Evaluating the Effectiveness of REDCAP to Recover the Locality Missed by Today's Linux Systems

Pilar González-Férez
Universidad de Murcia
pilar@ditec.um.es

Juan Piernas
Universidad de Murcia
piernas@ditec.um.es

Toni Cortes
U. Politècnica de Catalunya
and Barcelona Supercomp. Center
toni@ac.upc.es

Abstract

In order to know how different conditions influence the behavior of the RAM Enhanced Disk Cache Project (REDCAP), we have analyzed the impact of the file system and the REDCAP cache size. The results show that, for workloads which exhibit some spatial locality, the application time can be reduced by more than 80% for file systems that split the disk into block groups, while for those that do not use this division the reduction can be more than 55%. REDCAP has the same performance as a traditional system for those workloads with a random or sequential access pattern. The experiments also show that the cache size can determine the results, depending on the file system and the number of processes.

1. Introduction

One of the most important aspects in the design of a built–in cache (*disk cache*) of a disk drive is its size. Nowadays the size does not meet the system designers recommendations (0.1%-1% of the disk capacity [5, 3]). Indeed, it is still rather small compared to the disk size (a disk of 500 GB usually has 16 MB of cache, only a 0.003%). There are two main reasons for this small size: a tradeoff between cache size and cost; and space limitations.

One way of solving this problem is to integrate a larger cache in the disk, but this decision has to be taken by the manufacturers, and the current technology for hard disks suggests that this increase is not going to happen in the short term. Another option is to use the RAM Enhanced Disk Cache Project (REDCAP) [2] which solves the problem by using a small part of the main memory. REDCAP is a RAM-based disk cache which emulates the behavior of the disk cache and tries to benefit from its read–ahead mechanism, and takes advantage of the disk read requests by prefetching adjacent disk blocks.

In the first analysis of the REDCAP behavior [2], the results obtained proved that REDCAP, by exploiting the principle of locality of reference, takes advantage of the organization in block groups performed by some file systems. They also showed that, with a small portion of the main memory, it is able to considerably reduce the I/O time of the disk read requests, achieving reduction of up to 80%. Our first study had three limitations. The first one was that only one file system, Ext3 [8], was used in our tests. The second one was that, although different cache configurations were tested, all of them had the same cache size, 8 MB. And, the third one was that there were some performance problems in some workloads that were not entirely solved, and the maximum improvement was not achieved.

In this work we have studied the REDCAP behavior under different file systems and with two cache sizes. REDCAP has also been enhanced in such a way that the improvements achieved are now greater than that obtained by the previous version in many workloads.

Our results show that REDCAP can greatly reduce the I/O time of the disk read requests, for many workloads and any file system. The best results are achieved for those file systems which split the disk into several groups with improvements of up to 83%, whereas for those that do not use this division, improvements of up to 57% are achieved. On the other hand, REDCAP has the same performance as a traditional system for those workloads which have a random or sequential access pattern. The experiments also show that the cache size can determine the results depending on the file system and the number of processes. However, both cache sizes present a similar behavior in many cases, what suggests that the REDCAP cache size should be dynamic and dependent on the workload.

2. The RAM Enhanced Disk Cache Project

The RAM Enhanced Disk Cache Project (REDCAP) [2] is a new cache of disk blocks which greatly reduces the

978-1-4244-2818-2/08/$25.00 ©2008 IEEE

I/O time of the read requests. It introduces a new level in the cache hierarchy (the REDCAP cache), just between the page cache and the disk cache, which works as an extension in the main memory of the disk cache. Our cache imitates the behavior of the disk's one by prefetching some consecutive blocks in such a way that it takes advantage of the read–ahead mechanism of the disk drive. The prefetching is performed only when a read operation takes places and a cache miss occurs. It is not performed during write requests or on a cache hit. REDCAP also implements an activation–deactivation algorithm to control the performance achieved by its cache. This algorithm compares the time that the REDCAP cache needs to process the requests, with the estimated time to process them without cache, and turns the cache on/off accordingly. The algorithm is independent of the underlying device, because it only takes into account the I/O time of the issued disk requests. REDCAP does not take part in writes, it only updates its cache, and sends write requests to disk.

3. Experimental Results

In order to perform the study, we have used our REDCAP Kernel (a Linux Kernel 2.6.14 with REDCAP) and the vanilla Linux Kernel 2.6.14 without REDCAP ("the Original Kernel"). We have run 5 benchmarks with both kernels, and the results have been compared. A description of this benchmarks can be found in [2]. To trace disk I/O activity, the kernels record information about the requests, and the REDCAP kernel also about its cache.

Our experiments are conducted on a 800 MHZ Pentium–III system with 640 MB of main memory and two disks. The system disk, with the Fedora Core 4 operating system, is used to collect traces for the study. The test disk is a WD Caviar WD1200BB, with 120 GB of capacity and 2 MB of cache. The test disk has only one partition and the file system used depends on the tested configuration. The Complete Fair Queuing (CFQ) scheduler has been used in all the experiments.

In order to analyze the impact that the cache size and the number of segments have on the performance achieved by REDCAP, two configurations has been used. In the first one, the REDCAP cache size has been fixed at 8 MB, which is four times as large as the size of the disk cache, although its memory utilization is less than 1.5% of the main memory. In the other one, the cache size is 16 MB, eight times as large as the disk cache and less than 3% of memory utilization. In both cases, the segment size has been fixed at 128 KB, which showed the best behavior in our early tests and is the maximum request size allowed by the operating system. Therefore, the tests have been carried out with the configurations: **64x128KB** (64 segments of 128 KB) and **128x128KB**. The initial state of REDCAP is active.

The file system used determines the access pattern seen by the disk drive to a large extent. So, to evaluate how the file system influences the REDCAP behavior, five file systems with different features have been used: Ext2 [1], the default file system in Linux, and four with journal: Ext3 [8], XFS [6], JFS [4], and ReiserFS [7]. All of them are integrated in Linux 2.6.14 by default. We have used the default options for both formatting and mounting. It is important to note that our intention is to study the throughput achieved by REDCAP with each file system. We do not try to compare the results obtained by a file system with those obtained by others.

We have performed five runs for every benchmark, system configuration and file system. The mean results are showed. The confidence intervals for the means, for a 95% confidence level, are also included as error bars. The computer is restarted after every run, hence all benchmarks have been performed with a cold page cache and a cold REDCAP cache. The figures show the improvement in application time achieved by REDCAP with respect to the Original Kernel. Note that if REDCAP achieves a great improvement with a given file system, this does not necessarily mean that this file system has the best behavior.

Linux Kernel Read The application time improvement achieved by REDCAP with respect to the Original Kernel in the *Linux Kernel Read* benchmark are presented in Figures 1.a and 1.b as a function of the file systems used. For JFS, this test could not be executed for 32 processes because the computer ran out of memory.

The REDCAP results are always better than the Original Kernel ones, and the improvement becomes greater as the number of processes increase in almost all the cases, getting a reduction of up to 83%, and being the cache almost always active. The number of processes and the number of REDCAP segments could explain this fact. There are always more REDCAP segments than processes, while there are usually more processes than disk cache segments, which are continually evicted.

Except for ReiserFS, the results obtained by both REDCAP configurations are very similar, although the 128x128KB one achieves slightly better improvements. One reason why the increase in the cache size does not involve the same amount of improvement is because the cache in the 64x128KB configuration is almost always active and REDCAP is already taking maximum advantage of the blocks prefetched. Another reason is that the number of segments is greater than the number of processes.

If we compare the results obtained by REDCAP for ReiserFS with those obtained for the other file systems, we can see that REDCAP achieves a great performance for 1, 2, 4 and 8 processes, but its performance is relatively small for 16 and 32 processes. The main reason for this behav-

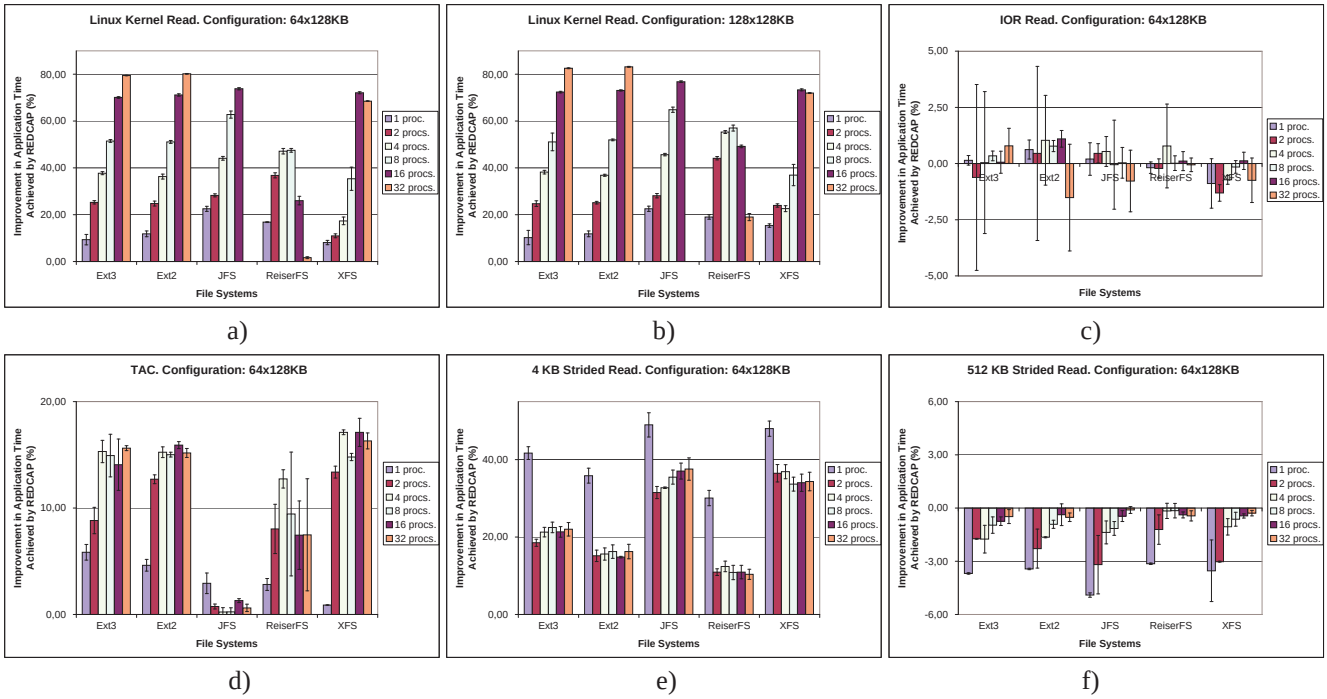

Figure 1. Improvements achieved by the REDCAP Kernel as compared to the Original Kernel.

ior is the structure of ReiserFS, which produces apparent random accesses. In a system without REDCAP, the random requests do not benefit from the disk cache which is not lager enough and causes the prefetched disk blocks to be evicted before being used. However, in a REDCAP system, for 1, 2, 4 and 8 processes our cache is big enough and many disk blocks are read before the corresponding segments are evicted. For 16 and 32 processes, the REDCAP cache is not large enough either, and many prefetched segments are evicted before being reused. The algorithm is also unable to decide the proper state of the cache, and it turns the cache on and off several times. The differences between the results achieved by the two REDCAP configurations confirm the segment eviction problem, and point out that with the 64x128KB configuration the number of segments is not large enough.

IOR Read Figure 1.c depicts, as a function of the file systems used, the application time improvement achieved by REDCAP with respect to the Original Kernel, for the *IOR Read* benchmark and the 64x128KB REDCAP configuration. The results obtained for the 128x128KB configuration are almost identical.

With all the file systems, the behavior of REDCAP is very similar to the Original Kernel one. Nevertheless, the confidence intervals are pretty big. We can hence say that statistically the REDCAP Kernel and the Original Kernel have the same performance. This benchmark has a sequential access pattern, and the prefetching techniques used by both the Original Kernel and the disk cache are optimized for this kind of pattern. The contribution of our method is rather small, so the activation–deactivation algorithm turns the cache off, which is inactive almost all the time.

TAC The results for the application time achieved by REDCAP, as compared to the Original Kernel results, for the *TAC* test and for the 64x128KB configuration, are presented in Figure 1.d as a function of the file systems used.

The REDCAP kernel always perform better than the Original Kernel with all the file systems (except JFS), achieving improvements of up to 17%. The cache is almost always active. The results obtained by the 128x128KB configuration are quite similar and, therefore, a cache size of 8 MB is enough for this test.

With JFS, the behavior of REDCAP is quite similar to that of the Original Kernel. The reason can be found in the *Allocation Groups* used by JFS to divide the disk, and in the way the data blocks are allocated in these groups. In our case, the allocation group size is 1 GB, which is also the size of the files read by *tac*. Although all the files are created in parallel, almost all the blocks of a file are stored in the same allocation group as only one extent. This block allocation benefits not only the REDCAP prefetching, but also the disk cache prefetching (the operating system does not detect the backward access pattern). If the cache was always active, the average improvement would be 4.6%. However, our algorithm is not able to detect this small benefit, and the cache is off all the time. It is interesting to

note that for the other file systems, on creating the files in parallel, the blocks are allocated in a more interleaved way. Therefore, benefit is gained by the prefetching of REDCAP, while the disk cache prefetching is not so beneficial.

4 KB Strided Read Figure 1.e shows the results for the application time achieved by REDCAP with the 64x128KB configuration as compared to the Original Kernel in the *4 KB Strided Read* test as a function of the file systems used.

For this access pattern, REDCAP always performs better than the Original Kernel, which does not detect this access pattern. We get improvements of up to 49%, although the result strongly depends on the file system used. The 128x128KB configuration obtains quite similar results.

Since we have enhanced our first implementation, these results are much better than those obtained in our first analysis [2].

512 KB Strided Read The results obtained by REDCAP in the *512 KB Strided Read* benchmark are presented, as a function of the file systems and for the 64x128KB configuration, in Figure 1.f. The results for the 128x128KB configuration are almost identical.

REDCAP has a quantitative similar behavior in both configurations and for all the file systems, but it does not perform better than the Original Kernel. For 1 and 2 processes, the problem is that the application time is rather small, and although our cache is always inactive, the time initially lost when it is still active cannot be recovered later. For 4, 8, 16 and 32 processes the loss can be considered negligible, and it is due to the time employed to simulate the cache behavior when it is inactive. In all the cases, the algorithm turns the cache off on the first chance and never turns it on again. As in the above benchmark, the Original Kernel does not detect this access pattern nor does it implement any prefetching technique.

4. Conclusions and Future Work

In this paper we have presented several experimental results obtained by REDCAP under five different Linux file systems (Ext2, Ext3, XFS, JFS and ReiserFS), different workloads, and two cache sizes.

The experimental results have proved that REDCAP can greatly reduce the I/O time of the disk read requests, for many workloads and any file system, by converting thousands of small requests into disk–optimal large sequential requests. It also achieves similar results to those obtained by a vanilla Linux kernel for workloads where an improvement in the I/O time is hard to obtain.

The results have also shown that the improvements achieved by REDCAP depend, to some extent, on the file system used. The best results are achieved for those file systems which divide the disk into several groups, such as Ext2, Ext3, XFS and JFS, because the groups produce data locality which is exploited by the prefetching mechanism of REDCAP. For these file systems, even more than an 80% reduction is achieved.

The block allocation policy of the file system also affects the improvements achieved. This is the case with JFS, which creates single–extent files even when the files are created in parallel in the same directory. These single–extent files are ideal for the disk controller read–ahead and/or the Linux kernel block prefetching mechanisms in some workloads, where the benefit provided by REDCAP is unavoidably small.

We have also observed that ReiserFS, which does not split the disk into groups, produces requests which are apparently random for REDCAP. The large disk print caused by these requests makes many cache segments be evicted before being re-used, limiting the effectiveness of the REDCAP cache for a large number of processes. Despite these problems, an improvement of more than 55% is achieved.

Finally, the results obtained by the two REDCAP configurations are very similar. Since the number of processes is never greater than the number of REDCAP segments in our tests, these results suggest that the REDCAP cache size should also be dynamic and dependent on the workload.

As future work, we plan to study the possible enhancements of the activation–deactivation mechanism, and to allow REDCAP to reconfigure itself dynamically.

Acknowledgments

This work has been jointly supported by the Spanish MEC and European Comission FEDER funds under grants “Consolider Ingenio–2010 CSD2006–00046”, “TIN2006–15516–C04–03” and “TIN2007–60625”.

References

[1] R. Card, T. Ts’o, and S. Tweedie. Design and Implementation of the Second Extended Filesystem. *In Proc. of the First Dutch International Symposium on Linux*, 1994.

[2] P. González-Férez, J. Piernas, and T. Cortés. The RAM Enhanced Disk Cache Project (REDCAP). In *Proceedings of the 24th IEEE Conference on MSST*, 2007.

[3] W. W. Hsu and A. J. Smith. The performance impact of I/O optimizations and disk improvements. In *IBM Journal of Research and Development*, volume 48, pages 255–289. 2004.

[4] JFS for Linux. http://jfs.sourceforge.net/, 2008.

[5] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. In *Computer*, volume 27, pages 38–46. IEEE Computer Society Press, 1994.

[6] Linux XFS. http://oss.sgi.com/projects/xfs/, 2008.

[7] ReiserFS. http://www.namesys.com, 2008.

[8] S. Tweedie. Journaling the Linux ext2fs Filesystem. In *LinuxExpo’98*. 1998.