e9c934ddaf6f310179488932cd154794`

[Roe] Dean Roe,
[IA64] - Avoid slow TLB purges on SGI Altix systems
Git commit key:
`c1902aae322952f8726469a6657df7b9d5c794fe`

[Lameter] Christoph Lameter,
[PATCH] Zone reclaim: Reclaim logic
Git commit key:
`9eeff2395e3cfd05c9b2e6074ff943a34b0c5c21`

[Hellwig] Christoph Hellwig and Nathan Scott,
[XFS] Initial pass at going directly-to-bio on the buffered IO path
Git commit key:
`f6d6d4fcd180f8e47bf6b13fc6cce1e6c156d0ea`

[Chinner] Dave Chinner
[XFS] On machines with more than 8 cpus, when running parallel I/O
Git commit key:
`8d280b98cfe3c0b69c37d355218975c1c0279bb0`