2010 22nd International Symposium on Computer Architecture and High Performance Computing

Simultaneous Evaluation of Multiple I/O Strategies

Pilar González-Férez*, Juan Piernas*, and Toni Cortes†
*Dept. de Ingeniería y Tecnología de Computadores
Universidad de Murcia
Murcia, Spain
Email: {pilar, piernas}@ditec.um.es
†Dept. de Arquitectura de Computadores
U. Politècnica de Catalunya and Barcelona Supercomputing C.
Barcelona, Spain
Email: toni@ac.upc.edu

***Abstract*—**
We present a framework for simulating the performance obtained by different I/O system mechanisms and algorithms at the same time, and for dynamically turning them on and off to improve the overall system performance. A key element of this framework is the the design and implementation of a *virtual disk* inside the Linux kernel. Our virtual disk creates a virtual block device which is able to simulate any hard drive with a negligible overhead, without interfering with regular I/O requests. We describe the potential of our proposal in REDCAP, a RAM-based disk cache which is dynamically activated/deactivated according to the throughput achieved. The results show that, by using our virtual disk, REDCAP obtains its maximum possible improvements: up to 80% for workloads with some spatial locality, and the same performance as a "normal system" for workloads with random or large sequential reads.

***Keywords*-Virtual disk; disk modeling; simultaneous evaluation; REDCAP.**

I. Introduction

Over the years, advances in disk technology have been very important, and vast improvements in disk drives have been made. But the disk I/O subsystem is still the major bottleneck for performance in many computer systems, because the mechanical operations considerably reduce its speed as compared to other components (CPU or RAM).

There are several mechanisms which play an important role in the performance achieved by the I/O subsystem: page and buffer caches of the operating system; built–in cache (*disk cache*) of the disk drives; prefetching of the operating system and of the disk drive itself; I/O schedulers; etc. Although these mechanisms can greatly reduce the I/O time, they are not optimal and its improvement depends on the workload. Moreover, all of them usually have a worst-case scenario which could downgrade the I/O performance.

So, it could be a good idea to activate/deactivate a mechanism, or to change from one to another, depending on the workload and expected performance. To achieve this dynamic behavior, we need a means to evaluate several I/O strategies at the same time.

As a first step to implement such a general system-wide simulation, we present the design and implementation of an in-kernel disk simulator which fulfills the above requirements. It is implemented inside the Linux kernel, creating a *virtual disk*. It has several important features: i) it is able to simulate *any disk* by using a table of I/O times addressable by seek distance, request size and operation type; ii) the system overhead produced by the simulator is negligible; iii) it does not interfere with regular I/O requests because *virtual* requests are processed out of the I/O path; and iv) by updating the table dynamically while the requests are served by the real disk, the disk behavior is modeled in a precise way.

As a use case, the virtual disk has been used for improving the effectiveness of REDCAP [1], [2], a RAM-based disk cache which uses a small portion of the main memory to enlarge the disk cache of the real disk, mitigating the problem of a premature eviction of blocks from the disk cache. REDCAP uses the disk simulator to evaluate the expected performance of its cache, turning it on and off accordingly. Its performance has been analyzed by using different workloads, and both a fresh and aged Ext3 file system. The results are consistent with those obtained in previous studies: REDCAP can greatly reduce the I/O time of the read requests (up to 80%) for workloads with some spatial locality, and that it roughly has the same performance as a traditional system for workloads with a random access pattern, or large sequential reads. But, unlike previous studies, this comparison also shows that, with the virtual disk, it can achieve its maximum possible improvements in all the workloads.

II. Related Work

There have been many proposals about disk simulators [3] and its possible applications, which have been used during many years to analyze the impact of disk trends [4], file system designs [5], buffer–cache replacement algorithms [6], and other architectural elements' designs and policies [7], on system's performance.

1550-6533/10 $26.00 © 2010 IEEE
DOI 10.1109/SBAC-PAD.2010.30

Simulators have usually been implemented in user space as standalone applications or integrated in a more general simulation environment, although, in some cases, they have also been implemented inside an operating system's kernel. Wang *et al.* [5], for example, implement a disk simulator inside the Solaris kernel, but it is specific to a disk model, and is not aimed at an on-line performance analysis of a system component, as ours is.

Table–models to simulate storage devices has also been explored previously [8], [9], [10], [11]. Popovici *et al.* [11] implement a table–based disk simulator of the underlying storage device inside the Linux kernel (Disk Mimic) which is quite similar to ours. As our virtual disk, it is portable across the full range of devices, uses automatic run–time simulation, and its computational overhead is small. However, there are also several important differences.

The first one is that their table–model only uses two input parameters: inter–request distance, and request type. However, we also take into account the request size. Our results show that the size is important for small inter–request distances, where both the disk cache, and the transfer time are the dominant factors in the I/O time, (see subsection V-B).

Memory overheads are also quite different. For instance, given a disk of 400 GB, the amount of memory required for their table is around 3 GB (there are more than 80 millions of possible of inter–request distances with interpolation). However, our tables only require 7 MB of memory (in our table each column represents a 1 GB of inter–request distance, except for the first columns).

Another difference is that Disk Mimic only captures the effects of simple prefetching, but our disk simulator implements a dynamic model which takes into account the effect of the disk cache on the current workload in a more accurate way. Moreover, thanks to the dynamic model, our virtual disk is also able to forget the past history, which depends on the past workload (in many cases, quite different to the current workload). By being dynamic and forgetting the past, our model performs a very quick adaptation to the real disk, when there is a change of the workload. But it seems that Popovici's on–line configuration takes a long time to equal the off–line configuration (which, in turn, needs two orders of magnitude more requests than our off–line training).

Finally, they only use Disk Mimic in a new disk scheduling algorithm to select the request with the shortest positioning time. But we can evaluate several I/O strategies in parallel, and change from one to another depending on the obtained performance.

The idea of issuing a "real" and a "virtual" request at the same time is similar to the I/O speculation proposed by Fraser and Chang [12], or Chang and Gibson [13]. But their goal is just to make prefetching more efficient.

III. In-Kernel Virtual Disk

A virtual disk inside the Linux kernel has been implemented to simulate the behavior of a real hard drive. The virtual disk is created by a driver which works much like a block device driver. Like a regular disk, it has its own I/O scheduler which sorts the incoming requests. These requests are a copy of those submitted to the real disk: before inserting a request in the scheduler queue of the real disk, a new "virtual" request is created with the same basic parameters of the real one. The virtual requests are then inserted in an auxiliary queue of the virtual disk to be processed.

The virtual disk driver creates a kernel thread that, after a creation and initialization phase, executes a routine which continuously performs the following actions:

1) Moves the requests from the auxiliary queues to the scheduler queue.
2) Fetches the next request from the scheduler queue.
3) Gets the estimated I/O time needed to attend the request from a table–based model of the real disk.
4) Sleeps the estimated time to simulate that the disk operation is being performed.
5) After waking up, completes the request, and deletes it from the scheduler queue.

A. Disk model

To model the storage device, a dynamic table is used. Given a request, the resulting table–based model receives a series of input parameters, and returns the I/O time needed to attend the request. As we will see, our disk model tables are trained by means of the requests issued to the real disk, without taking into account any disk–specific feature. Therefore, our method is able to model the behavior of any hard drive that could be used in practice.

Although the time of an I/O operation may depend on several factors [14], [15], our table–based model only uses the type (read or write), size, and inter–request distance [15] of a request to predict its I/O time. Section V-B shows that these three parameters, along with the dynamic behavior of the tables, are enough to accurately model the real disk.

Popovici *et al.* [11] claim that the logical distance between two requests and the request type are enough for predicting the positioning time. But we have also considered the request size for two reasons. First, because the transfer time is proportional to the request length [14], especially for small inter–request distances for which this time is the dominant factor in the I/O time. Second, because we take into account the disk cache, and the service time also depends on the request length in a cache hit. Subsection V-B shows that the request size is fundamental in our disk model.

Since read and write operations take different I/O times [14], [15], our model manages two tables: one for read requests and other for write requests. In our tables,

rows represent request sizes. Each table has thirty two rows, representing sizes from 1 (4 KB) to 32 blocks (128 KB), the minimum and maximum disk request sizes allowed by the operating system, respectively.

Columns represent inter–request distances. But, due to the huge number of possible inter–request distances in a modern disk, we have assigned ranges of inter–request distances per column. The first one represents a distance of 0 KB. Note that this column represents some disk cache hits. From the 2^{nd} to the 19^{th} columns, the table stores values for small inter–request distances to simulate, with a higher precision, the disk behavior, and the effect of its prefetching and cache. The $n-th$ column (n from 2 to 19) represents distances from $4 \cdot 2^{n-2}$KB to less than $4 \cdot 2^{n-1}$ KB. The largest inter–request distances are represented by the rest of the columns. The 20^{th} from 1 GB to less than 2 GB, the 21^{st} from 2 GB to less than 3 GB, and so on. Note that the disk capacity determines the number of columns.

To sum up, given a request, its type selects the table, its size the row, and its inter–request distance the column. The value of the cell is the I/O time needed to serve the request.

B. Training the table

The tables can be initialized on–line or off–line. For the off–line initialization, we have implemented a training program. This program has to be executed only once for every disk model, before actually using the real disk (due to the write operations). The training is usually "fast". In our system, it took 80 minutes for a 400 GB disk.

Although this program is executed in user space, tables are built inside the Linux kernel, which records the size, type, inter–request distance and disk I/O time of every request. The value of each cell is the average I/O time of the corresponding samples. Once obtained, the tables are copied from the kernel through the */proc* virtual file system, which is also used to provide the tables to the virtual disk after the off–line initialization.

The training program produces a random access pattern which does not appear in many workloads. So, to model the disk behavior in a more precise way, *to adapt the model to the current workload*, and *to catch the effects of the disk cache*, a dynamic method has been implemented. During regular system operation, cells are updated with the I/O times of the requests served by the real disk. Each cell stores the *last sixty four* measured I/O times (forgetting, in this way, past values which depends on past workloads), and its value is computed by averaging these times. Section V-B analyzes the sensitivity of the disk model to the number of averaged values per cell, and shows that, thanks to this dynamic approach, the virtual disk's behavior greatly matches the behavior of the real one.

With an on–line configuration, there is no training overhead; cells are just zeroed, and then dynamically updated as disk requests are served. For a not–yet–updated cell, the model will return the average of the corresponding column as I/O time, if this value is not zero; otherwise, it will return the average of the nearest column with non-zero cells.

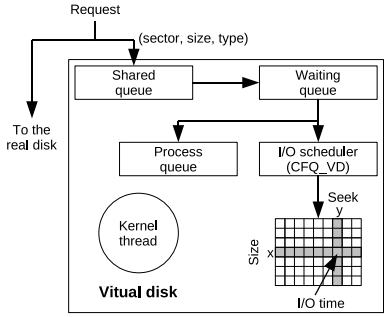

Figure 1. Overview of the virtual disk.

Note that our model does not explicitly consider several modern disk features, such as zoned recording, track/cylinder skew, and bad sector remapping. Its impact is indirectly taken into account through the I/O times obtained from the real disk during the dynamic update of the tables.

C. Request management

The virtual disk has to serve requests in the same order as the real disk had produced. Since our virtual disk may serve requests slower than the real one, dependencies between requests have to be controlled to allow the virtual disk to serve them in the "right" order.

A read request is usually synchronous: it blocks the corresponding application which can not issue new requests until the request is served. Read operations, hence, introduce dependencies among requests of the same process. Moreover, they also introduce dependencies among requests of related processes. A simple example is a parent process that executes the following piece of code:

1) Issue a synchronous read request (PRQ_1).
2) Create a child process and wait for it.
3) Issue a second synchronous read request (PRQ_2).

If the child process issues a synchronous read request (CRQ), the following dependencies will arise: (a) CRQ has to be issued once PRQ_1 has finished, and (b) PRQ_2 can not be issued until CRQ has not been completed.

However, there are also asynchronous reads. For instance, the Linux kernel supports file prefetching, and transforms (small) sequential read requests into large asynchronous readahead ones. So, we have to distinguish between *synchronous* and *readahead* operations.

Three queues, in addition to the scheduler one, are used to maintain those dependencies (see Figure 1). The **shared queue** communicates the disk simulator and the operating system. When a request is submitted, just before inserting it into the scheduler queue of the real disk, the system creates a *virtual request* which is inserted in the shared queue.

The **waiting queue** stores requests that can not be inserted into the scheduler queue, because they have dependencies to meet, and maintains their arrival order. Requests are moved from the shared to the waiting queue by the virtual disk after serving a request and before serving the next.

A request in the waiting queue is moved to the scheduler only if it does not have pending dependencies. The following heuristic implements the control of dependencies (although all the dependencies are not controlled, most of them are captured):

- Write requests are inserted immediately. They are usually asynchronous, and do not have dependencies.
- A *synchronous read* request will be inserted into the scheduler queue if there is not another *synchronous read* request of the same process, either ahead in the waiting queue, or in the scheduler queue. Note that a *synchronous read* request can be inserted in the scheduler queue even when there already exists a *readahead* request of the same process in this queue.
- A *readahead* request is inserted into the scheduler queue if there is no *synchronous read* request of the same process ahead in the waiting queue.
- The first request of a new process is inserted into the scheduler queue when the last request of its parent process, issued before child creation, has been served. On the other hand, if the parent process waits for the completion of the child, none of its new requests will be inserted into the scheduler queue until the child exits.

The last one, the **process queue**, just controls which processes have pending requests in the scheduler. The scheduler queue itself can not be used for this task because Linux manages it as a black box that can not be scanned.

Finally, although a virtual request has to be moved between different queues, its service time is computed in the same way as in a real disk: the elapsed time since it is inserted into the scheduler queue until its completion.

D. The I/O scheduler

We have adapted Complete Fair Queuing (CFQ) and Anticipatory (AS) schedulers to work with the request queue of the virtual disk. Moreover, since the virtual disk appears as a regular block device, it is even possible to change its scheduler on the fly, without rebooting the system.

Both schedulers use information about the process that issued a request to sort the queue. But, in the virtual disk, requests are submitted by the virtual disk itself and, hence, they belong to the kernel thread. So, the schedulers have been modified to store and use the process information in a different way. The new schedulers, *CFQ–VD* and *AS–VD*, have the same behavior as the corresponding original ones.

IV. The RAM Enhanced Disk Cache Project

The RAM Enhanced Disk Cache Project (REDCAP) [1] implements a new cache of disk blocks, between the page cache and the disk cache, which can greatly reduce the I/O time of the read requests. This cache works as an extension in RAM of the disk cache. A dynamic activation–deactivation algorithm controls the performance achieved by the REDCAP cache. The time that the cache needs to process the requests is compared with the estimated time to process them without cache, and, accordingly, the cache is turned on or off. When the cache is off, the algorithm keeps studying its possible success, and, when it detects that the cache could be efficient, it activates the cache again.

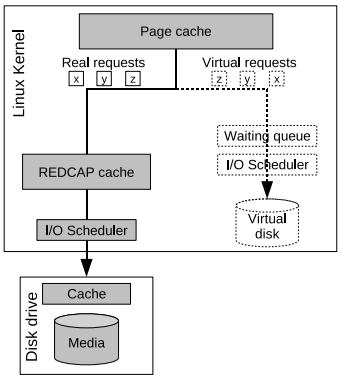

Figure 2. System with an active REDCAP

Originally, the algorithm used the I/O time of the read requests sent to disk to estimate other I/O times. Specifically, a "seconds per kilobyte" average was computed by using the I/O time and the size of the last 100 disk requests. This average actually provided a coarse model of the disk drive. It worked reasonably well, although had problems in some workloads, being unable to set the proper state of the cache [1], [2]. To solve these problems, the new implementation uses our disk simulator, which provides better estimated I/O times. When REDCAP is on, the virtual disk simulates the behavior of a normal system without REDCAP (Figure 2). When REDCAP is off, the virtual disk estimates the benefits that REDCAP could provide.

V. Experimental results

To analyze the performance of REDCAP with the in-kernel virtual disk, both have been implemented in a Linux kernel 2.6.23 (the *REDCAP kernel*). We have carried out several experiments to compare the REDCAP kernel with a vanilla Linux kernel 2.6.23 (the *original kernel*).

A. System under Test and Benchmarks

Our experiments are conducted on a 2.67 GHz Intel dual–core Xeon system with 1 GB of RAM and three disks. One is the system disk, with a Fedora 8 operating system, used for collecting traces. The other two are the test drives. One is a 400GB Seagate ST3400620AS disk with a 16 MB built–in cache. It has an *Ext3* file system, containing nothing but the files used for the tests. The other is a

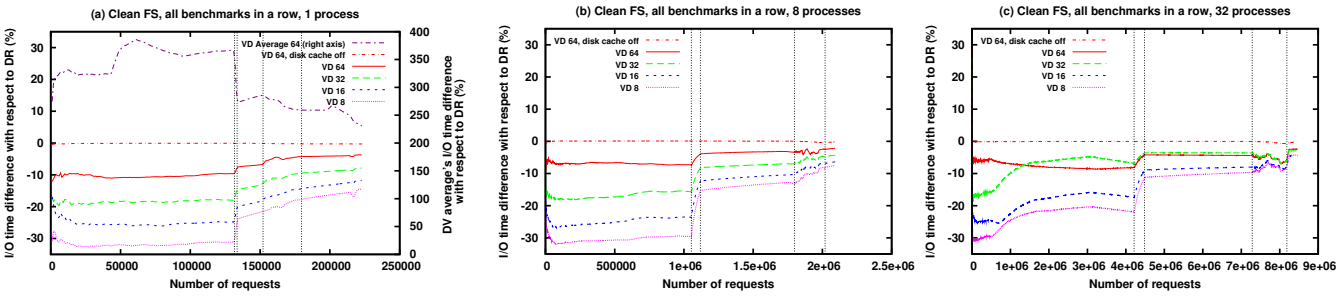

Figure 3. Virtual disk's percentage of I/O time difference with respect to the real disk in the *All the benchmarks in a row* test, when the clean file system is used for 1 (a), 8 (b) and 32 (c) processes. The vertical dashed lines mark the end of a benchmark and the beginning of the next one.

320 GB Samsung HD322HJ disk with a 16 MB built–in cache. It contains several aged Ext3 file systems in different partitions, obtained by copying sector by sector the disk of our department server. The file system containing the users' home directories has been selected to perform the tests; it is 270 GB in size and, at the time of the copy, was 84% full, and had been in use for several years. Files for carrying out the benchmarks have also been created in this file system.

The REDCAP cache has a size of 64 MB, with 512 segments of 128 KB each. In all the tests, the initial state of REDCAP is active.

Activity of the test disks has been traced by instrumenting the kernels to record when a request starts, finishes, and arrives at the queue. The REDCAP kernel also records information about the behavior of its cache.

Both CFQ and AS, and both CFQ-VD and AS-VD (see Subsection III-D), have been used in all the tests. Due to space constraints, however, AS results have been omitted, although they are very similar to those obtained by CFQ.

Our study uses the following benchmarks, each one executed for 1, 2, 4, 8, 16, and 32 processes:

Linux Kernel Read (LKR). This one consists of reading the sources of the Linux kernel 2.6.17 by using:

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

In the test disks, there are 32 copies of the kernel source, one for each process.

IOR Read (IOR–R). The IOR [16] test (version 2.9.1) is used for testing parallel sequential reads. This test has been configured to use the POSIX API for I/O, one file of 1GB per process, and a transfer unit of 64 KB.

TAC. Each process reads a file backward with the *tac* command. Files are the same as those of the IOR–R test.

8 KB Strided Read (8K–SR). Processes read files of 1 GB with a strided access pattern. Each process reads the first block of 4 KB of its file, skips two blocks (8 KB), reads the next 4 KB, skips another two blocks, and so on. Files are the same as those of the IOR–R and TAC tests.

512 KB Strided Read (512K–SR). Each process reads 4 KB, skips 512 KB, reads 4 KB, skips 512 KB, and so on. When the end of the file is reached, a new read with the same access pattern starts at a different offset. There are four read series at offsets 0, 4 KB, 8 KB, and 12 KB. Files used are the same as those of the IOR–R and TAC tests.

Directories Read (DR). This benchmark reads files in selected directories in the aged file system by using:

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

We have chosen 32 user's home directories, one per process, whose sizes range from 1 to 3 GB. This test is only executed in the aged file system, because the clean file system does not contain these directories.

All the benchmarks in a row. The previous benchmarks are run one after another, without restarting the computer until the last is done. Since some of them use the same files, the execution order tries to reduce the effect of the buffer cache. On the clean file system: *TAC*; *512K–SR*; *8K–SR*; *LKR*; and *IOR–R*. On the aged one: *TAC*; *DR*; *512K–SR*; *8K–SR*; *LKR*; and *IOR–R*. The goal is to show how the table–based model adapts to changes on the workload.

All the benchmarks at the same time. This test runs all the benchmarks in parallel, and finishes when each one has been run at least once (hence, if a benchmark ends when others are still in their first run, it is launched again). This test is only executed for 1, 2 and 4 processes, i.e., each benchmark is run for that number of processes. The aim is to analyze the behavior of our proposal when the workload is a blend of different access patterns.

B. Accuracy of the virtual disk model

In order to evaluate the accuracy of the disk model, the *All the benchmarks in a row* test has been run by making the real and virtual disks serve the same requests (however, requests can be served in different orders because each disk has its own I/O scheduler); I/O times achieved by both disks have been compared. This test has been selected because it shows how our virtual disk adapts to changes in the workload and, hence, indirectly, its accuracy in all the benchmarks. It also shows how the dynamic update of the tables allows the virtual disk to follow the real disk behavior.

Figure 3 presents the difference (in I/O time percentage) of the virtual disk with respect to real disk for 1, 8 and

32 processes, and the clean file system. It also shows the evaluation of different configurations of the virtual disk based on the number of values averaged per cell in the tables: eight ("VD 8" in Figure 3), sixteen ("VD 16"), thirty two ("VD 32"), and sixty four ("VD 64"). When each cell stores the average of the last sixty four values, our model has its best behavior by matching the real disk in a more accurate way.

The major differences are observed at the beginning of the test execution, when the **TAC** benchmark is run, due to the disk cache. Like a sequential one, a backward access pattern takes advantage of the prefetching performed by the disk cache. However, at a cache miss, the time needed to read the requested blocks is bigger than in the forward access due to the backward seeks. Although our read table is updated with all these times, the disk cache effect has a noticeable impact on the stored times, making the virtual disk faster than the real one. Despite this, the "VD 64" configuration of the virtual disk has the best behavior, greatly matching the real disk.

After the **TAC** execution, the difference between both disks decreases quickly. In fact, for the "VD 64" configuration, this difference is, on average, less than 5%. Hence, we can claim that *the virtual disk closely matches the real one*. The reason of this fast adaptation is that only a small percentage of table cells is used and updated, even with 32 processes. This can be observed in Figure 4, which shows the cells that have been modified once the execution has finished. The x–axis stands for the inter–request distance (table columns), and the request size (table rows) is represented in the y–axis.

One reason why the virtual disk behavior is not the same as the real disk's one is the difficulty to simulate the disk cache. To analyze this influence, the same test has been run with the disk cache off, and only for the "VD 64" configuration, line "VD 64, disk cache off" in Figure 3. Note that, without cache, the virtual disk matches the real one in a very accurate way, with a difference less than 0.2%.

Finally, to show that the request size is important in our model, we have run the same benchmark, but this time leaving out request sizes, and modeling the disk by using, for a given inter–request distance, the average of the values in its corresponding column. This study has been performed for only 1 process ("VD 64 average" in Figure 3.a), but it is enough to see that, if request sizes are not taken into account, the differences between our disk model and the real disk are very significant.

C. Benchmark Results

We have performed five runs for each benchmark and file system with both the REDCAP and the original kernel. Only the average results are presented, including their confidence intervals as error bars, for a 95% confidence level.

The computer is restarted after each run. Tables obtained from the off–line training are given to the virtual disk each time the system is initialized, and each cell averages the last sixty four values.

Benchmarks executed independently

We first analyze the results for the benchmarks run in an independent way. Figures 5.a and 5.b show the improvements achieved by REDCAP with respect to the original kernel for the clean and aged file systems, respectively.

TAC. With this test, REDCAP always performs better than the original kernel, obtaining improvements of up to 28.4%. The original kernel, unlike REDCAP, is unable to detect the backward access pattern, so it does not perform any prefetching. REDCAP cache is active most of the time,

DR. In this one, only run in the aged file system, our method reduces the application time (up to 9.7%) except for 2 and 4 processes where the confidence intervals say that the REDCAP and original kernels statistically have the same performance.

512K–SR. For this test, REDCAP provides no contribution because its cache is not effective, being almost impossible to profit the prefetching performed. The algorithm detects this fact and turns it off, which is inactive a long time. REDCAP behavior is quite similar to the original kernel's one, and, statistically, both have the same performance. The worst result is got for 1 process and the aged file system with a degradation of only 2.1%. This is due to the time initially lost while the cache is active at the beginning of the run.

8K–SR. In this case, REDCAP always performs better than the original kernel for both file systems. For this access pattern, REDCAP achieves the best results when its cache is always active, as it happens in this case. The operating system does not detect this access pattern nor does it implement any prefetching. But, with our technique, most of the requests take advantage of the prefetching performed by REDCAP, since almost nine out of every ten requests are cache hits. Reductions of up to 37% and 45% are achieved for the clean and the aged file systems, respectively.

LKR. Our method always performs better than the original one for this test. Our cache is always active for both file systems, what allows it to minimize the application time. REDCAP presents significant improvements, which increase as the number of processes grows. For 32 processes, the application time is reduced by up to 80% and 63% for the clean and aged file systems, respectively.

IOR–R. Since this test 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, the REDCAP contribution is rather small. Because, the I/O times of both disks are very similar, sometimes, our algorithm alternates the cache state between active and inactive. But the right decision would be to keep it inactive. For both

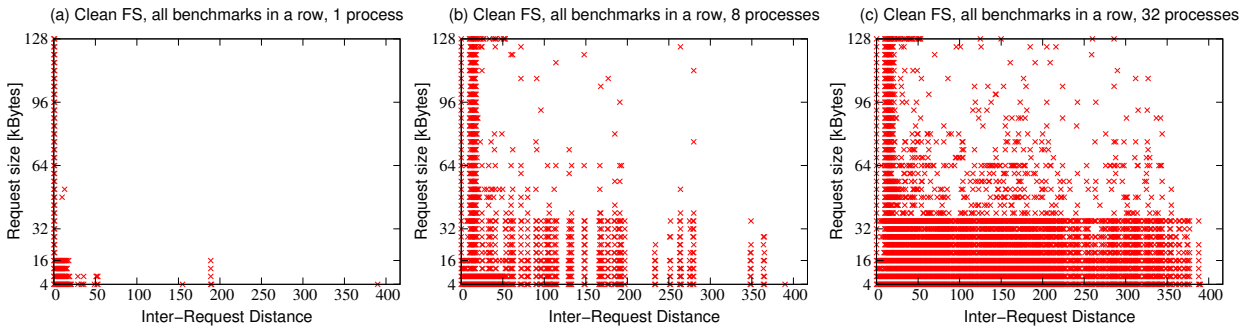

Figure 4. Modified cells in the read table once all the benchmarks in a row are executed for the clean file system, CFQ, and 1 (a), 8 (b) and 32 (c) processes.

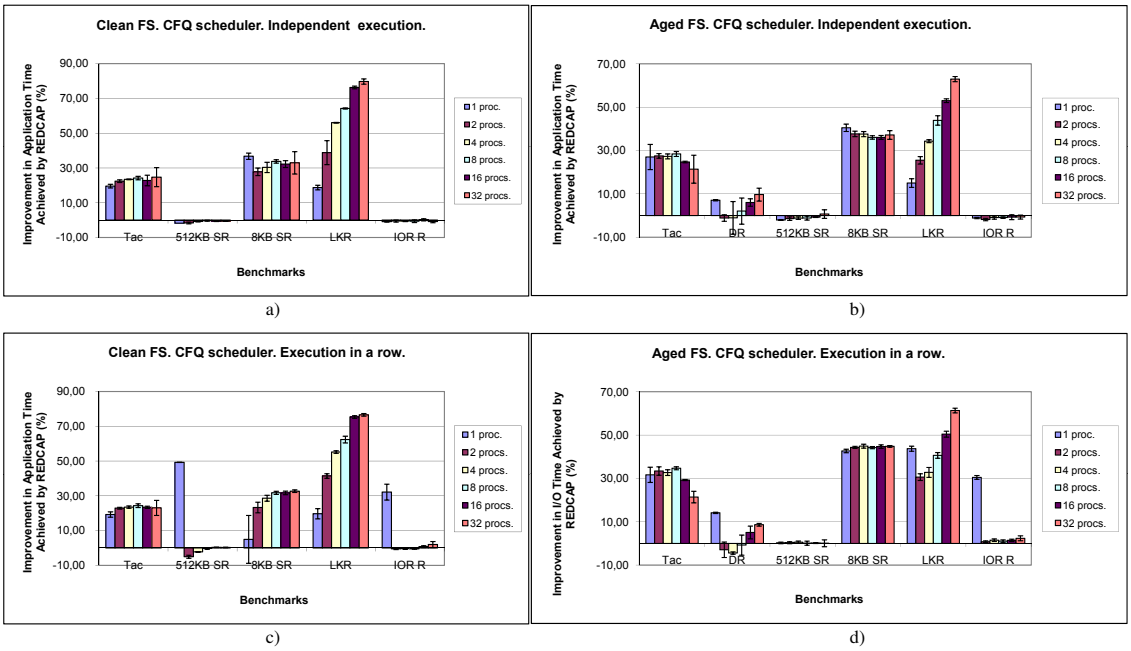

Figure 5. Independent execution of the benchmarks in a) and b), with the Clean and the Aged file system, respectively, and *all the benchmarks in a row* in c) and d), with the Clean and the Aged file system, respectively.

file systems, the behavior of the REDCAP kernel is very similar to that of the original one. Taking into account the confidence intervals, both present the same performance.

All the benchmarks in a row

Due to space constraints, the figures for the benchmarks executed in a row are not shown, and we only explain the more important aspects.

The results present a similar behavior to those obtained when the benchmarks are executed independently. Hence, the virtual disk adapts very quickly to the workload changes. Only minor differences are observed due to both the buffer and the REDCAP caches.

Clean file system and 1 process. When *TAC* has finished, a significant amount of file blocks are already in the buffer cache. So, *512K–SR* has to read only a small amount of data from the end of the file. After the first series, all the blocks requested by the other three series that are not in the buffer cache, are in our REDCAP cache. But the original kernel has to read all these blocks. For this reason, REDCAP unexpectedly gets an improvement of 50%.

When *8K–SR* is executed, our method reads more blocks than the original kernel, which reduces the improvement from 37% to 5%. This is due to the size of the kernel image, the REDCAP kernel image is larger than the original kernel one. So, after the execution of the two first test, with the REDCAP kernel, there are less file blocks in memory.

At the end of *8K–SR*, 4 out of every 12 file blocks are in RAM. And, when *LKR* ends, all these blocks are still in main memory. So, *IOR–R* produces a "strided" access pattern which prevents the original kernel from performing large prefetching requests. Whereas, the REDCAP prefetching is used widely, getting an improvement of 32%.

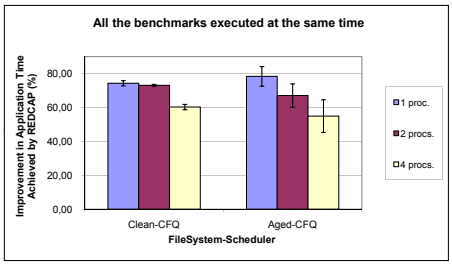

Figure 6. All the benchmarks executed at the same time.

Clean file system and 2 processes. At the end of the *TAC* execution, part of the files read by the two processes are in memory. But, due to the size of the kernel images, there are less file blocks in memory with REDCAP. So, when *512K–SR* is run, REDCAP reads more blocks than the original kernel. In this case, a degradation of up to 5% is produced.

Aged file system and 1 process. Now, there is an unexpected result only for *IOR–R* and 1 process. Again, the original kernel cannot perform large prefetching requests, whereas the REDCAP prefetching is completely exploited, achieving an application time reduction of 22%.

All the benchmarks at the same time

Results for the application time achieved by REDCAP, as compared to the original kernel's ones when all the benchmarks are executed in parallel, are presented in Figure 6. Our method always performs better than the original one, reducing the application time by up to 80%.

VI. Conclusions

This work presents the implementation of a virtual disk inside the Linux kernel which has several interesting properties: i) it creates a disk simulator which is able to simulate any disk by using a table of I/O times; ii) since it has the same interface as any other block device, it makes it possible to use any I/O scheduler with minimal modifications; iii) it does not interfere with regular I/O requests; and iv) it simulates the service order of the requests in a real disk by considering the possible dependencies between them.

Our virtual disk can be used for a simultaneous evaluation of different system mechanisms, and for dynamically turning them on and off depending on the performance obtained. Specifically, we have described how REDCAP can use the virtual disk to activate or deactivate its cache. Our results show that, by using our proposal, REDCAP usually obtains its maximum possible performance.

A line of future work is the application of the virtual disk to other I/O mechanisms, such as I/O scheduling.

Acknowledgments

Work supported by Spanish MEC and MICINN, and European Commission FEDER funds, under Grants TIN2009–14475–C04, TIN2007–60625, and CSD2006–00046.

References

[1] P. González-Férez, J. Piernas, and T. Cortés, "The RAM Enhanced Disk Cache Project (REDCAP)," in *Proc. of the IEEE Conference on MSST*, 2007.

[2] ——, "Evaluating the Effectiveness of REDCAP to Recover the Locality Missed by Today's Linux Systems," in *Proc. of the IEEE International Symposium on MASCOTS*, 2008.

[3] "The DiskSim Simulation Environment, http://www.pdl.cmu.edu/disksim/."

[4] C. L. Elford and D. A. Reed, "Technology trends and disk array performance," *Parallel and Distributed Computing*, vol. 46, no. 2, pp. 136–147, 1997.

[5] R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Virtual log based file systems for a programmable disk," in *Proc. of the USENIX Symposium on Operating systems design and implementation*, 1999.

[6] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: An effective buffer cache management scheme to exploit both temporal and spatial locality," in *Proc. of the USENIX Conference on File and Storage Technology, 2005*.

[7] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson, "Trading Capacity for Performance in Disk Array," in *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, 2000.

[8] C. C. Gotlieb and G. H. MacEwen, "Performance of movable-head disk storage devices," *J. ACM*, vol. 20, no. 4, pp. 604–623, 1973.

[9] N. C. Thornock, X. Tu, and J. K. Flanagan, "A stochastic disk i/o simulation technique," in *Proceedings of the 29th conference on Winter simulation*, 1997.

[10] E. Anderson, "Simple table-based modeling of storage devices," Tech. Rep., 2001.

[11] F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Robust, Portable I/O Scheduling with the Disk Mimic," in *Proc. of the USENIX Annual Technical Conference*, 2003.

[12] K. Fraser and F. Chang, "Operating System I/O Speculation: How two invocations are faster than one," in *Proc. of the USENIX Annual Technical Conference*, 2003.

[13] F. Chang and G. A. Gibson, "Automatic I/O hint generation through speculative execution," in *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[14] C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling," *Computer*, vol. 27, no. 3, pp. 17–28, 1994.

[15] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes, "On-Line Extraction of SCSI DISK Drive Parameters," Tech. Rep., 1997.

[16] "IOR Benchmark, http://ior-sio.sourceforge.net."