Addressing Large Sequential Read Performance In Linux's Ext2 Filesystem

Kevin O'Connor and Greg Tracy
koconnor@cs.wisc.edu, gtracy@cs.wisc.edu
University of Wisconsin, Madison

Abstract

As the Linux operating system increases in popularity, its use in broadband applications such as streaming media will also increase. The performance of these applications is bounded in part by the throughput of large sequential reads. The authors analyze the current performance of the ext2 filesystem, find its read ahead mechanism to be inadequate, and take steps to improve it. This paper describes these improvements and presents performance measurements that demonstrate the gains made in large sequential reads.

Introduction

The ext2 filesystem is a general-purpose filesystem designed for the Linux operating system [Card 1995]. It provides a standard, Unix-like interface and historically emphasizes extensibility more than performance. One entry point into the ext2 filesystem is through the kernel's *mmap* system call which maps an entire file into a process's virtual address space. We attempted, successfully, to improve the performance of the ext2 filesystem, using the *mmap* interface as the focal point for our analysis and improvement.

We have implemented a new system call that enables the programmer to give the Linux operating system advice about whether it will be accessing a particular file randomly or sequentially. We've also made two important changes to the 2.2 kernel's read ahead mechanism by eliminating blocking read aheads and by initiating read ahead on page cache hits as well as misses. The net effect of our work has been to produce a measurable improvement in data throughput for large sequential reads.

In this paper we describe the changes we made to the Linux kernel in order to improve its read ahead mechanism, and we present performance measurements that show the incremental improvements these changes made in the filesystem's throughput.

Motivation for Improving Large File Read Performance

We found a very compelling reason to improve the performance of large filesystem reads under Linux while investigating the performance of NTFS, the filesystem used in Windows 2000 (Win2k). Using Bonnie[1], a popular benchmark for ext2, it was discovered that NTFS outperformed ext2 by a factor of four when sequentially reading large files. Considering the fact that these measurements were taken on the same hardware[2], we felt there was considerable room for improvement in the ext2 filesystem.

Approach to Improving Large Read Performance

While it was clear in our mind that the ext2's filesystem performance could be improved, we needed to identify a manageable scope for our project. Just taking the ext2 filesystem as a whole was a bit daunting considering the time frame we were working within. We had limited resources in terms of manpower, and

[1] Bonnie is a Linux utility that can be used to measure various performance statistics related to a filesystem, including large sequential read performance. This utility was used by [Card et al] during the original design and testing of ext2fs.

[2] All performance measurements were taken on a PC running a Pentium III with 128MB of internal memory. The machine was equipped with an IDE, 10GB Quantum Fireball hard disk running at 7200rpm.

we felt that trying to tackle that much code in such a short period of time was unrealistic. With these things in mind, we chose to approach the performance issue by using the *mmap* system call as our entry point into the filesystem.

We found that focusing on the *mmap* entry point gave us the proper scope, allowing us to address the performance issues within our time constraints. This did not keep us out of the filesystem code entirely since there were still several occasions in which we needed to investigate the file buffering and driver code of Linux. However, by the time we started analyzing this code we had very specific questions we were trying to answer. As a result, we were able to avoid learning all of the filesystem code while still improving the overall performance for large sequential reads.

To measure the throughput of the filesystem, we wrote a program that is very similar to Bonnie. We called this program *mmapit*. *mmapit* takes a filename as an argument and "mmaps" that file into virtual memory. Like Bonnie, *mmapit* would read the file sequentially and touch every byte. Whereas Bonnie did this with calls to *fgetc*, *mmapit* used the *memcpy* call to copy each byte into an empty bit bucket.

Investigating Read Ahead in the 2.2 Kernel

In our initial foray into the Linux kernel source, we found that the 2.2 kernel had limited support for read ahead. While the code did include a function named *try_to_read_ahead*, it was only called on page cache misses. Moreover, it always read sixteen pages ahead.

We felt we could improve this in two distinct ways. For one, we believed that the number of pages that are read ahead should not be fixed by the kernel but should be configurable by user applications. After all, user applications know better than the kernel how they will be using their files and may be able to improve their performance by advising the kernel about this usage. Second, we believed that it was a mistake for the read ahead mechanism to be used only on page cache misses.

Giving the 2.2 Kernel Some Advice About Read Ahead

Giving the kernel advice about how a file will be accessed is not a new idea. Many flavors of Unix, including BSD, Solaris, and the 2.4 Linux kernel, already support this sort of advice through the *madvise* system call. The typical signature for *madvise* is the following:

*madvise(*unsigned long **addr**, unsigned long **length**, int **behavior***);*

madvise is used in conjunction with the *mmap* system call and advises the kernel about how the user application will use the file previously mapped into the virtual addresses specified by **addr** and **length**. The "advice" is encoded in the **behavior** parameter that informs the kernel whether the user application will be accessing the file sequentially or randomly. Thus, depending on the specified behavior read ahead is either turned on or off.

Linux's 2.2 kernel does not support the *madvise* system call, so we implemented it. This required making a handful of modifications to the system call handling code in addition to writing the actual *madvise* code. We also decided to add an extra parameter to our implementation of the *madvise* system call that, although making the **behavior** parameter redundant, greatly simplified our ability to test *madvise*. This extra parameter explicitly told the kernel how many pages to read when the kernel initiated read ahead[3].

The details of implementing *madvise* are straightforward. When a user application *mmap*s a file into its address space, Linux creates a data structure—a *vm_area_struct*—that describes the range of virtual addresses that will be used by the file. Our *madvise* call simply sets a field in this *vm_area_struct* to indicate the number of read ahead pages.

Effects of madvise on the Page Cache

Having provided read ahead flexibility to user applications, we investigated how varying the number of read ahead pages affected large sequential read performance. We first wanted to assure ourselves that *madvise* was working as expected. We did this by monitoring the page cache while sequentially reading a

[3] The original kernel code was designed in such a way that the number of read ahead pages was necessarily a power of 2. We did not change this.

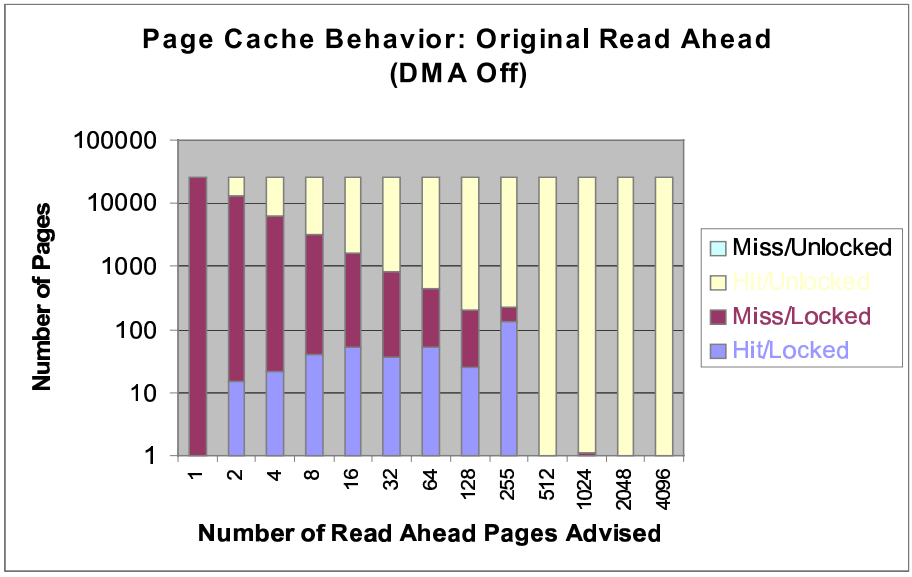

Figure 1: The page-cache behavior with the original read ahead mechanism. As the number of read ahead pages is increased, the number of misses on locked pages decreases as one would expect.

100MB file with different amounts of read ahead. As each page of the file was requested by the user application, we recorded whether or not the page was in the cache and whether or not the page was locked. If the page was in the cache, I/O had been previously started in this page. If the page was locked, that I/O was still in progress. In the worst case, the page would not be in the page cache when the user application asked for it, forcing the kernel to allocate and lock a new page, start I/O on the new page, and wait for the page to become unlocked. We termed such an event a "page cache miss on a locked page." We expected to see page cache misses on locked pages decrease as we increased the number of read ahead pages.

As Figure 1 illustrates, this is in fact what we observed. As we increase the number of read ahead pages, the number of misses on locked pages shrinks to almost zero while the number of hits on unlocked pages increases to nearly 25,600 (i.e., every page of the 100MB file). Figure 1 also illustrates that, up to a point, doing more read ahead also increases the number of page cache hits on *locked* pages. This is not surprising given that the more read ahead you initiate—where by "initiate" we refer to the three actions of allocating a page, locking that page, and requesting I/O in that page—the more likely the user application is to need a page whose I/O has not yet completed.

Effects of madvise on Runtime Usage

What *is* surprising is that we suddenly incur almost no locked pages when reading more than 256 pages ahead—almost every single page of the file is in the cache and unlocked when the user application needs it. Based on this fact alone, we would have expected these cases to have the greatest throughput. We were therefore surprised to discover that throughput in these cases was actually quite poor.

Figure 2 illustrates how the number of read ahead pages affects the total throughput of the system. These measurements were taken with our *mmapit* benchmark program. We remind the reader that *mmapit mmaps* a file into memory and begins reading it in sequentially. Timing measurements were made using the *getrusage* system call to determine how much time was spent in user space and system space during the execution of the file access. Using these measurements along with the total elapsed time we calculated the amount of time the CPU wasn't doing any work on behalf of the user application, which we fittingly refer to as idle time. We inferred that most of the idle time was spent waiting for pages to be read into the page cache since the load on the system while running the benchmark was minimal.

Figure 2 shows the amount of time spent in user, system, and idle spaces. This figure brings up several interesting issues. For one, why does user time vary so greatly when we do different amounts of read ahead? We expect it to be constant. We have been unable to come up with a good explanation for this. A

second question is, why is system time so variable? We came to learn that since the drive was configured to have its DMA functionality disabled by default, the CPU was actually managing all data transfers with programmed I/O. Therefore, a good portion of the time which is measured as system time can be attributed to the CPU transferring data into the page cache. We speculate that for read ahead of 256 or less, the speed at which the programmed I/O takes place is affected by the number of requests outstanding on the drive. Evidently, having roughly 64 outstanding request is most efficient.

For read ahead of more than 256 pages, something else entirely is happening. Even though every page is in the cache and unlocked when the user application needs it, throughput in these cases is almost as bad

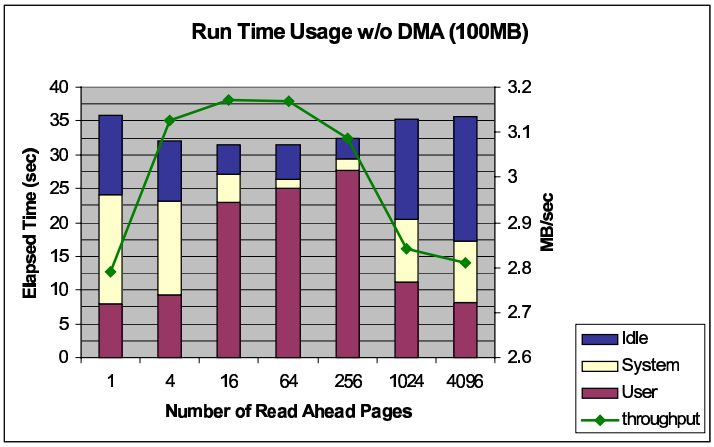

Figure 2: Runtime usage running *mmapit* on a 100MB file with our original Linux configuration. The time spent in user, system, and idle space are displayed for various amount of read ahead pages. The throughput peaks out at only 3.2MB/sec while NTFS peaks at around 13MB/sec.

as doing no read ahead at all. Although though the combined user and system time is small compared to other read ahead levels, the idle time increases by more than enough to negate this decrease in user and system time.

Extreme Advice Analysis

In order to understand why throughput did not improve even though nearly every page of the file was cached in memory and unlocked when the user application wanted it, we dug deeper into the 2.2 kernel code. The answer lay in how Linux handles requests for blocks from the hard drive. When Linux initiates read ahead of *n* pages, it does so in a tight loop that resembles the following:

```
for ( i = 0; i< n; i++ )
        try_to_read_ahead( file, offset + i * PAGE_CACHE_SIZE );
```

However, we discovered that the request queue for the hard drive is bounded at 128 requests[4]. If the above loop sends a read request to the hard drive when the drive's request queue is full, the requestor must wait until space becomes available in the queue. If many read aheads are sent at once (i.e., more than 256), the *try_to_read_ahead* loop effectively becomes one large synchronous read call that does not return until

[4] Empirically we found this number to be much higher, more like 260 requests. We have no explanation for this discrepancy since the code very explicitly sets the queue length to 128. We mention it here as a partial explanation of why the cases affected by "request waiting" occur when we do more than 256 read aheads, rather than 128 as we might expect.

nearly all of the requested read aheads have been read into memory. Hence, by the time control returns to the user application, nearly every read ahead page will be unlocked and ready for use. Of course, the price paid for all these cached and unlocked pages is the idle time wasted waiting for them to be read in during the *try_to_read_ahead* loop.

We devised the following experiment to prove that this request waiting explained the poor throughput we saw when reading more than 256 pages ahead. We created a simple program, the *reader*, that mapped a 100Mb test file into memory and *madvise*'d the kernel to read ahead all 25,600 pages of this file into memory as soon as the user application tried to access the file's first page. After forking the *reader* from a parent process, the parent waited for a pre-determined interval before killing the *reader*. The parent then used *getrusage* to determine how much time the *reader* had spent in user space and system space up to that point[5]. By running the parent process multiple times and each time allowing the *reader* to live for an additional five seconds, we were able to get a rough sense of how the *reader* spent its runtime. Specifically, we could see when, during the course of reading the file, the *reader* had spent time in user space, system space, and idle space. Figure 3 contains the results of this experiment.

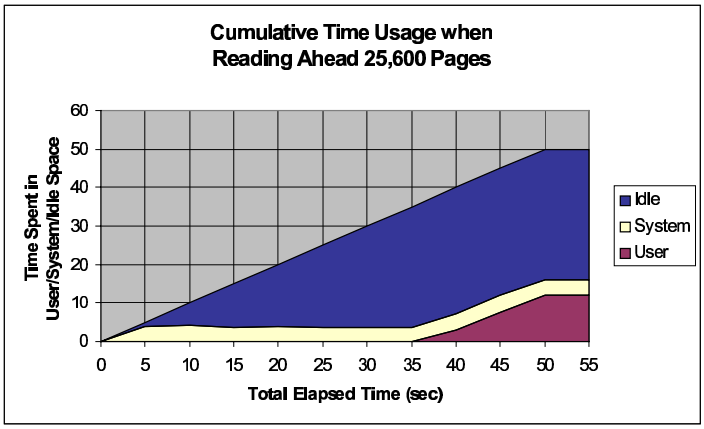

Figure 3: When the current kernel is advised to read ahead 25,600 pages at a time, one observes three periods of activity. From zero to five seconds, almost all of the time is spent in system space. From five to thirty five seconds, almost all of the time is spent in idle space. And from thirty five seconds on, all of the time is spent in user space.

As expected, the stampeding herd of 25,600 read requests generated by the *reader* produces three distinct phases in the *reader*'s lifetime. In the first 5 seconds of its lifetime, nearly all of its time is spent in system space, presumably incurring the overhead of initiating I/O and queuing up many of the 25,600 requests. From 5 seconds to 35 seconds, system time remains relatively constant and the *reader* spends nearly all of its time off the CPU. This time is spent waiting for the system to finish requesting blocks from the disk, which in turn requires waiting for the disk to service pending requests. Finally, 35 seconds into the *reader*'s lifetime, the entire 100Mb has been read into memory and the *reader* hits the CPU running. The *reader* spends the rest of its lifetime in user space since every page it needs is now cached in memory and unlocked, as we saw in Figure 1 for large read ahead.

[5] This method was necessary for two reasons. First, since the user program spent so little time on the CPU the only way we could measure its progress at regular intervals was via a parent process. Second, *getrusage* only returned non-zero numbers for the child process after the child process had finished running.

Understanding Idleness

Although this is an extreme example by construction, it nevertheless serves to highlight the inadequacies of the read ahead mechanism used by the 2.2 kernel. For one, we see that no matter how fast our drive gets, we will always be stuck with some amount of idle time. A faster drive will shrink the interval between 5 seconds and 35 seconds in Figure 3, but it cannot eliminate it (barring a drive with access times on the order of memory accesses). This argument is equally applicable for less extreme amounts of read ahead. If we advise the kernel to read ahead *n* pages in a large file of *N* pages, we will have roughly *N/n* page cache misses on locked pages and therefore *N/n* times where we must wait on the drive to transfer data. A faster drive can shorten this wait, but cannot reasonably be expected to eliminate it. Of course the catch-22 is that that the only way to make *N/n* a small number is to make *n* large, leading to the "stampeding herd effect" and the inevitable request queue waiting that accompanies it.

As evidence for these claims, we used a Linux tool—*hdparm*[6]—to configure the drive to use direct memory access (DMA) when transferring data to and from the drive. After running our *mmapit* benchmark, we saw that the idle time decreased but was not eliminated. Figure 4 shows the run time usage with our hard drive configured to use DMA transfers. Several things become clear. Unlike our measurements in which DMA was disabled, the amount of time spent at the user level is consistent across all advice levels. Since we are using the same version of *mmapit* and testing with files of the same size, we expect this to be the case. Also shown in Figure 4 is the significant increase in the overall throughput. It peaks out above 11MB/sec which is nearly four times faster then our non-DMA configuration. With the DMA feature enabled, we match the throughput obtained with NTFS. Conversely, if we force NTFS to use programmed I/O, the performance drops to the level of our original Linux configuration.

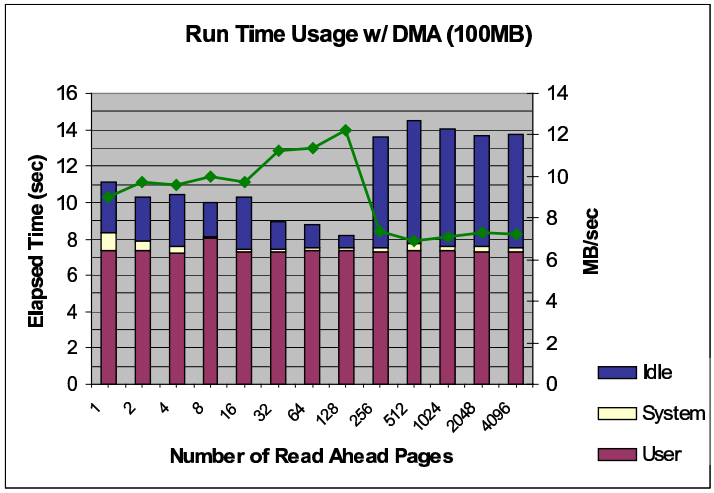

Figure 4: Run time usage for various advise levels and drive DMA enabled. The user time is consistent for each level as it should be. The idle time shows an area to focus on to decrease total elapsed time and increase overall throughput.

It was clear that if we were going to improve the overall performance, we would have to decrease the idle time since the user time is fixed and the system time is negligible. No matter how fast the underlying storage might be, we are still bounded by the inefficiency of initiating read ahead on page misses.

As an example of this, let us look at the case of advice four. Regardless of the storage bandwidth, we still face a cache page miss every sixteen pages. The speed of the drive only helps determine how quickly we can recover from that miss, but not whether we avoid the miss altogether. Our remaining work focused on shrinking these idle ratios by improving the read ahead mechanism. If the idle time could be reduced or

[6] *hdparm* uses *ioctl* calls to communicate with the peripheral devices and configure its firmware to behave in certain ways. Some examples of this include DMA enable/disable and setting read-only permissions.

even eliminated the user time would approach 100% of the total elapsed time, implying that every page of the file was in memory when the user application needed it—the holy grail of read ahead.

Implementing "True" Read Ahead

In our opinion, "true" read ahead meant preemptively reading ahead on page cache hits so that the file data was transferred from the disk while the user application was running on the CPU. This would effectively create a pipeline of file data that could be continuously pumped into the user application, minimizing the amount of time the user application would have to wait on the disk. Implementing "true" read ahead thus required making two changes to the 2.2 kernel. First, as we mentioned earlier, we wanted to periodically initiate read ahead on page cache hits instead of waiting for page cache misses. Second, we did not want to block on read ahead requests. Instead, we dropped them and retried them the next time read ahead was initiated. In this section we discuss in detail the changes we made in these respects.

Non-blocking read ahead:

Although the upper level file system code only supported sending standard read requests to the disk, the low-level driver code supported both "read" and "read ahead" requests. The distinction made between the two was the very one we ourselves wished to make—read ahead requests did not block if they found the drive's request queue full, they simply aborted the request[7]. Unfortunately, the upper level filesystem code was oblivious to the read ahead functionality and always sent standard read requests to the driver even if the request was sent by a *try_to_read_ahead* call (apparently, it wasn't trying very hard). With a few additions to the code in the filesystem level, we were able to bridge the gap between these two layers of code and send read ahead requests when they were appropriate.

Note that non-blocking read ahead requests introduce additional complexity into the kernel. It is no longer guaranteed that when a call to *try_to_read_ahead* returns it will have started I/O on the requested page of the file. Yet if, as we are assuming, the user application will eventually need these pages, we must do our best to make sure that any of the I/Os that are dropped are reinitiated well before the user application needs it. Periodically initiating read ahead on the same group of pages makes this possible, so long as we initiate the read ahead often enough to make up for all the I/Os that are dropped.

Periodically-initiated read ahead:

Periodically initiating read ahead requests brings up two questions: how often should the kernel initiate read ahead and how many pages ahead should the kernel read when it does initiate read ahead? In answering these questions we made the following assumption: when a user application advises the kernel to read *n* pages ahead in a file, that user application is likely processing the file's data *n* pages at a time[8]. For example, in the first pass of an external sort that is using *n* pages of physical memory, the sort might advise the kernel to read *n* pages ahead. Then as soon as it finished sorting one group of *n* pages, the next group of *n* pages could already be available in memory[9].

To accommodate this sort of access pattern we logically divide the *madvise*'d file into groups of *n* pages and always initiate read ahead on:

1) the unread pages in the group currently being accessed by the user application, followed by
2) the group of pages that immediately follows the currently accessed group.

---

[7] Specifically, the kernel would unlock the page that had been reserved for the I/O, leaving this page in the page cache but marking it as not being up to date.

[8] Note that while we designed our read ahead algorithm with this access pattern in mind, it will perform well even if the user application's accesses are not grouped together in this manner, so long as they are sequential. The ability to advise the kernel as to how many pages ahead the kernel should try to read simply becomes less relevant.

[9] To be honest, then, the external sort must be allocated $2n$ pages—n for the current group of data being sorted, and n for the next group of data that will be sorted.

In other words, while the user application is accessing pages in group *i*, all read ahead that is initiated attempts to read the unread pages in groups *i* and *i+1*[10]. To avoid repeatedly initiating read ahead on a page that has already been loaded into memory, we also keep track of the highest page that we have successfully started I/O in (this value thus increases sequentially over the duration of the file access).

We can therefore err on the side of excess when deciding how often to initiate read ahead and not have to worry about incurring much of a performance penalty. If we overzealously initiate read ahead more often than is needed to load the next group of pages into memory, each "extra" initiation is effectively a no-op. On the other hand, if we are stingy in how often we initiate read ahead, we may fall behind the user application's needs, forcing the user application to wait while the drive fetches pages from disk[11]. With this in mind, we elected to initiate read ahead in the following manner: If the kernel is advised to read ahead a number of pages, *n*, that is less than 16, the kernel initiates read ahead every *n* pages beginning with the second page of the file. Otherwise, the kernel initiates read ahead every *P* page accesses, beginning with the second page of the file[12]. An optimal value for *P* could in principle be determined adaptively by monitoring how fast the user application is running through its pages. However, we took the easier, empirical way out and found that, for our test programs, setting $P = 16$ ensured that the read ahead did not fall behind *mmapit's* needs.

"True" Read Ahead Performance

Using our modified read ahead algorithm, we made progress towards reaching our goal of eliminating idle time. With a fixed user time, every second of reduced idle time means an increase in overall throughput. Figure 5 shows the results of the *mmapit* benchmark when running the modified kernel.

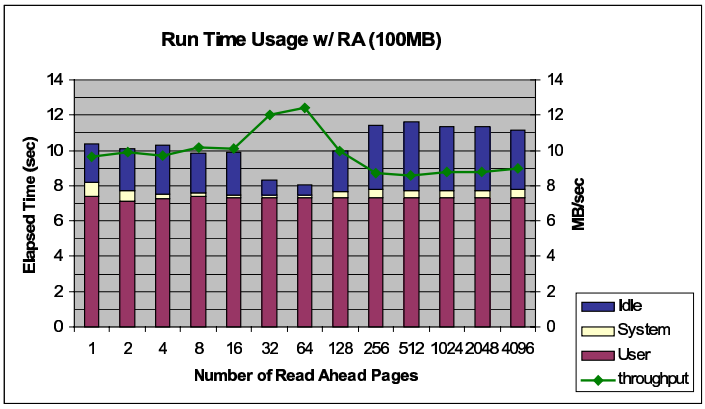

Figure 5: Runtime usage with the read ahead modifications. Idle time is approaching zero for read aheads of 32 and 64 pages (advice levels 6 and 7 respectively).

At an advice level of 64, the amount of idle time has shrunk to just over half a second and the throughput reaches 12.5MB/sec. There may still be some room for improvement since the idle time makes

[10] We always consider initiating I/O in group *i* before $i + 1$ for the obvious reason that if we haven't yet started I/O on all the pages in group *i*, we have no business reading ahead in group $i + 1$! This is especially important when we just begin accessing a file.

[11] Of course, if the user application is using pages at a rate that the drive simply is incapable of delivering, then this waiting is inevitable and read ahead will be unable to reduce it.

[12] For large read ahead groups it would be overkill to start read ahead on the second page access since the first page access would have just filled up the request queue. A smarter approach might be to start reading ahead after some fraction of the group size has been accessed, giving the request queue more time to be emptied. However, as we have mentioned, the penalty for initiating excessive amounts of read ahead is very small, so we elected not complicate things in this manner.

up nearly 10% of the total elapsed time. We speculate that all of the idle time occurs up front while we are trying to establish the pipeline of data. Once this pipeline is established, the user application's page requests are satisfied immediately.

When reading 256 or more read ahead pages, there has been a drastic reduction in idle cost as can clearly be seen in Figure 6. This can be attributed to a more efficient use of read ahead. Since the new algorithm no longer blocks when the request queue is full, the user application is guaranteed to never wait on a request that isn't absolutely needed.

Figure 7 demonstrates the reason behind these reductions in idle time. Note that to "level the playing field" we have plotted the total number of times the user application must wait during its runtime. This includes not only when needed pages are locked but also when the user application must wait on a full request queue. The latter only affects the original read ahead mechanism's numbers of course, and then only in the cases where the kernel reads 512 or more pages ahead. This request waiting accounts for more than 7000 waits in each case.

When the kernel is advised to read 16 or fewer pages ahead, the number of locked pages that the user application must wait for is essentially the same between the two read ahead mechanisms. More pages count as page cache "hits" in the modified mechanism, but ultimately the same number of pages are locked when the user application needs them whether we use the modified read ahead algorithm or not. The advantage of the modified algorithm is that these waits are shorter since we have requested the data in advance of its being needed.

When the kernel is advised to read 64 or more pages ahead, the number of times the user application must wait is reduced dramatically when we use the modified read ahead algorithm. Request waiting is eliminated from the modified algorithm and traded in for a significantly smaller number of waits on locked pages. This can be seen in Figure 7 for read aheads of 512 and more pages. This does not mean that the idle time is necessarily reduced, however, since a request wait is likely to be much shorter than a locked page wait. After all, the servicing of *any* request in the request queue is sufficient to end a request wait whereas a user application waiting on a locked page must wait until one particular request in the queue is serviced. Nevertheless, Figure 6 demonstrates that there is a net savings in idle time. Figure 7 also illustrates that the modified read ahead algorithm's ability to eliminate waits essentially plateaus beyond 512 read ahead pages. This suggests that beyond 512 read ahead pages, the request queue is saturated with requests for the entire time the user application is running.

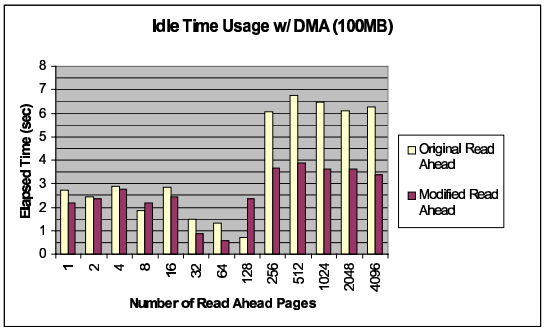

Figure 6: An illustration of the difference in idle time at various read ahead page counts when running the standard read ahead mechanism and our modified version.

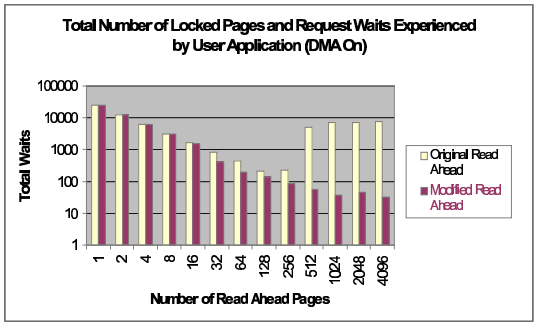

Figure 7: In every case the modified read ahead algorithm is seen to reduce the total number of times that the user application must wait, whether this waiting be due to locked pages ore a full requesti queue.

As for the cases of reading ahead 64 and 256 pages, the only waiting that occurs in either read ahead mechanism is attributable to locked pages. It is therefore a big win that the modified read ahead algorithm reduces this number of waits by more than half when compared with the original read ahead algorithm.

Although Figures 6 and 7 demonstrate that the modified read ahead algorithm is successfully decreasing idle time and increasing throughput, they also raise a question. If the reading ahead of 512 or more pages leads to the least number of waits on locked pages *and* this is the only source of waiting, why

doesn't reading this much ahead give the best throughput? Assuming that we have not missed other sources of explicit waiting, the answer may be that even though we have fewer waits when we read 512 or more pages ahead, these waits *last longer* than when we read, say, 64 pages ahead. We speculate that there is some inherent efficiency in queuing 64 requests at a time that has to do with how the drive's request queue is serviced. However, this is certainly a question that could use further investigation.

Conclusion

In this paper we have presented three shortcomings in the 2.2 kernel's implementation of read ahead. These are:

1) Always reading a fixed number of pages ahead regardless of a user application's needs,
2) Blocking on read ahead requests, and
3) Only initiating read ahead on page cache misses.

We have addressed each of these shortcomings in turn. First, we implemented a simple version of the *madvise* system call that enables user applications to advise the kernel as to how many pages the kernel should try to read ahead when it initiates read ahead. Second we provided the upper-level file system code with the ability to send non-blocking read ahead requests to the low-level disk driver. Lastly, we periodically initiated read ahead on page cache hits as well as page cache misses to increase the chances that data would be available to the user application when it needs it.

Using *mmapit* we have shown that our modifications have improved the throughput of large sequential reads. Given that our *mmapit* benchmark does little more than touch every byte of the file, we are optimistic that our read ahead algorithm could benefit many "real world" user applications that perform more involved operations on their data. In this vein, the authors regret to admit that despite their best intentions they did not have time to test their modified read ahead algorithm against an external sort, an application that could greatly benefit from properly executed read ahead.

Another possibility for future work includes explaining why throughput peaks around 64 and 128 read ahead pages. Such work should begin by relaxing the restriction that the number of read ahead pages be a power of 2. By using a continuous range of read ahead pages, it should be possible to gain greater insight into why the region between 64 and 128 read ahead pages is unique.

Lastly, it might be useful to make our read ahead algorithm more adaptive, varying the amount of read ahead done based on the recent access patterns of the user application. This would eliminate our reliance on so-called "magic" numbers and improve the overall robustness of our read ahead algorithm.

References

[Card 1995] R. Card, T Ts'o, S. Tweedie. Design and Implementation of the Second Extended Filesystem. In *Proceedings of the first Dutch International Symposium on Linux*, ISBN 90-367-0385-9, 1995.

[Rusling 1999] D. Rusling. The Linux Kernel. http://www.linuxdoc.org/LDP/tlk/tlk.html, v0.8-3 1999.

[MM] Linux Memory Management. http://www.linux.eu.org/Linux-MM/index.html, 2000.

[Wilson 1999] P. Wilson. The GNU/Linux 2.2 Virtual Memory System. wilson@cs.utexas.edu, 1999

[PC-guide 2000] C. Kozierok. The PC Guide. http://www.pcguide.com, 2000.

[Cows 1999] B. Vibbor, Tucows Inc. The Linux Ultra-DMA Mini-Howto. http://howto.tucows.com, v3.0. November 1999.

[Haan 1998] P. den Hann. The Enhanced IDE FAQ. http://thef-nym.sci.kun.nl/~pieterh/storage.html#eidefaq, 1998