DADS: Dynamic and Automatic Disk Scheduling

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

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

Toni Cortes
Universitat Politècnica de Catalunya and Barcelona Supercomputing Center
Barcelona, Spain
toni@ac.upc.edu

ABSTRACT

The selection of the right I/O scheduler for a given workload can greatly improve the performance of a system. But, this is not an easy task because several factors should be considered, and even the scheduler deemed the "best" can change at any moment. So, we present a Dynamic and Automatic Disk Scheduling framework (*DADS*) that compares different Linux I/O schedulers and automatically and dynamically selects that which achieves the best performance for any workload. The implementation described here compares two schedulers by running two instances of a disk simulator inside the Linux kernel, each one having a different scheduler. Our proposal compares the schedulers' service times, and changes the scheduler in the real disk if the performance is expected to improve. *DADS* has been analyzed by using different workloads, hard disks, and schedulers. Results show that it selects the best scheduler of the two compared at each moment, improving the performance and exempting system administrators from selecting a suboptimal scheduler.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Storage Management—*Secondary storage*

General Terms

I/O scheduling, hard disk

Keywords

I/O disk scheduler, DADS

1. INTRODUCTION

Nowadays, disk performance is a dominant factor in a system's overall behavior. Indeed, because mechanical operations considerably reduce disk speed as compared to other components (e.g., CPU [3]), the I/O subsystem is still the major performance bottleneck in many computer systems.

Despite this fact, over the last decades, advances in disk technology have been very important, and many proposals have improved the overall I/O performance. Hence, several mechanisms play an important role in the I/O subsystem: operating system's page and buffer caches; disk drives' built–in caches (*disk caches*); prefetching; schedulers; etc.

Regarding I/O schedulers, many policies have been proposed to improve the I/O performance. Some of them try to minimize seek time, other proposals also take rotational delay into account, and even there are algorithms that assign deadlines to requests and try not to violate them. However, none of the scheduling algorithms is optimal in the sense that the improvement that they provide depends on several factors: workload characteristics, file systems, hard disks, tunable parameters, etc. They even usually have a worst–case scenario which could downgrade I/O performance.

In Linux 2.6.23 (used in our experiments), there exist four I/O schedulers: Anticipatory (AS), Complete Fair Queuing (CFQ), Deadline, and Noop. System administrators can select any of them, but choosing the one that always achieves the best performance is not easy. Most of the times, they do not make any selection, and the default scheduler (CFQ) is used, when a different one could improve the throughput.

Therefore, we present the design and implementation of a Dynamic and Automatic Disk Scheduling framework (*DADS*) that, by using an enhanced version of an existing in–kernel disk simulator [2], is able to automatically and dynamically select the best Linux I/O scheduler by comparing the expected performance achieved by each one. Our first implementation compares two schedulers, by running two instances of the disk simulator. One of the instances has the same scheduler as the target real hard disk, simulating its behavior. The other one has the scheduler with which the comparison is made, simulating the behavior of the real disk with a different scheduler. Disk instances calculate the service time of all the served requests. *DADS* then compares these service times, and decides a scheduler change in the real disk if its performance is expected to improve.

DADS has two important features: i) it can be used on any hard disk, since the new in–kernel disk simulator is able to simulate *any disk*, including its built-in cache and request arrival order; ii) it does not interfere with regular I/O requests because simulation and comparison are performed out of the I/O path.

DADS performance has been analyzed by using different workloads, four different hard disks, both fresh and aged Ext3 file systems, and the four Linux schedulers. The results show that it selects the best scheduler of the two compared at each moment, improving I/O performance.

To sum up, the main two contributions of this paper are: (a) the design and implementation of the Dynamic and Automatic Disk Scheduling framework, which is able to choose the I/O scheduler that achieves the greatest performance, exempting sysadmins from selecting a suboptimal scheduler; and (b) a new disk simulator which not only estimates disk I/O times but also simulates thinking times and disk caches.

2. RELATED WORK

The idea of an adaptable operating system or its components is not new. VINO [7] is an example of such operating systems. Denys *et al.* [1] also provide a good survey of this kind of systems. However, unlike *DADS*, VINO and other existing proposals have never been implemented.

Regarding scheduling, several proposals exist. ADIO is an Automatic and Dynamic I/O scheduler selection algorithm which selects among Deadline and CFQ based on expired deadlines [6]. It has two shortcomings: (a) deadlines can wrongly make ADIO select the scheduler which produces the worst performance, and (b) comparison always has to be done with Deadline. *DADS*, however, optimizes service time, and can compare any schedulers.

The Two–layer Learning Scheme [10] automates the scheduler selection by combining workload and request–level learning algorithms, and by using machine learning techniques. Two drawbacks of this proposals are that there must exist a learning phase, and the appearance of unexpected access patterns can produce the selection of the wrong scheduler. *DADS* does not have these problems because it dynamically decides the scheduler by means of the current requests.

3. DADS

DADS is a mechanism that compares any Linux I/O schedulers and automatically and dynamically selects that which, among them, obtains the best I/O performance. For simplicity, the version described here only compares two schedulers.

To carry out the analysis, we use a much–enhanced version of an existing disk simulator (see Section 4), run inside the Linux kernel. *DADS* uses two instances of the simulator (configured to mimic the target real disk) to evaluate each scheduler's performance. One instance, called *VD_RD* (Virtual Disk of the Real Disk), has the same I/O scheduler as the real disk, simulating its behavior. The other, called *VD_VD* (Virtual Disk of the Virtual Disk), has the scheduler with which the comparison is made, simulating the behavior of the real disk with a different scheduler (see Figure 1). By using two instances, and not the real disk and one instance of the disk simulator, comparison is fairer, and allows us to know that differences are due to the schedulers' performance, and not to error in the disk simulation itself.

Service time of a request is calculated as the elapsed time since it is inserted into a scheduler queue until its completion. A scheduler's performance is measured as the sum of the service times of all requests that it serves. Our proposal selects the scheduler that optimizes this total service time.

4. IN-KERNEL VIRTUAL DISK

The new in–kernel virtual disk simulator, used for implementing *DADS*, consists of two subsystems: a *virtual disk* (*VD*), which works as a block device driver and hard disk

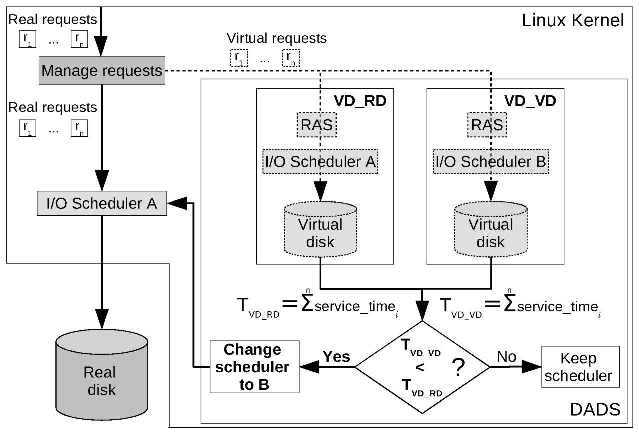


Figure 1: Overview of DADS.

drive, and a *request arrival simulator* (*RAS*), which simulates the request arrival to the I/O scheduler of the *VD*.

The virtual disk is implemented by using a kernel thread that continuously performs the following routine: fetch the next request from the scheduler queue; get, from a disk model, the estimated I/O time needed to attend the request; sleep the estimated time to simulate the disk operation; and after waking up, finish the request and inform to RAS.

RAS is also a kernel thread which inserts requests into the I/O scheduler queue by simulating the arrival of the request to the block device driver (see Section 5). It simulates *thinking times*: times elapsed between the completion of a request and the arrival of the next request issued by the same application. Since the virtual requests of a process have to be served in the same order as they were produced, *RAS* also controls dependencies between requests to allow VD to serve them in the "right" order. This control is done among requests of related processes too.

We have added a new table of read I/O times to the initial table–based disk model [2], and introduced a disk cache too.

Since read operations take different times depending on if they are cache hits or misses, now there are two read tables, making the disk model manage three tables in total. The three tables have the same structure, the only difference is the kind of values they store in each cell: I/O write times, cache–miss read times, and cache–hit read times.

Although the I/O time of a request may depend on several factors [4], to predict its disk time, our table–based model only uses its type, size, and inter–request distance [9], and, for read operations, information about its possible cache hit or miss. So, given a request, its type and the simulated cache determine the table to use, its size specifies the row, and its inter–request distance the column. The corresponding cell gives the I/O time for the request. As in the original model [2], cell values are dynamically updated through the I/O times provides by the real hard disk when serving requests issued by user applications (realize that the disk simulator itself never submits requests to the real disk). This allows DADS to adapt the cell values to the workload characteristics. Specifically, the value of each cell is the average of the last 64 corresponding samples.

Note that, if we used a single read table as in the original model [2], the estimated read times would not be as exact

Table 1: Parameters of the computers.

	Computer 1	Computer 2
CPU	Intel dual–core Xeon 2.67 GHz	Intel dual–core 1.86 GHz
RAM	1 GB	2 GB
1st Test Disk	Seagate ST3250310NS	Seagate ST3500630AS
Capacity	250 GB	500 GB
Cache	32 MB	16 MB
2nd Test Disk	Samsung HD322HJ	Seagate ST3500320NS
Capacity	320 GB	500 GB
Cache	16 MB	32 MB

as we need, since a cell's value would be the average of both cache hit and miss times, and the differences between them are usually very large. *DADS*, however, requires the disk cache, and the corresponding hit and miss times, to be precisely simulated, because the hit ratio produced by a scheduler greatly determines its performance. Also note that we need a single write table because a write–back policy and an immediate reporting are normally used in disk caches [8], and a request is considered "done" as soon as it is in cache.

There are many properties and features, most of them considered a trade secret, that specify a disk cache's behavior [4, 8]. Hence, it is not easy to simulate a disk cache, specially if it has a dynamic behavior. To reduce the number of possibilities, we model a cache that uses both read–ahead and immediate reporting, is divided into segments of equal size, and uses LRU as replacement algorithm. We have not considered a dynamic division of the cache. The number of segments and read–ahead sizes are calculated with a capturing program that uses the instructions given by Worthington *et al.* [9], and Schindler and Ganger [5], whereas the cache size is obtained from the manufacturer's specification.

We only perform read–aheads on cache misses, or partial misses. And, we have considered an adaptive read–ahead policy which uses two read–ahead sizes: one for sequential accesses (the maximum size), and one for random accesses.

Of course, our disk cache model does not fully simulate a disk cache and it is just an approximation. But, our intent is to develop one "alike enough" that allows us to study the performance of a system with different I/O schedulers. Our results show that our cache model meets this requirement.

The virtual disk has an I/O scheduler to manage its request queue. Moreover, since it appears as a regular block device, it is even possible to change its scheduler on the fly.

We have adapted CFQ and AS to work with the virtual disk. Both use information about the process that issued a request to sort the queue, but, in the virtual disk, all the requests are submitted by *RAS*. Consequently, CFQ and AS have been modified to use the process information in a different way, although the new *CFQ–VD* and *AS–VD* schedulers have the same behavior as the corresponding original ones.

Noop and Deadline, on the contrary, do not use any process information, and can be used without any modification.

5. DADS: IMPLEMENTATION

As aforementioned, *DADS* has been implemented to select between two Linux I/O schedulers by using two instances of the disk simulator and comparing their service times. Both instances serve the same requests, which are a copy of those issued to the real disk. "Virtual" requests are then inserted into the virtual disks for a later processing (see Figure 1).

Table 2: Parameters of the simulated disk caches.

			read–ahead size	
Disk Model	Size	# seg.	sequential	non sequential
ST3250310NS	32 MB	63	256 sectors	256 sectors
HD322HJ	16 MB	64	256 sectors	96 sectors
ST3500630AS	16 MB	20	256 sectors	32 sectors
ST3500320NS	32 MB	128	256 sectors	256 sectors

A first issue to address is that schedulers on the virtual disks (and, hence, the disk simulator instances themselves) tend to be alike. The problem is that the request arrival order in the *VD_VD* and *VD_RD* instances depends on the order in which the real disk's scheduler attends requests.

To avoid this mimicry problem, the simulation process is run in three phases. The first one collects requests from the real disk. During a time interval, the system copies to RAS requests issued to the real disk. For each virtual request, RAS calculates its dependencies and its thinking time. No request is queued into the scheduler; no simulation is done.

The second phase runs the simulation itself. RAS queues requests into its scheduler, and controls their dependencies and thinking times while the virtual disk serves requests simulating the real disk. For each served request, its service time is calculated. During this phase, the system does not copy any new requests. This phase finishes when both instances have served all the collected requests.

The last one compares the total service times achieved by the two disk simulator instances. If the performance achieved by *VD_VD* improves that obtained by *VD_DR*, it is expected that the performance of the real disk will also improve with a scheduler change. So, schedulers on the real disk and *VD_VD* are exchanged. Note that the change is also done in *VD_DR*. Once this phase is finished, the process starts over by collecting new requests.

As we are aware that a scheduler switch is time consuming (the active scheduler's queue has to be drained, and the new one initialized), a change is done iff the improvement achieved by *VD_VD* is greater than 5%. Moreover, an estimation of the time needed to do a change is also considered. So, if T_{VD_RD} and T_{VD_VD} are the service times of *VD_RD* and *VD_VD*, respectively, and T_{Change} is the time estimation to carry out a change, a scheduler switch is done when:

$$T_{VD\_VD} + T_{Change} < 0.95 \cdot T_{VD\_RD}. \qquad (1)$$

6. EXPERIMENTAL RESULTS

DADS and the in–kernel virtual disk simulator has been implemented in a Linux kernel 2.6.23 (the *DADS kernel*). We have carried out several experiments comparing two by two the Linux schedulers. The results have, in turn, been compared with those achieved by a vanilla Linux kernel 2.6.23 (the *original kernel*) with the same schedulers.

We have used two computers, with three disks each. One is the system disk, with a Fedora Core 8 operating system, used for collecting traces to evaluate the proposal. The other two are the test drives. Table 1 presents the main features of the computers and test disks.

All the test disks have an *Ext3* file system. Three have a clean file system, but Samsung HD322HJ has an aged file system, in use for several years. Files for carrying out tests have been created in all of them.

We run the following tests, each one executed for 1, 2, 4, 8, 16, and 32 processes:

Linux Kernel Read (LKR). Each process reads its own copy of sources of the Linux kernel 2.6.17 by using:
```
find -type f -exec cat {} > /dev/null \;
```
IOR Read (IOR–R). IOR (version 2.9.1) is used for testing parallel sequential reads. It has been configured to use the POSIX API for I/O, and a transfer unit of 64 kB.

TAC. Each process reads a file backward with `tac`.

8 kB Strided Read (8k–SR). 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.

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.

To show how *DADS* adapts to changes on the workload, switching the scheduler if necessary, the previous tests are run in a row, without restarting the computer until the last is done. The order is: *IOR–R*; *LKR*; *512k–SR*; *TAC*; and *8k–SR*. To reduce the effect of the buffer cache, and since all the tests but *LKR* use the same 32 files of 1 GB (one file per process), we have established that, until 16 processes, *IOR–R*, *512k–SR*, *TAC* and *8k–SR* use files which have not been used by the previous test. But, since there are only 32 files, for 32 processes it is not possible to meet this restriction. Due to space constraints, results obtained when the tests are run independently are not presented here, but they are similar to those achieved when the tests are run in a row.

Although we have compared all the I/O schedulers in Linux 2.6.23, due to space constraints, we only present the results of comparing CFQ vs Deadline. Nevertheless, the omitted results are very similar to those shown here. For the DADS kernel, two configurations have been tested: CFQ–Deadline and Deadline–CFQ. CFQ–Deadline, for example, means that, initially, the real disk uses CFQ, *VD_RD* uses CFQ–VD, and *VD_VD* Deadline. It also means that CFQ is the *default* scheduler, that is, the scheduler selected when a high rate of changes occurs and the change mechanism is deactivated for not hurting the system performance (the mechanism is reactivated later if the rate decreases). In the case of the original kernel, two configurations have also been tested: one using CFQ, and another one with Deadline.

Figures show how *DADS* adapts to the best scheduler, and the improvement achieved by each configuration over the worst one. Specifically, figures show:

$$\frac{T_{conf}}{Max(T_{CFQ-Deadline}, T_{Deadline-CFQ}, T_{CFQ}, T_{Deadline})} \quad (2)$$

where $T_{CFQ-Deadline}$, $T_{Deadline-CFQ}$, T_{CFQ} and $T_{Deadline}$ are the application times for the two DADS kernel and the two original kernel configurations, respectively, and T_{conf} is one of those four application times. Lines in figures represent the results of the original kernel, and histograms are the results of the DADS kernel.

The results are the average of five runs. The confidence intervals, for a 95% confidence level, have been calculated (but omitted for clarity), and are less than 5% of the mean.

Since the computer is restarted after each run, all tests have been done with a cold page cache. Initial disk simulator tables, obtained from an off–line training, are given to the virtual disks each time the system is initialized. The configuration of the simulated disk caches appears in Table 2. Every first phase of the simulation takes 5 seconds.

Figure 2 depicts the results for the four test disks. The first histogram shows the total application time, calculated as the sum of the application times of the five benchmarks, and summarizes our proposal's behavior. The other histograms show the application time of each individual test.

As we can see, for a given number of processes, *DADS* follows the best scheduler, changing the scheduler, if necessary, when moving from one test to the next one. Adaptation can easily be seen in figures 2.a and 2.d. Also note that *DADS* even outperforms the best scheduler in several cases.

IOR. For the sequential access pattern, *DADS* works as expected, and it adapts to the best scheduler. However, for *Deadline–CFQ* and any number of processes but 32, there is a small degradation with respect to the best scheduler because the worst one (Deadline) is initially used. A change is usually made in the first check, but the time initially lost during the first interval is not recoverable.

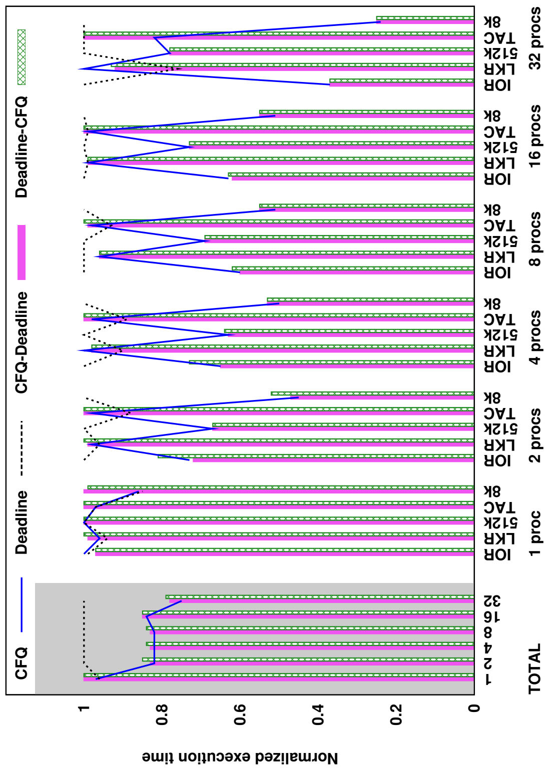
LKR. With this test, our mechanism adapts to the best scheduler, although there are a small degradation with respect to the best one in some cases. Since the scheduler that provides the best behavior with this benchmark is different from the one that presents the greatest performance with IOR (executed previously), LKR initially has the worst scheduler for its access pattern. Although a change is done in the first check, the increase in I/O time produced by the bad scheduler hurts the final result. This problem appears in all the test disks except the ST3200630AS model.

512k–SR. The *DADS kernel* presents the same behavior than the best scheduler, and only one case is remarkable. With the ST3500630AS disk and 32 processes, *DADS* is not able to select Deadline and spends all the time with CFQ. The problem is that, in the vanilla kernel, the application time difference between both schedulers is less than 5.7%, and *DADS* is not able to detect such a small difference.

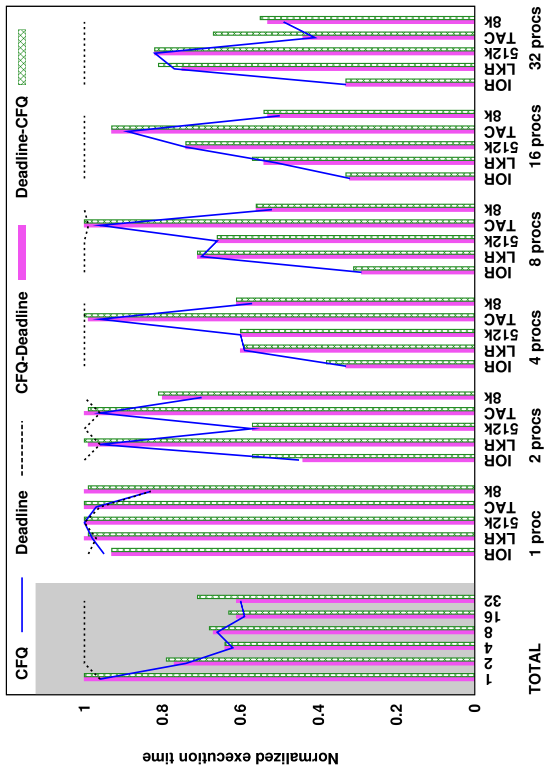
TAC. With a backward access pattern, the *DADS* behavior depends on the disk used. For ST3500630AS, the best scheduler is selected, achieving the same performance as the original kernel. But, for the other disks, it is not able to perform this selection, and increases the service time with respect to the best achievable result. For any number of processes except 32, the schedulers obtain almost the same performance, and our mechanism is not able to catch this difference. It introduces a small overhead which increases slightly the application time, although I/O time remains the same. For 32 processes, *DADS* frequently alternates from one scheduler to another, and the control mechanism to avoid such frequent changes is put into effect. Performance of each configuration is close to its default scheduler's one.

8k–SR. *DADS* always selects the best scheduler in this test. But, in some cases, it introduces a small overhead that slightly increases application time with respect to the original kernel and the best scheduler. The overhead is more noticeable for 1 process since application time is quite small.

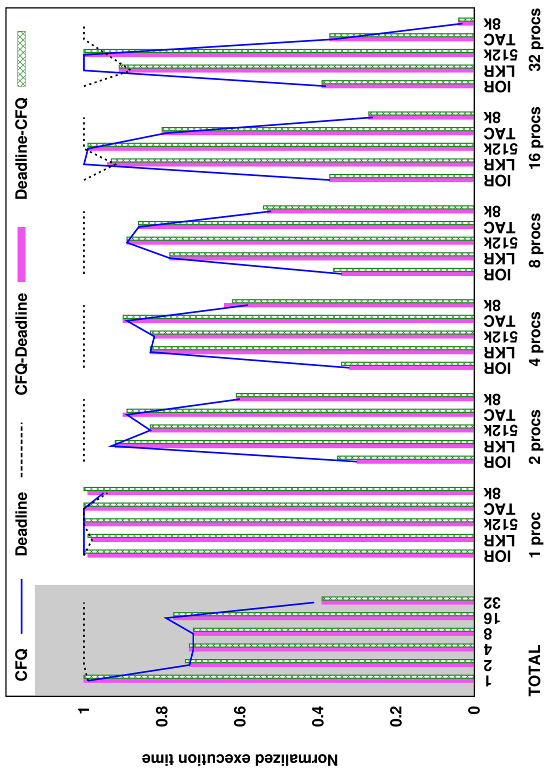
One unexpected result in this test is that the DADS kernel slightly increases I/O time (and, hence, application time) with respect to the original kernel when both use the same scheduler. The cause is the small overhead added when it copies a real request. This overhead delays the arrival of the request to the scheduler queue of the real disk. The delay is small, but big enough to make the disk wait for the time of almost a full rotation, because the requested sectors have



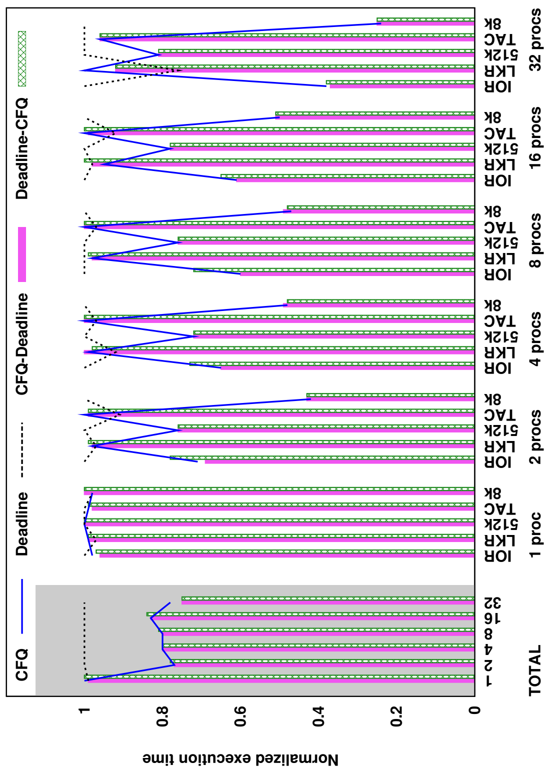
(a) Seagate ST3250310NS



(b) Samsung HD322HJ



(c) Seagate ST3500630AS



(d) Seagate ST3500320NS

Figure 2: Configurations CFQ–Deadline and Deadline–CFQ

just passed. Nevertheless, it is important to note that this problem only affect requests which are cache misses, and jump a small amount of sectors with respect to a previous one; it also depends on the disk model (different hard drive can have different sector layouts). So, the problem does not appear in the other tests and is almost negligible for the ST3500630AS and ST3500320NS drives.

7. CONCLUSIONS

DADS is a framework that automatically and dynamically selects the best of any Linux I/O schedulers for a given workload. The implementation presented here evaluates the expected performance of two I/O schedulers by running two instances of an in–kernel disk simulator. Simulations calculate the service times achieved by the schedulers for the served requests. *DADS* then compares these service times, and decides a scheduler change in the real disk if its performance is expected to improve.

Results show that *DADS* selects the best scheduler at each moment, improving the I/O performance. Hence, by using DADS, system administrators are exempted from selecting a suboptimal scheduler, which can provide a good performance for some workloads, but may downgrade the system throughput when workloads change.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

[1] G. Denys, F. Piessens, and F. Matthijs. A survey of customizability in operating systems research. *ACM Comput. Surv.*, 34:450–468, December 2002.

[2] P. González-Férez, J. Piernas, and T. Cortés. Simultaneous evaluation of multiple I/O strategies. In *Proceeding of the SBAC-PAD 2010.*

[3] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *USENIX Summer*. USENIX Association, 1990.

[4] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *Computer*, 27(3):17–28, 1994.

[5] J. Schindler and G. R. Ganger. Automated Disk Drive Characterization. Technical report, 12 1999.

[6] S. Seelam, J. S. Babu, and P. Teller. Automatic I/O Scheduler Selection for Latency and Bandwidth Optimization. In *Proc. of the Workshop on Operating System Interface on High Per. Applications*, 2005.

[7] M. Seltzer and C. Small. Self-Monitoring and Self-Adapting Operating Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems.*

[8] E. Shriver. *Performance modeling for realistic storage devices.* PhD thesis, 5 1997.

[9] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-Line Extraction of SCSI DISK Drive Parameters. Technical report, 1997.

[10] Y. Zhang and B. Bhargava. Self-learning disk scheduling. *IEEE Trans. on Knowl. and Data Eng.*, 21:50–65, January 2009.