User-Controlled File Caching in Linux

Alaa R. Alameldeen and David A. Schneider
Dept. of Computer Sciences, University of Wisconsin-Madison
(alaa,das @cs.wisc.edu)

ABSTRACT

The Unix file system and its many versions have been successful so far in providing a powerful yet simple interface to the file system calls. The performance of the file system is greatly affected by its caching behavior. However, UNIX does not allow users to have any control of its caching policies. This means that useful information that a user application might know about, such as its file cache requirements, is ignored, and users cannot optimize cache behavior according to their needs.

In this paper, we studied the design of file caching for a Unix sibling, Linux. We designed and implemented a user-controllable file cache for Linux. The user is enabled to control the Linux file cache by providing a per-file limit on the page cache consumption, as well as a per-file page replacement policy. This allows applications to control their file system performance, and we believe that trusted user applications should be able to make use of new features for their advantage. Experiments were conducted using micro-benchmarks that show the performance of this modified file cache system. It was shown that user-applications can perform much better in certain situations if they were able to control their file cache behavior.

Key Words:
File systems, Unix file system, Linux file system, File cache, caching, user-controllable file caching

1. INTRODUCTION

The Unix Operating system [7] had a wide success since it was developed in the late 1960s-early 70s. Its success is based on its powerful yet simple interface to many system functions. As in any operating system, the file system is a basic part of the OS kernel [5, 7], and it also has a great impact on OS performance and user satisfaction.

Unix is by nature a monolithic operating system, providing many features for users in the way the Unix designers saw best. This might be one of the factors of the Unix success, since most users were satisfied with most of the OS features. However, this does not apply for everything, and the OS can certainly improve its performance by knowing something about the nature of applications submitting demands to it. If the user applications had some control on system resources, they might use it to their advantage and to improves their performance. However, OS designers tend not to give users control of many resources. This is due to three main disadvantages of user control on resources. User control may add security hazards, may affect performance for other users, or may make OS implementation more complex.

Linux is a descendant of Unix that was designed to work in a PC environment. PCs tend to be used by a single-user at a time, which eliminates two of the major disadvantages of user-control on resources. Since all resources are essentially given to the single PC user, affecting performance for other users or causing security hazards is not much of an issue, and should not prevent Linux from giving the sole PC user some control on system resources. As for the third disadvantage, better performance justifies any small sacrifice in system simplicity.

One of the most important aspects of Linux is its caching policies, which have a great impact on file system performance. Caching is implemented at many levels in Linux, but its basic objective is to speed up many system calls by storing some information in memory, rather than reading it from disk. Due to its great impact on performance, the tuning caching policies is an important aspect in file systems research. This was studied by Cao et al. [2]. Successful caching policies are those who can make use of available information in order to determine what to keep or to remove from the file cache. Linux, for example, tries to capture file access patterns (sequential or random) and decide accordingly its caching policy for different files.

However, since user programs can provide useful and easy-to-get information to the OS about their specific caching requirements, the OS can definitely make better use of this information in order to provide the user the required service. The OS cannot easily obtain this information by just observing file access patterns. Users should have the ability to control their file cache behavior, and should be encouraged to do so to get better performance.

An example of a case in which user control on file caching is a decisive factor in performance is described in the following situation: a certain user reads continuously from files A and B, and reads a file C only once. If file C is big enough, it can consume all pages in the page cache, throwing pages of files A and B out of the page cache. This would not be the case if the user was allowed to specify a limit on the maximum number of pages file C can occupy in the cache, so that it does not interfere with files A and B. Another example is when a file is read sequentially multiple times. In this case, LRU replacement might not be the best policy to implement if the number of available pages in the page cache is slightly less than the total number of files. In this case, LRU provides worse performance than other policies, like MRU.

Application hints used to control file caching policies were studied in the literature for various operating systems. For example, Patterson et al. [6] presented aggressive mechanisms that use application-disclosed access patterns to expose and exploit I/O parallelism in Digital's OSF/1 operating system. They used application access-patterns in order to dynamically allocate file buffers among three competing demands: prefetching-hinted blocks, caching-hinted blocks, and caching LRU blocks.

User-controlled file caching was also previously studied in the literature. Cao et al. [1] used a two-level cache management in which the kernel allocates physical pages to individual applications, and each application is then responsible for deciding how to use its physical pages. This suggestion was studied using traces for Ultrix and Sprite, but it has the drawback of increasing application complexity, and migrating functions that are usually located in the kernel to the code of the application. Cao [3] presented a design and validation for a user-controlled file-cache allocation policy, LRU-SP. This policy aims to provide applications with some control on their file cache, while protecting performance of "smart" applications from the "stupid" once.

In this paper, we studied file-caching policies implemented in Linux. We also designed and implemented modifications to the Linux kernel that allow user some control on the file cache behavior. Under the modified system, user applications can specify the maximum number of pages a file can occupy in the file cache, and can specify a per-file replacement policy for replacing pages of a certain file. Experiments were conducted to show that the file cache performs as expected using different micro-benchmarks.

The rest of this paper is organized as follows. Section 2 describes the Linux page cache and the policies it implements. Section 3 presents an explanation of our modifications on the Linux page cache policies. Section 4 describes the micro-benchmarks used to evaluate the new modifications, as well as the methods used in conducting experiments. Section 5 presents performance results which show that our system modifications behave as expected. We conclude and describe possible future extensions in section 6.

2. LINUX PAGE CACHE

Linux makes use of different caching structures in order to make use of locality of reference to speed up the file system implementation. The (Page Cache) caches pages used for normal file data read in the ext2 file system, which is the default file system in Linux. The (Buffer Cache) caches all memory blocks assigned to directories or special files, and blocks to be written to the file. The (Name Cache) provides caching for mappings from filenames to (device number, inode number) pairs.

In this paper, we concentrate on user control on data reads from Linux normal files, which is cached in the page cache. Most of the description that follows was obtained from [4, W1, W2, W3, W4].

In the page cache, all memory pages assigned to an inode are recorded in a hash list, whether they are code pages or memory-mapped file pages [4]. It is managed dynamically. When a new page assigned to an inode is to be loaded into memory, a new page is allocated, inserted to the page cache, and data is read from the disk to that page. Later accesses to contents of this page will not load data from the disk since it is already mapped to memory. When Linux is short of memory pages, it selects pages to replace based on an LRU-type replacement algorithm. The actual implementation of page replacement is done using the clock algorithm.

The next two subsections describe the Linux kernel code for data file read, and its page prefetching strategy that uses the file access pattern (sequential or random) in order to fetch pages from disk before they are actually requested.

2.1 File Read in Linux

A file read from a file in the ext2 file system is handled using the kernel function generic_file_read(). This section provides an outline of steps involved in file read, including code from called functions when appropriate. Page-prefetching related code will be explained in the next subsection. This function is passed a pointer to an inode and a pointer to the file attributes structure in memory, as well as the buffer into which data will be read. The algorithm is described in the following steps:

1. Check whether the data to be read is outside the current readahead window (for data that was previously read during a previous prefetching operation). If not, then the file is accessed randomly and there is no need to read ahead next time, so prefetching is disabled.
2. Adjust the current value for readahead maximum value for other "sequentially-accessed" files, as described in the next subsection.
3. Try to find page in the page cache. If not found, goto step 7.
4. Page is found in the page cache:
   If page is filled or being filled, perform a prefetching operation depending on the previously set parameters for readahead. Otherwise, adjust the readahead limit to the minimum.
5. If page in the page cache is not up-to-date, go to step 11
6. Copy the page to user space, and go to step 13
7. Page is not in the cache:
   If there are free pages in the page cache, go to step 10
8. No free pages in the page cache:
   8.1 Try to allocate pages in the page cache using the clock replacement algorithm.
   8.2 If this succeeds, go to step 10.
9. Failure to allocate pages in page cache: Print out of memory error and exit.
10. Page to be read into is found:
    10.1 Add page to page cache hash structure
    10.2 Update readahead parameters
    10.3 Read page from the disk
    10.4 Go to step 13
11. Page Not Up-to-date:
    11.1Try to read the page once more from disk
    11.2If reading fails, print an I/O error message and exit
12. Adjust file parameters (file pointer, readahead, inode access time, dirty bit, ... etc.)
13. Return

2.2 Page Prefetching in Linux

The function description given above shows that Linux tries to determine whether the file is accessed sequentially or randomly, and decide whether to prefetch more pages (for sequential access patterns) or not (for random access patterns). Prefetching is implemented using the following fields in the open-file data:
f_ramax: maximum number of pages to be read in anticipated mode
f_ralen: maximum number of bytes the last time anticipated mode was used to read pages.
f_rawin: window size for anticipated reading
f_raend: position in file for the first byte unread in anticipated mode
In addition, there are two global constants that are also used to update the readahead limit: MIN_READAHEAD and MAX_READAHEAD. In the Linux kernel version we used, MIN_READAHEAD was set to 3 pages and MAX_READAHEAD was set to 18 pages.

Each time anticipated reads are issued, it tries to read f_rawin pages of the file. For randomly accessed files, subsequent reads will try to read pages not currently in the page cache, and thus deduce that prefetching is useless and it disables read-aheads. For sequentially accessed files, f_rawin is set as the maximum between f_ramax and the number of bytes that are needed to be read, such that it is between the boundaries of [MIN_READAHEAD, MAX_READAHEAD]. For each successful readahead, f_rawin and f_ramax are doubled so as to read twice as many pages during the next read, however, f_ramax is not allowed to exceed MAX_READAHEAD.

3. USER-CONTROLED PAGE CACHE

This section describes implementation of the new features that were added to Linux kernel in order to support user control on page caching policies. These features are concerned with limiting the amount of pages a file can occupy in the page cache, removing all pages of a file from the page cache, and controlling the per-file page replacement policy. A user program can use any of these features by calling a file control function using appropriate arguments.

3.1 Limit on Number of File Cache Pages

A user application might need to set a limit on the number of pages a file can occupy in the page cache, so as not to degrade cache performance for other files. The user application can achieve that goal using the function fcntl() in a C program, as follows:

```
fcntl (fd, F_SETMAX, New_Max);
```

where fd is the open file descriptor, and F_SETMAX is a new file control function type, equal to 12, and New_Max represents the maximum number of pages this file should occupy in the page cache. The actual way this was implemented is summarized in following changes:

1. Two new fields were added to the open-file structure, f_nrpages and f_maxpages. f_nrpages represents the current number of pages the file occupies in the page cache, and f_maxpages represents the maximum number of pages that file can possibly occupy.
2. When a file is opened, f_nrpages is set to 0 (since the file currently has no pages allocated in the page cache), and f_maxpages is set to a default of 10,000,000 pages (which can be up to 40GB for each file since the page size is 4KB).
3. When a page is read to the page cache during a read or an anticipated read operation, f_nrpages is incremented accordingly.
4. Anticipated read operations cannot proceed past the number of pages allowed for the file to occupy. This is done by adjusting f_rawin so as not to exceed the difference between what the file currently has and what the file could possibly have.
5. Functions were implemented that select a page for replacement in the page cache from pages of a particular file. An appropriate function is called depending on the current page replacement policy when a file needs more pages than what it can occupy. This is known by comparing f_nrpages with f_maxpages.
6. A new file control function type (F_SETMAX) was added to the code of fcntl() that sets f_maxpages for a particular file.

3.2 Removing All Pages of a File from Page Cache

This feature allows a user to remove all pages of a file from the page cache, in order to free pages for other files. This feature was basically implemented in order to allow us to start testing the page cache anew without being affected by previously read pages, instead of rebooting the system or unmounting then mounting the root file system each time we want to conduct new experiments. A user application that wants to use this feature can do so by calling:

```
fcntl (fd, F_INVPAGES, 0);
```

where fd is the open-file descriptor and F_INVPAGES is a constant set to 13.

The actual implementation of this function was easy, since it just involved calling the function invalidate_inode_pages() on this file's inode. It is not obvious to us how user applications might use this feature, but it can be useful for testing page cache performance (which was our case!).

3.3 Per-File Page Replacement Policy

This feature allows user applications to select a per-file page replacement policy when they set a maximum number of allowable pages for a file. When a file demands a page to replace from its own page pool, one of these policies is followed. We implemented two policies: Least-Recently-Allocated (which is an approximation for LRU for sequential reads) and Most-Recently-Allocated (which approximates MRU for sequential reads). The default is least-recently allocated.

A user application can select the per-file replacement policy by calling:

```
fcntl (fd, F_SETREPL, Policy);
```

where fd is the open-file descriptor and F_SETREPL is a constant set to 14. Policy should be set to 0 for least-recently allocated and to 1 for most-recently allocated.

The actual way this was implemented is summarized in following changes:

1. A new field is added to the the open-file structure, f_repl. This field represents the selected replacement policy.
2. Two function were added to the page cache. These are: invalidate_inode_pages_mru and invalidate_inode_pages_lru. The first function frees a page from the front of the linked list of pages pointed to by the inode field i_pages, and the linked list is updated appropriately. Since the front of this linked list represents the most-recently-allocated page, this is the target for replacement under this policy. The second function frees a page from the rear of this linked list, which is the least recently allocated page.
3. When a file is open, the default of f_repl is set to least-recently-allocated.
4. A new file control function type (F_SETREPL) was added to the code of fcntl() that sets f_repl for a particular file.

Other possible replacement policies to implement are actual LRU or MRU. These can be implemented by adding a new field to the page descriptor to indicate the last access time, but this would have complicated the implementation of both file reads and replacements. Linux does not actually implement true LRU, but a clock algorithm that approximates it, due to the overhead of setting the time. We could have used a similar clock algorithm for LRU, which requires maintaining a per-file clock. This part is left for future work.

4. EXPERIMENTS SETUP

4.1 Machine and OS Configuration

Modifications were made to kernel of Red Hat Linux 6.1 version 2.2.3. The computer on which the kernel resided had a 90MHz Pentium processor, 162MB RAM, and 524MB-disk space, of which there was only 50MB of free space. This fact is significant because in its present state, the single largest file that can be read is 50MB, which can fit entirely in main memory with room left over for other files being read. The size of a page was 4KB (4096 bytes). All benchmarks were performed with this page size in mind; e.g., 100 pages of cache memory can hold 409600 bytes of data.

4.2 Discussion of Used Micro-benchmarks

A number of micro-benchmarks were used to both show that the modifications made to the Linux kernel actually enhanced the functionality of the kernel in the way it was intended and to show how these modifications can affect the performance of applications. Three micro-benchmarks were used; each of which was designed to accomplish these two "tasks".

4.2.1 Benchmark 1: Testing Limits on Maximum Pages of a File

The first set of micro-benchmarks was used to show that a user application could limit a particular file's maximum allotment of pages in the page cache through a simple function call prior to the call to read. This set of micro-benchmarks performed two reads of the same data in succession of one another. The first read was always performed with a "fresh" page cache, i.e., one that is guaranteed not to contain any pages of the file to be read. The cache was made "fresh" by a call to a file control function implemented specifically for this purpose, which was described in section 3.2. Essentially, this file control function removes all pages of a particular file from the page cache. The second read was performed on the same file, and, due to the first read, the page cache would have contained pages of this file.

The number of bytes that were read was the same between the first and second reads. The pairs of reads were divided into four categories based on the type of reading performed on the file in the first read and the second read. The categories are as follows (first read, second read): read-sequential, read-sequential (RS-RS); read-sequential, read-random (RS-RR); read-random, read-sequential (RR-RS); and, read-random, read-random (RR-RR). Each of these benchmarks was run with a varying number of maximum cache pages allotted to the file that was read. The maximum number of pages allotted to a file was modified by a call to the file control function described in section 3.1, which limits a specific file's maximum allotment of pages in the page cache. We used the most-recently-allocated page replacement policy for this set of benchmarks. The tests were done multiple times and average values are represented

In general, for each of the first set of micro-benchmarks, it was expected that a certain pattern of read behavior would develop. It was expected that all of the first reads from the file, whether read-sequential or read-random, would be "slow" because an I/O operation needs to take place for every new page that is read. The maximum number of cache pages allotted to the file would not affect the speed of the first read when read-sequential was performed because no matter what the size of the cache space is, pages of the file currently being read would never be in the cache. Conversely, as the maximum number of pages allotted to the file increases, the read-random times of the first read would improve. This is because as the size of the cache spaces grows, there is a better chance that the next random read would be on a page that was recently read into the cache. For the second read, it was expected that the times would be "fast" if all of the pages fit into the cache from the first read, and similar in speed to the first read if the number of cache pages allotted to the file was greatly smaller than the number of pages of data that was read. Finally, in general, it was expected that read-random would be slower than read-sequential unless it was a second read and all of the pages of the file fit into the cache.

4.2.2 Benchmark 2: Testing Per-File Page Replacement Policy:

The second set of benchmarks also was intended to show that the modifications made to the per-file page replacement policy did what they were supposed to do. Here, we compared the performance of a least-recently-allocated page replacement policy, which we will refer to as LRU, versus the performance of a most-recently-allocated page replacement policy, referred to as MRU. For this benchmark, two identical large sequential reads were performed in succession. The page cache was refreshed as previously done to ensure that no pages reside in the cache at the beginning of each pair of reads.

The number of pages read from the file was 4000 pages, and the maximum number of allocated pages to that file was one page less, 3999. Under MRU, the second read should suffer from only two misses. Due to page 4000 replacing page 3999 at the end of the first read, all pages from 1-3998 will be present in the cache for the second read. LRU on the other hand should have worse performance due to the fact that page 4000 replaces page 1, which is going to be read immediately at the beginning of the second read, and so on. We expected that the performance of the first reads would be about the same for LRU and MRU, but that the performance of the second reads would be better for MRU than for LRU.

4.2.3 Benchmark 3: Testing Reads from File of Different Sizes

The third and final set of micro-benchmarks is really just one benchmark that was used to show how the modifications that were made could affect the performance of applications. The scenario that was simulated was that of a user running many applications that require many small reads, possibly random, and at some point in the middle of the many small reads, a large file is read sequentially. Finally, the many small reads continue as before. In this scenario, if the Linux kernel was unmodified, then the large file read in the middle of the small files could "blow away" the cached pages of the small files and significantly degrade the performance of subsequent small file reads. With the ability to limit the large file's maximum allotment of cache pages, the small files' pages could remain in the cache upon read of the large file which would allow the performance of subsequent small file reads to be enhanced.

Unfortunately, because the free disk space (50MB) is less than that of main memory (162MB) on the computer we used, this benchmark could not be properly conducted. We were unable to read from a file that was big enough to "blow away" the cache, and because of this we were unable to show that this behavior occurs in the first place. This prevented us from showing that our modifications actually would improve performance for such a situation. We attempted performing some additional modifications to the kernel which would have "reduced" the amount of perceived main memory. This would have given the illusion that our "big" file of 40MB would be larger that the altered main memory of 32MB. Our attempts at limiting the perceived main memory failed and thus we were unable to test this benchmark for the version of unlimited big file pages.

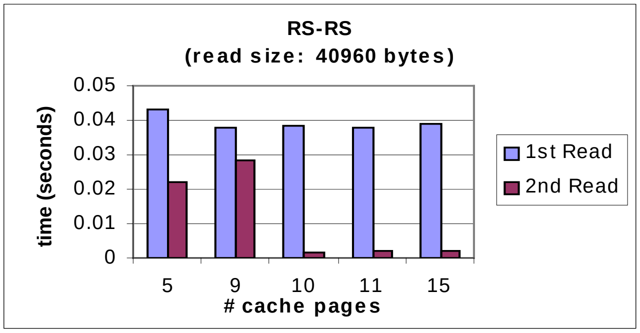
Had we had enough disk space, we would have tested this benchmark by performing a single initial sequential read to 20 small files (each 1MB) followed by one sequential read to one large file (1GB), and then again a single read to the 20 small files. Each run of the benchmark would have started by "refreshing" the page cache through the aforementioned file control function for all small files and the one large file. For each run, a varying maximum number of cache pages would have been allotted to the large file while the small files would have been statically allotted a maximum of 1000 cached pages, far more than needed to fit the all pages of the small files into memory.

It is expected that if the maximum number of cache pages allotted for the large file exceeds the amount of memory, then reading this large file would degrade the performance of the second small file reads. The performance of the second small file reads would then resemble that of the first reads when the cache was fresh. If more restrictions are put on the maximum number of cache pages allotted to the large file, then performance of the second small file reads should get better, up to the point that the big file read has no effect on performance at all.

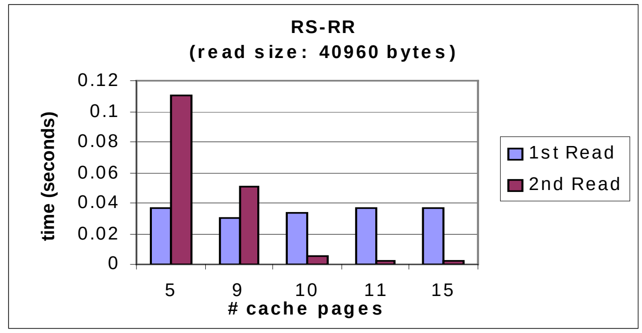
5. PERFORMANCE EVALUATION

5.1 Benchmark 1:

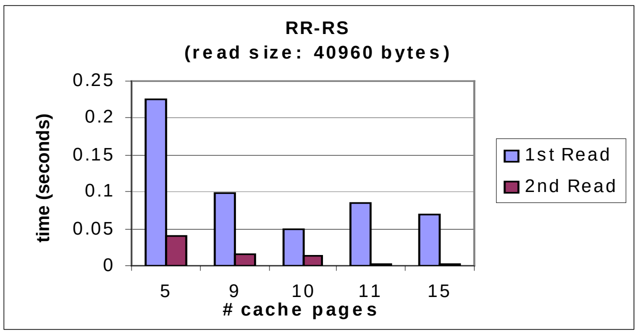
The first set of benchmarks generated results consistent with our expectations. Figures 1-3 show the results of reading a 40 KB file (10 pages), 400 KB file (100 pages), and 4MB file (1000 pages) respectively. Maximum number of page cache pages allotted was varied for each file size. Each figure depicts a different category based on the type of first read and the type of second read. The first reads for all categories have almost the same performance for the same file size. When second reads are sequential reads, the speed of reads improve if the file has more than its requirement of pages. For files with less maximum pages in the cache than what they require, speed of reads is affected, since not all pages are in the cache, and some reads have to go to disk. There is an observed difference in the case when a file has only one page less than its requirement from the case where it has exactly its page requirements. When the second read is a random read, it is more severely affected when the file has much less pages than its requirement.



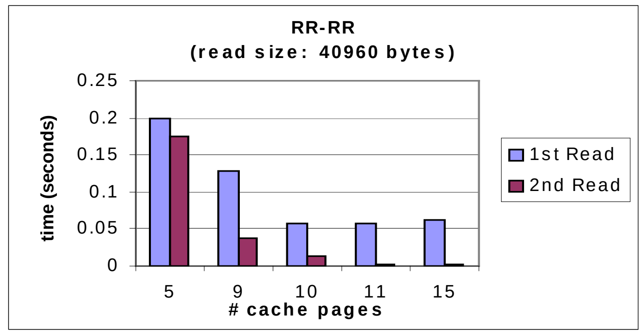
(a)



(b)

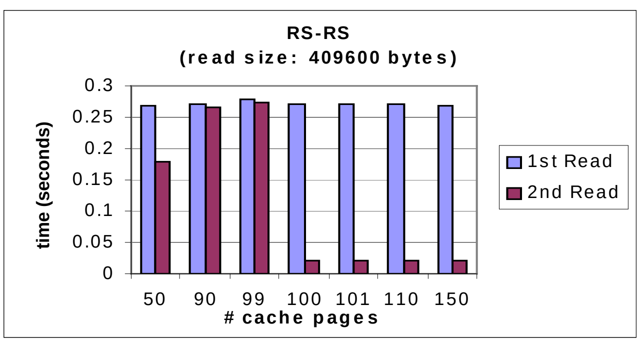


(c)

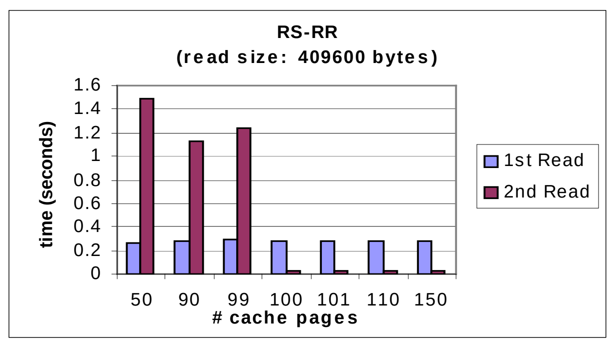


(d)

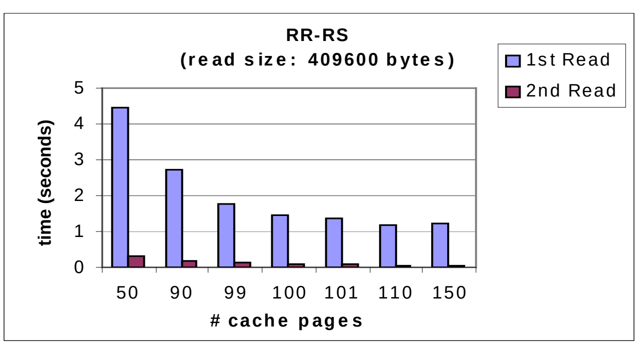
Figure 1: Benchmark 1 for a 40K file



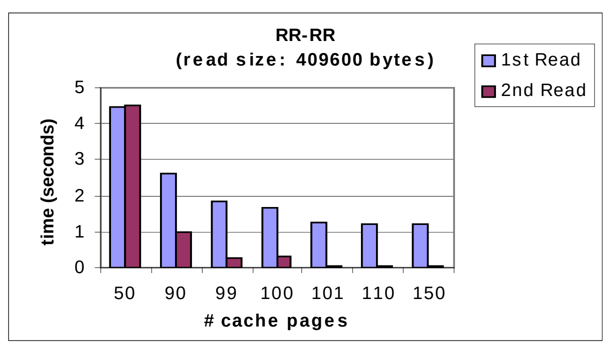
(a)



(b)

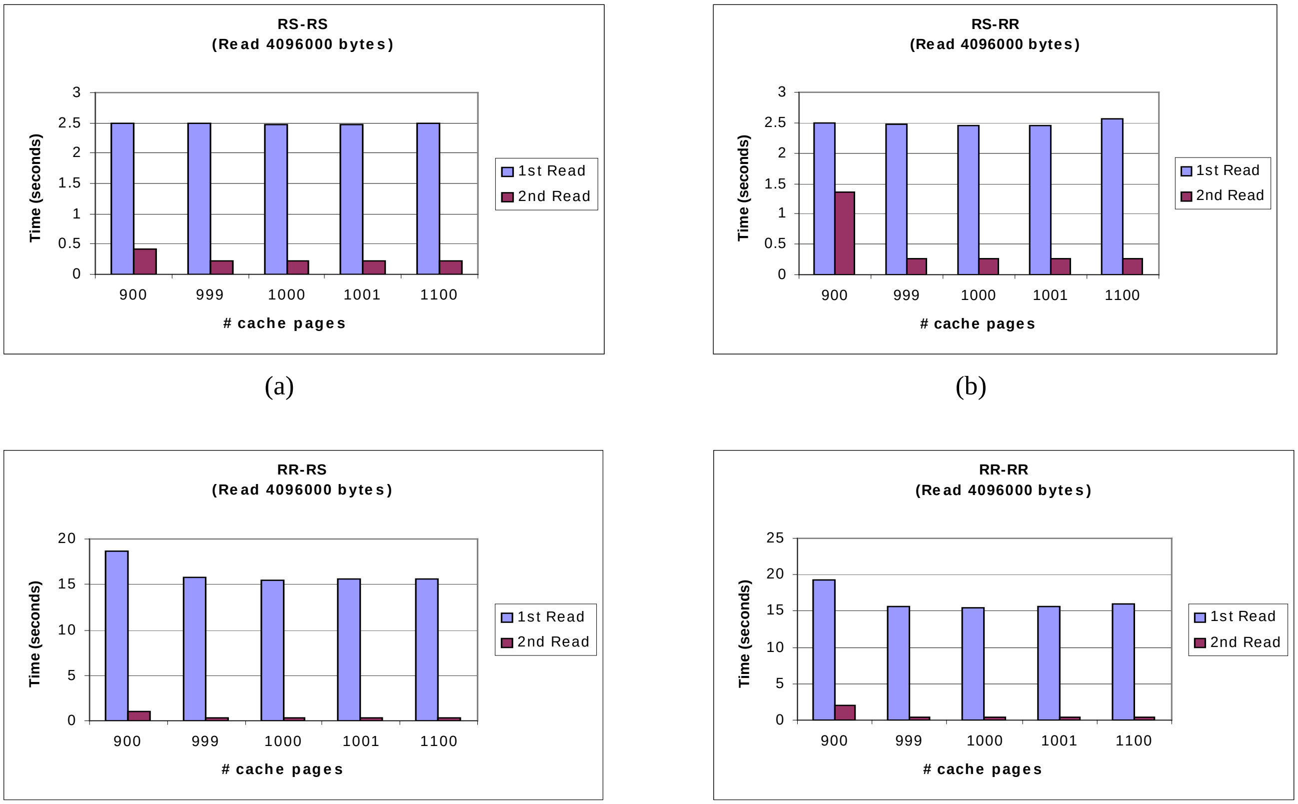


(c)



(d)

Figure 2: Benchmark 1 for a 400 KB file



(a) (b) (c) (d)

Figure 3: Benchmark 1 for a 4 MB file

5.2 Benchmark 2

Table 1 summarizes the results obtained from performing a comparison between a least-recently-allocated page replacement policy versus a most-recently-allocated page replacement policy. As we expected, the times for the first reads for both policies were similarly slow. The times for second read did show that MRU consistently has faster times, however, the difference in times for the second read between the two policies was not as great as anticipated. This is due to read-ahead optimizations done by Linux and described in Section 2.2.

Average times (seconds) for 1st and 2nd sequential reads of 16MB (4000 pages) (the file that was read from was given a maximum allotment of 3999 pages)		
	MRU	LRU
1st Read	8.775222	8.736969
2nd Read	0.736589	0.762074

Table 1: Benchmark 2 Performance

5.3 Benchmark 3

Table 2 shows the results of running the benchmark on an unmodified version of Linux along with the version of Linux containing our modifications. It should again be noted that there was not enough disk space available to allow for a big enough file to exist such that a large sequential read from it would "blow away" the cache. Thus, as is seen in the unmodified linux, the second small file reads (1MB each) are unaffected by the the "large" (20MB) sequential read existing between the first and second small file reads. Our test on the modified version of Linux unsurprisingly showed approximately the same results. As discussed earlier, in order to make this benchmark significant and have relevant data, more disk memory space is required.

	Unmodified Linux	**Modified Linus**
1st Read (20 small files-1MB each)	17.313 seconds	17.141 seconds
2nd Read (20 small files-1MB each)	3.866 seconds	3.659 seconds

Table 2: Benchmark 3 Performance

6. CONCLUSIONS AND FUTURE WORK

In this paper, we studied file-caching policies implemented in Linux, and implemented some features that enable user-application to have some control on file caching policies. Experiments were conducted that show that these modifications behave as expected.

Linux provides an implementation of file caching to speed up disk I/O speed. The Linux page cache is responsible for handling normal file reads, and it captures recent references to file pages in order to read them from memory when they are accessed next. Linux page cache captures file access patterns and decides whether to prefetch more pages than required or not when the file is accessed sequentially or randomly. We described how Linux implements reads and anticipated reads.

Our new features enable the user application to have some control on the file caching policies. An application can limit the number of pages allocated to a file in the page cache. It can also choose to remove all pages of a particular file from the page cache, and it can control the page replacement policy on a per-file basis. Experiments were conducted using micro-benchmarks in order to verify that our new features perform as they are supposed to. Results show this fact, and show that a certain kind of user control can be achieved to tune file caching behavior to the benefit of a particular user application.

In general, it is a good idea to give user applications some form of control on operating system resources when this does not interfere with other users. Implementing some kernel functions to provide these features is normally better than having each application write its own resource-handling code, since this latter approach greatly increases the complexity of user applications. "Smart" applications can make use of added features in the system to improve their performance without affecting their complexity, whereas other applications that choose not to use these features are not affected.

Other possible features can be added to the user-controlled interface of file caching include the following. User applications can be given control on prefetching policies, by setting upper and lower bounds on the number of pages read ahead in anticipated mode. More replacement policies (as random, true LRU and MRU) can be also implemented. More rigorous tests need to be conducted on a suitable platform in order to prove the usability of these modifications. Benchmark 2 would be very useful to verify the expected behavior from Linux and the behavior of our added features.

ACKNOWLEDGMENTS

We would like to thank Professor Remzi Arpaci-Dusseau for his helpful comments and hints throughout the project. Thanks are also due to Sambavi Muthukrishnan for sharing the “PC” in the crash and burn lab.

REFERENCES

[1] Pei Cao, Edward W. Felten, and Kai Li, “Application-Controlled File Caching Policies,” Proceedings of the USENIX Summer 1994 Technical Conference.

[2]Pei Cao, Edward W. Felten, Anna Karlin and Kai Li, “A Study of Integrated Prefetching and Caching Strategies,”CS-TR-94-479, Princeton University (also in Proceedings of SIGMETRICS/Peformance '95).

[3] Pei Cao, “LRU-SP: An Allocation Algorithm For Application-Controlled Caching,” Proceedings of the 1997 Grace Hopper Celebration of Women in Computing Conference, May 1997.

[4] Remy Card, Eric Dumas and Franck Mevel, “The Linux Kernel Book,” John Wiley & Sons, 1998.

[5] M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry, “A Fast File System for UNIX,” ACM Transactions on Computer Systems, Vol. 2, No. 3, pp. 181-197, August 1984.

[6] R.H Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, “Informed Prefetching and Caching,” Proceedings of the Fifteenth Symposium on Operating Systems Principles, Copper Mountain, CO, pp. 79-95, December 1995.

[7] D. M. Ritchie and K. Thompson, “The UNIX Time-Sharing System,” Communications of the ACM, Vol. 17, No. 7, pp. 365-375, July 1974.

REFERENCES ON THE WEB

[W1] Linux Cross-Reference, http://lxr.linux.no/
[W2] Linux Kernel Configuration Help Texts, http://math-www.uni-paderborn.de/~axel/config_help.html
[W3] Linux Source Navigator, http://metalab.unc.edu/linux-source/
[W4] The Linux Kernel Archives, http://www.kernel.org/