Linux 2.6 performance improvement through readahead optimization

Ram Pai
IBM Corporation
`linuxram@us.ibm.com`

Badari Pulavarty
IBM Corporation
`badari@us.ibm.com`

Mingming Cao
IBM Corporation
`mcao@us.ibm.com`

Abstract

Readahead design is one of the crucial aspects of filesystem performance. In this paper, we analyze and identify the bottlenecks in the redesigned Linux 2.6 readahead code. Through various benchmarks we identify that 2.6 readahead design handles database workloads inefficiently. We discuss various improvements made to the 2.6 readahead design and their performance implications. These modifications resulted in impressive performance improvements ranging from 25%–100% with various benchmarks. We also take a closer look at our modified 2.6 readahead algorithm and discuss current issues and future improvements.

1 Introduction

Consider an application that reads data sequentially in some fixed-size chunks. The kernel reads data sufficiently enough to satisfy the request from the backing storage and hands it over to the application. In the meantime the application ends up waiting for the data to arrive from the backing store. The next request also takes the same amount of time. This is quite inefficient. What if the kernel anticipated the future requests and cached more data? If it could do so, the next read request could be satisfied much faster, decreasing the overall read latency.

Like all other operating systems, Linux uses this technique called *readahead* to improve read throughput. Although readahead is a great mechanism for improving sequential reads, it can hurt the system performance if used blindly for random reads.

We studied the performance of the readahead algorithm implemented in 2.6.0 and noticed the following behavior for large random read requests.

1. reads smaller chunks of data many times, instead of reading the required size chunk of data once.
2. reads more data than required and hence wasted resources.

In Section 2, we discuss the readahead algorithm implemented in 2.6 and identify and fix the inefficient behavior. We explain the performance benefits achieved through these fixes in Section 3. Finally, we list the limitations of our fixes in Section 4.

2 Readahead Algorithm in 2.6

2.1 Goal

Our initial investigation showed the performance on Linux 2.6 of the Decision Support System (DSS) benchmark on filesystem was about 58% of the same benchmark run on raw devices. Note that the DSS workload is characterized by large-size random reads. In general, other micro-benchmarks like rawio-bench and aio-stress showed degraded performance with random workloads. The suboptimal readahead behavior contributed significantly toward degraded performance. With these inputs, we set the following goals.

1. Exceed the performance of 2.4 large random workloads.
2. DSS workload on filesystem performs at least 75% as well as the same on raw devices.
3. Maintain or exceed sequential read performance.

2.2 Introduction to the 2.6 readahead algorithm

Figure 1 presents the behavior of 2.6.0 readahead. The `current_window` holds pages that satisfy the current requests. The `readahead_window` holds pages that satisfy the anticipated future request. As more page requests are satisfied by the `current_window` the estimated size of the next `readahead_window` expands. And if page requests miss the `current_window` the estimated size of the `readahead_window` shrinks. As soon as the read request cross `current_window` boundary and steps into the first page of the `readahead_window`, the `readahead_window` becomes the `current_window` and the `readahead_window` is reset. However, if the requested page misses any page in the `current_window` and also the first page in the `readahead_window`, both the `current_window` and the `readahead_window` are reset and a new set of pages are read into the `current_window`. The number of pages read in the current window depends upon the estimated size of the `readahead_window`. If the estimated size of the `readahead_window` drop down to zero, the algorithm stops reading ahead, and enters the slow-read mode till page request pattern become sufficiently contiguous. Once the request pattern become sufficiently contiguous the algorithm re-enters into readahead-mode.

2.3 Optimization For Random Workload

We developed a user-level simulator program that mimicked the behavior of the above readahead algorithm. Using this program we studied the read patterns generated by the algorithm in response to the application's read request pattern.

In the next few subsections we identify the bottlenecks, provide fixes and then explain the results of the fix. As a running example we use a read sequence consisting of 100 random read-requests each of size 16 pages.

2.3.1 First Miss

Using the above read pattern, we noticed that the readahead algorithm generated 1600 requests of size one page. The algorithm penalized the application by shutting down readahead immediately, for not reading from the beginning of the file. It is sub-optimal to assume that application's read pattern is random, just because it did not read the file from the beginning. The offending code is at line 16 in Figure 1. Once shut down, the slow-read mode made readahead to not resume since the `current_window` never becomes large enough. For the ext2/ext3 filesystem, the `current_window` must become 32 pages large, for readahead to resume. Since the application's requests were all 16 pages large, the `current_window` never opened. We re-

```
for each page in the current request
do
      if readahead is shutdown
      then  // read one page at a time (SLOW-READ MODE)
        if requested page is next to the previously requested page
        then
             open the current_window by one more page
        else
             close the current_window entirely
        fi

        if the current_window opens up by maximum readahead_size
        then
              activate readahead  // enter READAHEAD-MODE
        fi
        read in the requested page

      else  // read many pages at a time (READAHEAD MODE)
        if this is the first read request and is for the first page
                          of this open file instance
               set the estimated readahead_size to half the size of
                       maximum readahead_size
        fi

        if the requested page is within the current_window
               increase the estimated readahead_size by 2
               ensure that this size does not exceed maximum
                        readahead_size
        else
               decrease the estimated readahead_size by 2
               if this estimate becomes zero, shutdown readahead
        fi

        if the requested page is the first page in the readahead_window
        then
                move the pages in the readahead_window to the
                      current_window and reset the readahead_window
                continue
        fi

        if the requested page is not in the current_window
        then
               delete all the page in current_window and readahead_window
               read the estimated number of readahead pages starting
                    from the requested page and place them into the current
                    window.
               if all these pages already reside in the page cache
               then
                    shrink the estimated readahead_size by 1 and
                    shutdown readahead if the estimate touches zero
               fi
            else if the readahead_window is reset
                 then
                       read the estimated number of readahead pages
                       starting from the page adjacent to the last page
                       in the current window and place them in the
                       readahead_window.
                       if all these pages already reside in the page cache
                       then
                           shrink the estimated readahead_size by 1 and
                           shutdown readahead if the estimate touches zero
                       fi
                 fi
        fi
      fi
done
```

Figure 1: *Readahead algorithm in 2.6.0*

moved the check at line 16 to not expect read access to start from the beginning.

For the same read pattern the simulator showed 99 32-page requests, 99 30-page requests, one 16-page request, and one 18-page request to the block layer. This was a significant improvement over 1600 1-page requests seen without these changes.

However, the DSS workload did not show any significant improvement.

2.3.2 First Hit

The reason why DSS workload did not show significant improvement was that readahead shut down because the accessed pages already resided in the page-cache. This behavior is partly correct by design, because there is no advantage in reading ahead if all the required pages are available in the cache. The corresponding code is at line 43. But shutting down readahead by just confirming that the initial few pages are in the page-cache and assuming that future pages will also be in the page cache, leads to worse performance. We fixed the behavior, to not close the `readahead_window` the first time, even if all the requested pages were in the page-cache. The combination of the above two changes ensured continuous large-size read activity.

The simulator showed the same results as the First-Miss fix.

However, the DSS workload showed 6% improvement.

2.3.3 Extremely Slow Slow-read Mode

We also observed that the slow-read mode of the algorithm expected 32 contiguous page access to resume large size reads. This is not a realistic expectation for random workload. Hence, we changed the behavior at line 9 to shrink the `current_window` by one page if it lost contiguity.

The simulator and DSS workload did not show any better results because the combination of First-Hit and First-Miss fixes ensured that the algorithm did not switch to the slow-read mode. However a request pattern comprising of 10 single page random requests followed by a continuous stream of 4-page random requests can certainly see the benefits of this optimization.

2.3.4 Upfront Readahead

Note that readahead is triggered as soon as some page is accessed in the `current_window`. For random workloads, this is not ideal because none of the pages in the `readahead_window` are accessed. We changed line 45, to ensure that the readahead is triggered only when the last page in the `current_window` is accessed. Essentially, the algorithm waits until the last page in the `current_window` is accessed. This increases the probability that the pages in the `readahead_window` if brought in, will get used.

With these changes, the simulator generated 99 30-page requests, one 32-page request, and one 16-page request.

There was a significant 16% increase in performance with the DSS workload.

2.3.5 Large `current_window`

Ideally, the readahead algorithm must generate around 100 16-page requests. Observe however that almost all the page requests are of size 30 pages. When the algorithm observes that a page request has missed the `current_window`, it scraps both the `current_window` and the `readahead_window`, if one exists. It ends up reading in a new `current_window`, whose size is based on the estimated `readahead_size`. Since all of the pages in a given application's read request are contiguous, the estimated readahead size tends to reach the maximum `readahead_size`. Hence, the size of the new `current_window` is too large; most of the pages in the window tend to be wasted. We ensured that the new `current_window` is as large as the number of pages that were used in the present `current_window`.

With this change, the simulator generated 100 16-page requests, and 100 32-page requests. These results are awful because the last page of the application's request almost always coincides with the last page of the `current_window`. Hence, the readahead is triggered when the last page of the `current_window` is accessed, only to be scrapped.

We further modified the design to read the new `current_window` with one more page than the number of pages accessed in the present `current_window`.

With this change, the simulator for the same read pattern generated 99 17-page requests, one 32-page request, and one 16-page request to the block layer, which is close to ideal!

The DSS workload showed another 4% better performance.

The collective changes were:

1. Miss fix: Do not close readahead if the first access to the file-instance does not start from offset zero.

2. Hit fix: Do not close readahead if the first access to the requested pages are already found in the page cache.

3. Slow-read Fix: In the slow-read path, reduce one page from the `current_window` if the request is not contiguous.

4. Lazy-read: Defer reading the `readahead_window` until the last page in the `current_window` is accessed.

5. Large `current_window` fix: Read one page more than the number of pages accessed in the current window if the request misses the current window.

These collective changes resulted in an impressive 26% performance boost on DSS workload.

2.4 Sequential Workload

The previously described modifications were not without side effects! The sequential workload was badly effected. Trond Myklebust reported 10 times worse performance on sequential reads using the iozone benchmark on an NFS based filesystem. The lazy read optimization broke the pipeline effect designed for sequential workload. For sequential workload, readahead must be triggered as soon as some page in the current window is accessed. The application can crunch through pages in the `current_window` as the new pages get loaded in the `readahead_window`.

The key observation is that upfront readahead helps sequential workload and lazy readahead helps random workload. We developed logic that tracked the average size of the read requests. If the average size is larger than the maximum readahead size, we treat that workload as sequential and adapt the algorithm to do upfront readahead. However, if the average size is less than the maximum `readahead_`

```
1 for each page in the current request ; do
3       if readahead is shutdown
4       then // read one page at a time (SLOW-READ MODE)
5            if requested page is next to the previously requested page
6            then
7                 open the current_window by one more page
8            else
9                 shrink current_window by one page
10           fi
11           if the current_window opens up by maximum readahead_size
12           then
13                  activate readahead  // enter READAHEAD-MODE
14           fi
15           read in the requested page
        else // read many pages at a time (READAHEAD MODE)
16-17        if this is the first read request for this open file-instance ; then
18              set the estimated readahead_size to half the size of maximum readahead_size
19           fi
20           if the requested page is within the current_window
21              increase the estimated readahead_size by 2
22              ensure that this size does not exceed maximum readahead_size
23           else
24               decrease the estimated readahead_size by 2
25               if this estimate becomes zero, shutdown readahead
26           fi
27           if requested page is contiguous to the previously requested page
28           then
29              Increase the size of the present read request by one more page.
30           else
31               Update the average size of the reads with the size of the previous request.
32           fi
33           if the requested page is the first page in the readahead_window
34           then
35                move the pages in current_window to the  readahead_window
36                reset readahead_window
37                continue
38           fi
39-40        if the requested page is not in the current_window ; then
41              delete all pages in current_window and readahead_window
42              if this is not the first access to this file-instance
43              then
44                 set the estimated number of readahead pages to the
                   average size of the read requests.
45              fi
46              read the estimated number of readahead pages starting from
                    the requested page and place them into the current window.
47              if this not the first access to this file instance and
                      all these pages already reside in the page cache
48              then
49                    shrink the estimated readahead_size by 1 and
                             shutdown readahead if the estimate touches zero
50              fi
51           else if the readahead_window is reset and if the average
                      size of the reads is above the maximum readahead_size
52                then
53                     read the readahead_window with the estimated
54                       number of readahead pages starting from the
55                       page adjacent to the last page in the current window.
56                       if all these pages already reside in the page cache
57                       then
58                           shrink the estimated readahead_size by 1 and
                             shutdown readahead if the estimate touches zero
59                       fi
60                fi
61           fi
62-63    fi ; done
```

Figure 2: *Optimized Readahead algorithm*

`size`, we treat that workload as random and adapt the algorithm to do lazy readahead.

This adaptive-readahead fixed the regression seen with sequential workload while sustaining the performance gains of random workload.

Also we ran a sequential read pattern through the simulator and found that it generated large size upfront readahead. For large random workload it hardly read ahead.

2.4.1 Simplification

Andrew Morton rightly noted that reading an extra page in the `current_window` to avoid lazy-readahead was not elegant. Why have lazy-readahead and also try to avoid lazy-readahead by reading one extra page? The logic is convoluted. We simplified the logic through the following modifications.

1. Read ahead only when the average size of the read request exceeds the maximum `readahead_size`. This helped the sequential workload.
2. When the requested page is not in the `current_window`, replace the `current_window`, with a new `current_window` the size of which is equal to the average size of the application's read request.

This simplification produced another percent gain in DSS performance, by trimming down the `current_window` size by a page. More significantly the sequential performance returned back to initial levels. We ran the above modified algorithm on the simulator with various kinds of workload and got close to ideal request patterns submitted to the block layer.

To summarize, the new readahead algorithm has the following modifications.

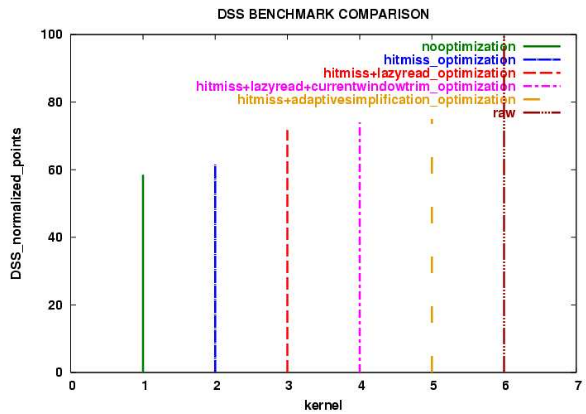


Figure 3: *Progressive improvement in DSS benchmark, normalized with respect to the performance of DSS on raw devices.*

1. Miss fix: Do not close readahead if the first access to the file-instance does not start from offset zero.
2. Hit fix: Do not close readahead if the first access to the requested pages are already found in the page cache.
3. Slow-read Fix: Decrement one page from the `current_window` if the request is not contiguous in the slow-read path.
4. Adaptive readahead: Keep a running count of the average size of the application's read requests. If the average size is above the maximum `readahead_size`, readahead up front. If the request misses the `current_window`, replace it with a new `current_window` whose size is the average size of the application's read requests.

Figure 2 shows the new algorithm with all the optimization incorporated.

Figure 3 illustrates the normalized steady increase in the DSS workload performance with each incremental optimization. The graph is normalized with respect to the performance of

DSS on raw devices. Column 1 is the base performance on filesystem. Column 2 is the performance on filesystem with the hit, miss and slow-read optimization. Column 3 is the performance on filesystem with first-hit, first-miss, slow-read and lazy-read optimization. Column 4 is the performance on filesystem with first-hit, first-miss, slow-read, and large `current_window` optimization. Column 5 is the performance on filesystem with first-hit, first-miss, slow-read, and adaptive read simplification. Column 6 is the performance on raw device.

3 Overall Performance Results

In this section we summarize the results collected through simulator, DSS workload, and iozone benchmark.

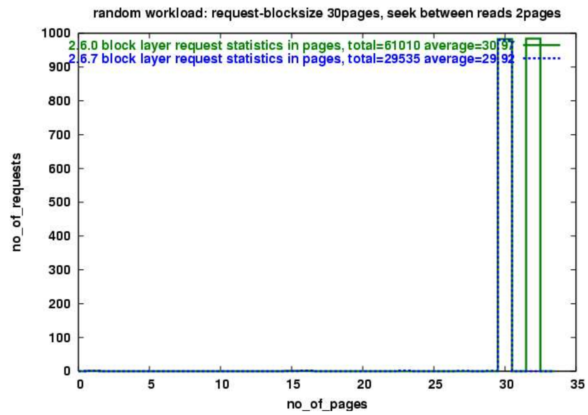
3.1 Results Seen Through Simulator

We generated different types of input read patterns. There is no particular reason behind these particular read pattern. However, we ensured that we get enough coverage. Overall the read requests generated by our optimized readahead algorithm outperformed the original algorithm. The graphs refer to our optimized algorithm as 2.6.7 because all these optimizations are merged in the 2.6.7 release candidate.

Figure 4 shows the output of readahead algorithm with and without optimization for 30-page read request followed by 2-page seek, repeated 984 times.

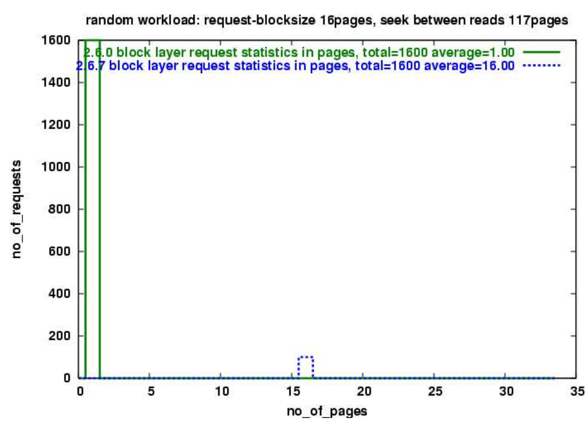
Figure 5 shows the output of readahead algorithm with and without optimization for 16-page read request followed by 117-page seek, repeated 100 times.

Figure 6 shows the output of readahead algorithm with and without optimization for 32-



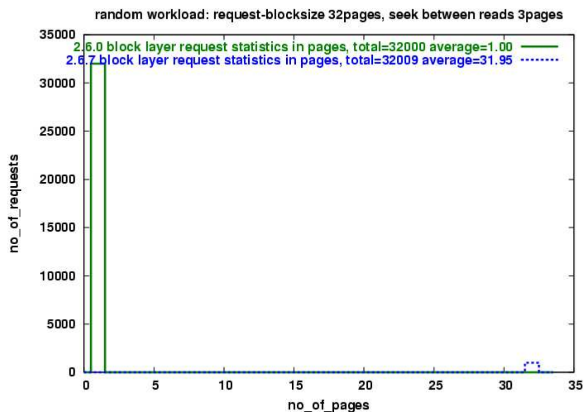
	2.6.0	2.6.7
Average Size	31	30
Pages Read	61010	29535
Wasted Pages	31490	15
No Of Read Requests	1970	987

Figure 4: *Application generates 30-page read request followed by 2-page seek, repeating 984 times. Totally 29520 pages requested.*



	2.6.0	2.6.7
Average Size	1	16
Pages Read	1600	1600
Wasted Pages	0	0
No Of Read Requests	1600	100

Figure 5: *Application generates 16-page read request followed by 117-page seek, repeating 100 times. Totally 1600 pages requested.*



	2.6.0	2.6.7
Average Size	1	31.95
Pages Read	32000	32009
Wasted Pages	0	9
No Of Read Requests	32000	1002

Figure 6: *Application generates 32-page read request followed by 3-page seek, repeating 1000 times. Totally 32000 pages requested.*

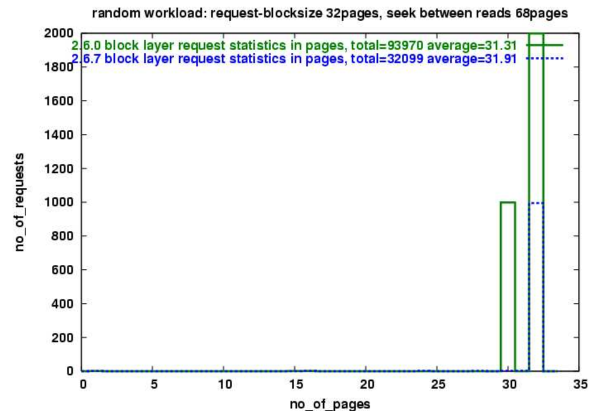
page read request followed by 3-page seek, repeated 1000 times.

Figure 7 shows the output of readahead algorithm with and without optimization for 32-page read request followed by 68-page seek, repeated 1000 times.

Figure 8 shows the output of readahead algorithm with and without optimization for 40-page read request followed by 5-page seek, repeated 1000 times.

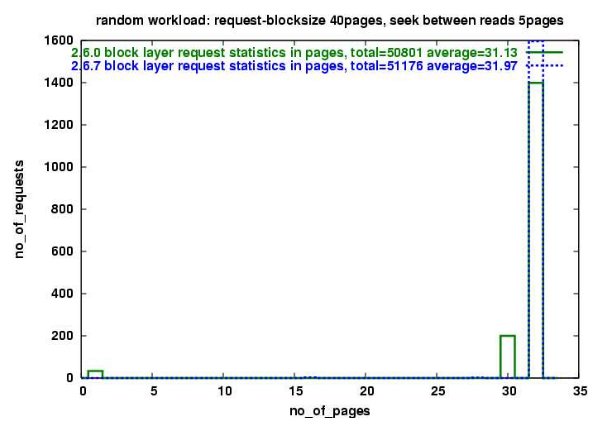
Figure 9 shows the output of readahead algorithm with and without optimization for 4-page read request followed by 96-page seek, repeated 1000 times.

Figure 10 shows the output of readahead algorithm with and without optimization for 16-page read request followed by 0-page seek, repeated 1000 times.



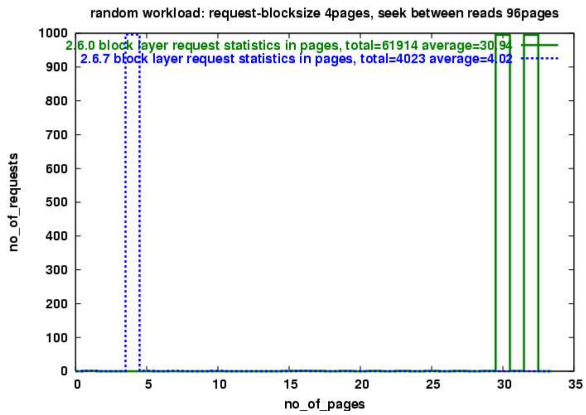
	2.6.0	2.6.7
Average Size	31.31	31.91
Pages Read	93970	32099
Wasted Pages	61970	99
No Of Read Requests	3001	1006

Figure 7: *Application generates 32-page read request followed by 68-page seek, repeating 1000 times. Totally 32000 pages requested.*



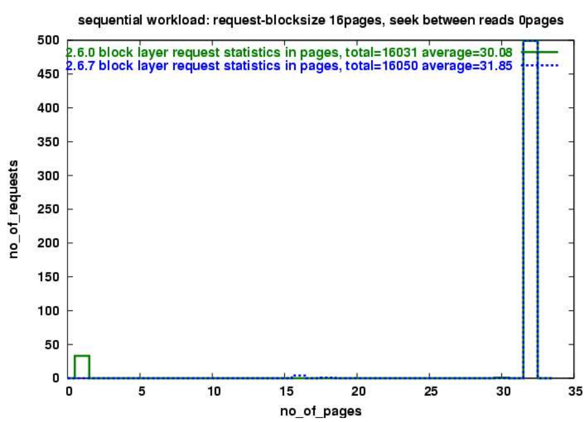
	2.6.0	2.6.7
Average Size	31.13	31.91
Pages Read	50810	51176
Wasted Pages	10801	11176
No Of Read Requests	1631	1601

Figure 8: *Application generates 40-page read request followed by 5-page seek, repeating 1000 times. Totally 40000 pages requested.*



	2.6.0	2.6.7
Average Size	30.94	4.02
Pages Read	61914	4023
Wasted Pages	57914	23
No Of Read Requests	2001	1001

Figure 9: *Application generates 4-page read request followed by 96-page seek, repeating 1000 times. Totally 4000 pages requested.*



	2.6.0	2.6.7
Average Size	30.08	31.85
Pages Read	16031	16050
Wasted Pages	31	50
No Of Read Requests	533	504

Figure 10: *Application generates 16-page read request with no seek, repeating 1000 times. Totally 16000 pages requested.*

3.2 DSS Workload

The configuration of our setup is as follows:

- 8-way Pentium III machine.
- 4GB RAM
- 5 fiber-channel controllers connected to 50 disks.
- 250 partitions in total each containing a ext2 filesystem.
- 30GB Database is striped across all these filesystems. No filesystem contains more than one table.
- Workload is mostly read intensive, generating mostly large 256KB random reads.

With this setup we saw an impressive 26% increase in performance. The DSS workload on filesystems is roughly about 75% to DSS workload on raw disks. There is more work to do, although the bottlenecks may not necessarily be in the readahead algorithm.

3.3 Iozone Results

The iozone benchmark was run a NFS based filesystem. The command used was `iozone -c -t1 -s 4096m -r 128k`. This command creates one thread that reads a file of size 4194304 KB, generating reads of size 128 KB. The results in Table 1 show an impressive 100% improvement on random read workloads. However we do see 0.5% degradation with sequential read workload.

4 Future Work

There are a couple of concerns with the above optimizations. Firstly, we see a small 0.5%

Read Pattern	2.4.20	2.6.0	2.6.0 + optimization
Sequential Read	10846.87	14464.20	13614.49
Sequential Re-read	10865.39	14591.19	13715.94
Reverse Read	10340.34	10125.13	20138.83
Stride Read	10193.87	7210.96	14461.63
Random Read	10839.57	10056.49	19968.79
Random Mix Read	10779.17	10053.37	21565.43
Pread	10863.56	11703.76	13668.21

Table 1: *Iozone benchmark Throughput in KB/sec for different workloads.*

degradation with the sequential workload using the iozone benchmark. The optimized code assumes the given workload to be random to begin with, and then adapts to the workload depending on the read patterns. This behavior can slightly affect the sequential workload, since it takes a few initial sequential reads before the algorithm adapts and does upfront readahead.

The optimizations introduce a subtle change in behavior. The modified algorithm does not correctly handle inherently-sequential clustered read patterns. It wrongly thinks that such read patterns seek after every page-read. The original 2.6 algorithm did accommodate such patterns to some extent. Assume an application with 16 threads reading 16 contiguous pages in parallel, one per thread. Based on how the threads are scheduled, the read patterns could be some combination of those 16 pages. An example pattern could be 1,15,8,12,9,6,2,14,10,7,5,3,4,11,12,13. The original 2.6.0 readahead algorithm did not care which order the page requests came in as long as the pages were in the current-window. With the adaptive readahead, we expect the pages to be read exactly in sequential order.

Issues have been raised regularly that the readahead algorithm should consider the size of the current read request to make intelligent decisions. Currently, the readahead logic bases its readahead decision on the read patterns seen in the past, including the request for the current page without considering the size of the current request. This idea has merit and needs investigation. We probably can ensure that we at least read the requested number of pages if readahead has been shutdown because of page-misses.

5 Conclusion

This work has significantly improved random workloads, but we have not yet reached our goal. We believe we have squeezed as much as possible performance from the readahead algorithm, though there is some work to be done to improve some special case workloads, as mentioned in Section 4. There may be other subsystems that need to be profiled to identify bottlenecks. There is a lot more to do!

6 Legal Statement

Proceedings of the Linux Symposium

Volume Two

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

Conference Organizers

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

Review Committee

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

Proceedings Formatting Team

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