The RAM Enhanced Disk Cache Project (REDCAP)

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

This paper presents the RAM Enhanced Disk Cache Project, REDCAP, a new cache of disk blocks which reduces the read I/O time by using a small portion of the main memory. The essential ideas behind REDCAP are to enlarge the built–in cache of the disk drive, imitate its behavior, and take advantage of its read–ahead mechanism by prefetching disk blocks. REDCAP is intended to be I/O–time efficient, and implements an activation–deactivation algorithm which turns its cache on and off dynamically depending on the I/O time improvement achieved. REDCAP has been implemented in the Linux kernel. The experimental results show that our proposal reduces the application time by up to 80% for workloads which exhibit some spatial locality, while it has the same performance as a traditional system for those workloads which have a random access pattern, or perform large sequential reads.

1. Introduction

Nowadays, all disk drives have a built–in cache (called in the rest of this paper *disk cache*) that acts both as a speed–matching buffer and as a block cache. In all modern computers, this cache plays a crucial role in the I/O subsystem, because it reduces to a large extent the bottleneck that means the secondary storage of many systems.

The size is one of the most important aspect in the design of a disk cache. Although the manufactures tend to integrate larger caches, their sizes don't meet the system designers' recommendations (0.1%–0.3% of the disk capacity [6], or even up to 1% [4]). For instance, a current disk of 500 GB usually has 16 MB of cache, i.e., only 0.003% of its capacity. The main reasons of this small size are a tradeoff between cache size and cost, and space limitations.

Since it is expected that the larger the disk cache, the better the performance achieved, the potential benefits of using a small part of the main memory to enlarge the disk cache should be considered. The aim of this paper is to present the RAM Enhanced Disk Cache Project (REDCAP), which is a new cache of disk blocks. The fundamental ideas behind our proposal are to use the new cache as an extension, in main memory, of the small disk cache, to imitate its behavior, and to take advantage of its read–ahead mechanism by prefetching some disk blocks, with the purpose of improving the I/O time, specially the read I/O time. To extend the disk cache, a new level in the cache hierarchy, referred to as REDCAP cache, is introduced just between the page cache and the disk cache (see Figure 1). In order to emulate the behavior of the disk cache, and to make use of its prefetching, our technique also prefetches consecutive disk blocks that could be served to a new read I/O request later. The I/O performance will be improved whenever a read operation can be satisfied from the REDCAP cache, without sending it to disk.

One might also consider the enlargement of the *page cache* provided by the operating system as a way to obtain a similar behavior. However, REDCAP is based on a prefetching policy which is completely different from that of the page cache. REDCAP prefetches blocks which are adjacent on disk, while the page cache usually reads in advance data blocks of the same file, which can be fragmented on disk. As our results will prove, even with a small use of main memory, REDCAP is able to obtain a performance which is much better than that obtained by the usual policies of the page cache.

Furthermore, our technique implements an *activation–deactivation algorithm* which studies the improvement achieved by its cache all the time and turns it on and off depending on this improvement.

We have implemented REDCAP in the Linux Kernel 2.6.14. The resulting implementation has been evaluated by using several benchmarks, and its results has been compared with that obtained by an original system without modifications. For some benchmarks, the application time has been greatly reduced and improvements of up to 80% have been achieved. For those benchmarks with random reads, or large sequential read accesses, REDCAP has a similar behavior to a traditional system.

24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)
0-7695-3025-7/07 $25.00 © 2007 IEEE

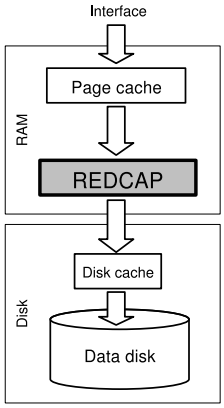

Figure 1. REDCAP cache hierarchy

The rest of the paper is organized as follows. Section 2 contains a brief description of previous work related to our proposed technique. REDCAP's design and implementation are discussed in section 3. A comparison of our results with respect to a traditional system is performed in section 4. Finally, some conclusions and an outlook for future work are provided in section 5.

2. Related Work

Since Maurice Wilkes proposed the cache concept [13], there has been an extensive and rich research in improving computing systems' performance by using several types of cache hierarchies. We will consider only work related to disk caches or techniques that take advantage of them.

Ruemmler and Wilkes [9] present a good summary of the disk characteristics and its operation. Shriver [10] gives an extensive description of disk caches, their parameters and their behavior. Karedla *et al.* [6] examine the use of caching as a means of decreasing the system response time and improving the data throughput of the disk system. Hsu and Smith [4] analyze the performance impact of various I/O optimization techniques and suggest that a reliable means of improving performance is to use larger caches.

Worthington *et al.* [15] propose two algorithms that use the contents of the disk cache, *Shortest Positioning (w/Cache) Time First (SPCTF)* and *Aged Shortest Positioning (w/Cache) Time First (ASPCTF)*. However, their results are not conclusive [14, 3], because they depend on both the workloads and the type of system model used.

Operating systems also incorporate some kind of caching and prefetching in their file systems. Linux implements a simple mechanism which prefetches file blocks when it detects a sequential access to a file. Other mechanisms are also possible, like the method which prefetches entire files by taking into account the file access patterns [7]. Note that our technique is independent of the prefetching of the operating system and/or application.

Finally, a disk cache that turns itself on and off dynamically depending on the performance achieved was suggested by Smith [11], but the idea was not tested, and no algorithm was developed to manage the cache state. REDCAP, however, implements a real mechanism to turn the cache on and off dynamically. Smith also suggests the possibility of using the main memory for the disk cache, *eliminating* the cache of the controller. Nevertheless, our cache does not replace the disk cache of the drive but instead *takes advantage* of it.

3. Design and Implementation

REDCAP consists of three parts: its cache, a prefetching technique to manage it, and an activation–deactivation algorithm to control the performance achieved.

The REDCAP cache is a cache of prefetched disk blocks. It is stored in RAM and has a fixed size of C blocks. The new cache, just like a disk cache, is split into N segments which are managed independently of each other and have a size of S blocks (where $C = N \times S$). A REDCAP segment is a group of consecutive disk blocks and is the transfer unit used by the prefetching technique.

When all the segments are in use, the REDCAP cache uses a Least Recently Used (LRU) replacement algorithm to determine which segment should be freed.

REDCAP sees the disk as a contiguous sequence of blocks, referenced by their logical block numbers, which is split into segments of the same size as the REDCAP segments. The first disk segment begins at disk block 0 and finishes at disk block $S-1$, the second one is from S to $2S-1$, and so on. Every REDCAP segment will have a disk segment, i.e., S consecutive disk blocks.

The prefetching technique implemented can be considered as a variant of the read–ahead of the disk cache. 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. The prefetching technique itself is quite simple yet effective, and it is applicable to any operating system, any file system or any storage device.

In a normal system, when an I/O request arrives to the block device layer, it is inserted into the request queue of the I/O scheduler. However, in a REDCAP system, it will be managed by REDCAP in the first place.

When a read I/O request arrives, we calculate the amount of affected disk segments and then look for them in the REDCAP cache. If the desired disk blocks are found in a REDCAP segment, a cache hit occurs, and they are serviced from the cache and no reads are performed.

However, a cache miss occurs if the data is not in the REDCAP cache. In this case, all the blocks of the corre-

0-7695-3025-7/07 $25.00

sponding disk segment will be read from disk. Note that some blocks will be those requested by the original read operation, while the others, read to complete the segment, will be the prefetched ones. Therefore, the amount of data to prefetch always depends on the size of both the request and the REDCAP segment. Our method exploits the principle of locality of reference: if a block is referenced, then nearby blocks will also soon be accessed.

It is interesting to note that when all the segments are busy attending to the ongoing requests, subsequent requests that cause a cache miss are sent to disk directly without cache intervention. In this way, REDCAP does not become a bottleneck when there are more requests than segments.

One important characteristic of REDCAP is the *activation–deactivation algorithm*. This algorithm watches the read disk requests, and estimates the time needed to process those requests with and without cache, activating and deactivating the REDCAP cache accordingly.

The REDCAP cache works in two states: *active* and *inactive*. In the active state, the REDCAP cache dispatches read requests. If the algorithm detects that the access time is getting worse with than without cache, it moves REDCAP to the inactive state. In the inactive state, the cache does not process requests and no prefetching is performed. However, the algorithm keeps studying the possible success of the cache. This study assumes that the cache is still working, and records the hits and misses on each read request. When the algorithm detects that the cache could be efficient, moves it to the active state again.

When the cache is active, REDCAP calculates, for each request, the "cache time" as the time needed to copy data from its cache to the original request plus the time needed to prefetch segments plus the time waiting on disk blocks to arrive from disk. REDCAP also estimates the time needed to read from disk the blocks served by the cache, if the cache would be inactive.

The algorithm says that if the time needed by the cache is less than or equal to the estimated time to read from disk the blocks served by the cache, the cache is been effective and does not have to be inactive.

If REDCAP is inactive, the algorithm uses the same condition to determine whether the cache has to be active again o not. But in that state, the cache time has to be estimated by using values stored during the active state, while the time of reading each request is now exactly calculated.

REDCAP does not take part in writes. It only updates its own cache by invalidating the appropriated disk blocks, and sends write requests to disk without any modifications.

4. Experimental Results

In order to study the behavior of the new cache, REDCAP has been implemented on the block device layer of the Linux kernel 2.6.14, under the page cache and just over the request queue of the I/O scheduler. We have run several benchmarks and compared the results obtained in our REDCAP Kernel (Linux Kernel 2.6.14 modified with our proposal) and in the vanilla Linux Kernel 2.6.14 (the Original Kernel without REDCAP).

Our experiments are conducted on an 800 MHZ Pentium–III system with 640 MB of main memory and two disks. The first one is the system disk, with the Fedora Core 4 operating system, and is used to collect the traces for a later study. The second one is the test disk and is a WD Caviar WD1200BB disk. The test disk has a capacity of 120 GB and a 2 MB built–in cache. It has one disk partition and the file system used is *Ext3* [12] with a logical block size of 4 KB.

In order to perform the study, the REDCAP cache size has been set up with 8 MB, which is four times as large as the cache of the test disk, although its memory utilization is less than 1.5% of the main memory. The tests have been carried out with four different configurations of the REDCAP Kernel: **256x32KB** (256 segments of 32 KB), **128x64KB**, **64x128KB** and **32x256KB**. The initial state of REDCAP is active.

In order to trace disk I/O activity, all the kernels record when a request starts and finishes, and when it arrives to the request queue. The REDCAP kernels also record information about the behavior of its cache, such as the hits and misses, and the time needed to copy the data on a cache hit.

Two I/O schedulers have been used in all the experiments: the Complete Fair Queuing (CFQ) scheduler [8] and the Anticipatory (AS) scheduler [5]. Since the results obtained for both schedulers are similar, we will focus on those obtained by CFQ. It is interesting to note that the CFQ scheduler is the default I/O scheduler in the latest "official" versions of the Linux Kernel, and even many distributions have been used it for a long time.

4.1. Results

We have performed five runs for every benchmark and system configuration. The average 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 application time improvement achieved by REDCAP with respect to the Original Kernel.

4.1.1. Linux Kernel Read This benchmark reads the sources of the Linux Kernel 2.6.17 by using the command:

```
find -type f -exec cat {} > /dev/null \;
```

In the test disk, there are 32 copies of the kernel files. This test is executed for 1, 2, 4, 8, 16, and 32 processes,

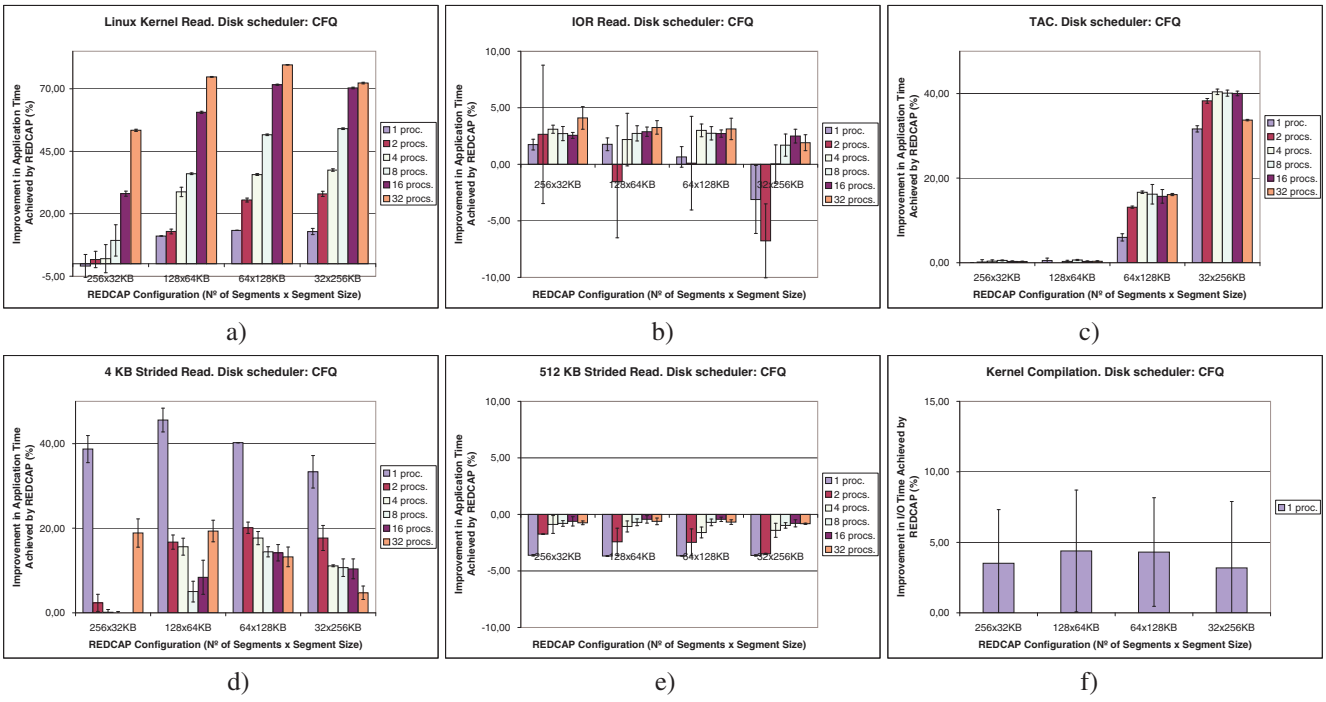

Figure 2. Improvements achieved by the REDCAP Kernel as compared to the Original Kernel.

each working on one of the copies of the Linux source tree. The corresponding results can be seen in Figure 2.a.

The REDCAP results are always better than the Original Kernel ones, and the improvement increases with the number of processes. The 64x128KB configuration shows the best performance for 1, 16 and 32 processes, whereas for 2, 4 and 8 process this is achieved by 32x256KB. Although the 256x32KB configuration presents the smallest improvements, it is still clearly better than the Original Kernel for 8 or more processes, reducing the application time by 54% for 32 processes. For 1, 2, and 4 processes, REDCAP and the Original Kernel statistically get the same results. The application time reduction achieved by REDCAP ranges from 1% (2 processes and the 256x32KB configuration) to 79% (32 processes and 64x128KB).

The REDCAP behavior strongly depends on the segment size (prefetching size), and the time reduction increases with it.

An explanation for these good results can be found in the way this test reads the large amount of small files of a Linux kernel source tree. The reading process is performed directory by directory. In an *Ext3* file system, the regular files of the same directory are stored together in disk in the group assigned to the directory (or in nearby groups if the corresponding group is full) [12]. The Original Kernel is not able to notice this pattern of close disk accesses nor does it perform a prefetching of files because they are small. However, REDCAP exploits the principle of locality of reference, so almost all the blocks prefetched are taken advantage of, and our cache is almost always active.

4.1.2. IOR Read The IOR benchmark (version 2.9.1) [1] have been used for testing the behavior of REDCAP in large sequential reads. We have used the POSIX API for I/O, and one file per process. The file size is 1 GB, and 64 KB is the size to be transferred in a single I/O call. This test is run for 1, 2, ..., and 32 tasks, each one reading its own file. Figure 2.b depicts the results.

The behavior of REDCAP is very similar to the Original Kernel one, and even small improvements are achieved by REDCAP. The exceptions are in the 32x256KB configuration with 1 and 2 processes, where the REDCAP performance is slightly worse than that of the Original Kernel. In these cases, the activation–deactivation algorithm turns the REDCAP cache on and off several times, when the best performance is achieved by an inactive cache. The problem is that sometimes, when REDCAP is inactive, it receives series of small requests (caused by meta–data reads) which turn the cache on before the algorithm realizes that the subsequent requests advice to keep the cache off.

This benchmark has a sequential access pattern, and the prefetching techniques of both the Original Kernel and the disk cache are optimized for this kind of pattern. Therefore, the contribution of our method is rather small, so the activation–deactivation algorithm detects this behavior, and the cache is turned off and is inactive almost all the time.

4.1.3. TAC This benchmark reads backward files with the command *tac*. The test is executed for 1, 2, ..., and 32 processes. Each process reads its own file. The files are the same as those used by the IOR–Read benchmark. The results are displayed in Figure 2.c.

The 32x256KB and 64x128KB configurations, which greatly improve the application time, show a qualitatively similar but quantitatively different behavior, which strongly depends on its configuration, i.e., the segment size.

The best performance is achieved by the 32x256KB configuration, with improvements of up to 40% for 4, 8 and 16 processes. The cache is always active.

Respecting to the 64x128KB configuration, REDCAP decreases the application time with respect to the Original Kernel in all the tests. The best performance is achieved with 4 and 16 processes, with a 16% of reduction. The cache is almost always active and is rarely turned off.

The reason why these benefits are achieved by these two configurations is that the Ext3 file system tries to allocate all the data blocks of a regular file together in disk, in such a way that the sequential access is optimized [12]. This block allocation benefits the REDCAP prefetching. However, the Original Kernel is not able to detect the backward access pattern, so it does not perform any prefetching.

On the other hand, for the 256x32KB and 128x64KB configurations, the results achieved by REDCAP are similar to those obtained by the Original Kernel. Since the *tac* command reads backward the file with requests of 64 KB, and because of the segment size of both configurations, the prefetching performed is very small, and does not contribute any benefit, so its cache is inactive all the time.

4.1.4. 4 KB Strided Read This benchmark reads a file with a strided access pattern (reads a first block of 4 KB at offset 0, skips a block of 4 KB, reads the next 4 KB block, skips another block, and so on). Again, the test is executed for 1, 2, ..., and 32 processes, and uses the same files of the benchmarks IOR–Read and TAC. This benchmark has been written in C, and uses the POSIX *read* and *lseek* functions. The results are shown in Figure 2.d.

The REDCAP behavior strongly depends on its configuration and on the number of processes. For the 128x64KB, 64x128KB and 32x256KB configurations, our proposal performs always better than the Original Kernel. The best results are achieved for 1 process with reductions of 45%, 40% and 33% for each configuration. Although the improvements of the 64x128KB and 32x256KB configurations decrease with the number of processes, the time for 32 processes is reduced by 13% and 4,7%, respectively. For the 128x64KB configuration, the worst performance is achieved for 8 processes, being still better than the Original Kernel with a 5% of reduction.

For the 256x32KB configuration and for 1, 2 and 32 processes our results improve the Original Kernel ones, whereas for 4, 8 and 16 processes they are similar to those obtained by the Original Kernel and no further improvements are achieved.

Since in this benchmark the algorithm is not able to decide the proper state, the REDCAP cache is active–inactive many times. The problem, which only appears in this case, can be easily explained. When the REDCAP cache is active, the disk drive detects a sequential access pattern, and activates its read–ahead mechanism. Then, the original requests take a small time so the algorithm decides that the cache cost is greater than directly reading the data from disk, and the cache becomes inactive. However, since in that state the requests are not sequential, the disk does not activate its read–ahead mechanism, and the original requests take more time. The algorithm decides that it is better to turn the cache on again. This state switch is successively repeated. Therefore, the results obtained do not show a systematic behavior as in previous cases. We are working on this issue in order to improve the algorithm and solve this problem in the near future.

It is also interesting to note that the Original Kernel does not detect this behavior nor does it implement any technique to enhance the performance under this type of access.

4.1.5. 512 KB Strided Read This test is similar to the previous one, but uses a different access pattern. In this case, every process reads 4 KB, skips 512 KB, reads 4 KB again, skips 512 KB, and so. When the end of the file is reached, a new read with the same access pattern starts again at a different offset. There are four read series. The first one begins at offset 0, the second at offset 4 KB, the third at 8 KB, and the fourth series at 12 KB. Once again, the test is executed for 1, 2, ..., and 32 processes by using the same files of the previous benchmarks. The results can be seen in Figure 2.e.

REDCAP has a quantitative similar behavior for all the configurations, but it does not perform better than the Original Kernel. The worst results are achieved for 1 and 2 processes, where the application time is rather small and although the REDCAP cache is all the time inactive, the time initially lost when the cache is still active can not be recovered later. For 4, 8, 16 and 32 processes the loss can be consider negligible, and it is mainly due to the time employed to simulate the behavior of the cache when it is inactive.

Again, the Original Kernel does not implement any prefetching technique for this access pattern.

4.1.6. Kernel Compilation for 4 processes This test compiles the vanilla Linux Kernel 2.6.17 with 4 processes (*make –j 4*) with the configuration of the 2.6.17–1.2142 kernel used by the Fedora Core 4 distribution. The "–j 4" op-

tion is used to saturate the CPU and send to disk as many requests as possible. The results can be seen in Figure 2.f, which shows the improvement with respect to the I/O time, since this benchmark is CPU–bound in our system.

The REDCAP results are always better than the Original Kernel ones. The best performance, 4,4% of improvement, is provided by the 128x64KB configuration, whereas the 32x256KB gives the smallest improvement, 3,5%. During the test, the cache is turned on and off several times. The number of requests served by the REDCAP cache when it is active ranges from 5% for the 256x32KB configuration to 43% for the 64x128KB and 32x256KB configurations.

Note that, during the kernel compilation, files from different directories are used at the same time, and not all the files of a directory are read consecutively, so the spatial locality, that exists in the *Linux Kernel Read* benchmark, is partly lost in this test. Hence, the results can not be compared to those obtained by the *Linux Kernel Read* test.

4.1.7. TPCC TPCC–UVa (version 1.2.3) is a free open–source implementation of the TPC–C Benchmark developed at the University of Valladolid (Spain) [2]. We have used 10 warehouses and 10 terminals. The benchmark is run with an initial 20 minutes warm–up stage and a subsequent measure time of 2 hours.

The behavior of REDCAP is very similar to the Original Kernel one and statistically gets the same results. TPCC has a random read pattern, which causes our cache not to be effective and it is inactive for long time in all the tests.

5. Conclusions and Future Work

In this paper we have introduced REDCAP, a RAM–based disk cache which is able to greatly reduce the I/O time of the disk read requests by using a small portion of the main memory. With segments as large as the maximum request size allowed by the operating systems (128 KB), REDCAP can improve the performance up to 80% for some workloads, while achieves similar results to that obtained by a vanilla Linux kernel for workloads where an improvement in the I/O is hard to obtain.

REDCAP has several features which make it unique. First, it is I/O–time efficient, since takes advantage of the disk read requests issued by the application in order to prefetch adjacent disk blocks. Second, it converts workloads with thousands of small requests into workloads with disk–optimal large sequential requests. Third, it implements an activation–deactivation algorithm which makes it dynamic. The algorithm is quite simple, although has proved to be very effective for a wide range of workloads. And fourth, REDCAP is independent of the underlying device. The activation–deactivation algorithm does not take into account any of the physical characteristics of the disk, it only uses the times obtained experimentally.

As future work, we plan to study the impact of the file system on REDCAP, the performance obtained by larger REDCAP caches and the possibility of reconfiguring REDCAP dynamically, changing both the segment size and the amount of segments.

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 “TIN2004–07739–C02–01”.

References

[1] IOR Benchmark, `http://www.llnl.gov/icc/lc/siop/downloads/download.html`.

[2] Tpcc-uva, `http://www.infor.uva.es/~diego/tpcc-uva.html`.

[3] G. R. Ganger. *System-Oriented Evaluation of I/O Subsystem Performance*. PhD thesis, 1995.

[4] W. W. Hsu and A. J. Smith. The performance impact of I/O optimizations and disk improvements. *IBM Journal of Research and Development*, 48(2):255–289, 2004.

[5] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Symposium on Operating Systems Principles*, pages 117–130, 2001.

[6] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[7] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *1997 USENIX Annual Tech. Conf.*

[8] R. Love. *Linux Kernel Development. Second edition.* Novell, 2005. ISBN 0–672–32720–1.

[9] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[10] E. Shriver. *Performance modeling for realistic storage devices*. PhD thesis, 1997.

[11] A. J. Smith. Disk cache–miss ratio analysis and design considerations. *ACM Trans. on Computer Systems*, 3:161–203, 1985.

[12] S. Tweedie. Journaling the Linux ext2fs filesystem. In *LinuxExpo’98*. 1998.

[13] M. V. Wilkes. Slave memories and dynamic storage allocation. *IEEE Tran. on Electronic Com.*, EC-14:270–271, 1965.

[14] B. L. Worthington. *Aggressive Centralized and Distributed Scheduling of Disk Requests*. PhD thesis, 1996.

[15] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. In *Proceedings of the 1994 ACM SIGMETRICS Conf.*, pages 241–251.