J. Parallel Distrib. Comput. 74 (2014) 2380–2391

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A general framework for dynamic and automatic I/O scheduling in hard and solid-state drives☆

Pilar González-Férez [a,*], Juan Piernas [a], Toni Cortes [b]

[a] Universidad de Murcia, Spain

[b] Universitat Politècnica de Catalunya and Barcelona Supercomputing Center, Spain

HIGHLIGHTS

- DADS concurrently compares several I/O schedulers in real time.
- DADS automatically selects the I/O scheduler that provides the best performance.
- DADS works at any time for any workload without previous knowledge.
- DADS can be used in both hard disks and solid state drives.
- System administrators are exempted from selecting a suboptimal I/O scheduler.

ARTICLE INFO

Article history:
Received 3 December 2012
Received in revised form
14 January 2014
Accepted 2 February 2014
Available online 10 February 2014

Keywords:
I/O scheduler
DADS
Simultaneous evaluation
Virtual disk
Disk modeling

ABSTRACT

The selection of the right I/O scheduler for a given workload can significantly improve the I/O performance. However, this is not an easy task because several factors should be considered, and even the "best" scheduler can change over the time, specially if the workload's characteristics change too. To address this problem, we present a Dynamic and Automatic Disk Scheduling framework (*DADS*) that simultaneously compares two different Linux I/O schedulers, and dynamically selects that which achieves the best I/O performance for any workload at any time. The comparison is made by running two instances of a disk simulator inside the Linux kernel. Results show that, by using DADS, the performance achieved is always close to that obtained by the best scheduler. Thus, 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 the workloads change.

☆ A prior publication, related to this manuscript, is the following:

(i) P. González-Férez, J. Piernas, T. Cortes, DADS: Dynamic and Automatic Disk Scheduling, in: Proceeding of the 27th Symposium On Applied Computing, 2012.

Contributions of this manuscript beyond the above publication are the following:

1. All the results and comments for Solid State Drives (SSD) are new. New results appear in "Section 6.2. SSD drives".
2. "Section 6.1. Hard disk drives" includes new results for the comparison between Anticipatory (AS) and Deadline schedulers on hard disk drives. Previous work only provides results for the comparison between the Complete Fair Queuing (CFQ) and Deadline schedulers.
3. "Section 6.3. Webserver + Video" and "Section 6.4. Overhead of DADS and the disk simulator" are also new. The former presents results for more realistic workloads. The latter analyzes the overhead of our proposal.
4. Finally, in this manuscript, "Section 2. DADS overview", "Section 3. Modification of the in-kernel disk simulator", and "Section 4. DADS implementation" have been further elaborated.

All these new contents are more than 30% of the manuscript.

* Correspondence to: Facultad de Informatica, Campus Universitario de Espinardo, 30100 Murcia, Spain.
E-mail address: pilar@ditec.um.es (P. González-Férez).

http://dx.doi.org/10.1016/j.jpdc.2014.02.002
0743-7315/

1. Motivation

Since hard disk drives became the dominant secondary storage device in the 1960s, many scheduling policies have been proposed to improve I/O performance in different ways: bandwidth, quality of service, fairness, etc. There are policies that try to minimize seek times [27,6], take rotational delays into account [23,2,15], and even assign deadlines to requests [13]. However, none of the scheduling algorithms proposed so far is optimal: its results depend on several factors (workload, file system, device, etc.) that can change at any moment.

For instance, in Linux, while the Complete Fair Queuing (CFQ) scheduler, in general, provides good results for hard disks, there are workloads where Anticipatory (AS) gets a much better result. This behavior can be observed in Fig. 1, that shows application time improvements obtained by AS over CFQ when the Linux Kernel Read and IOR Read tests (described in Section 5) are run on a vanilla Linux kernel 2.6.23. The former test is a traversal of a directory tree

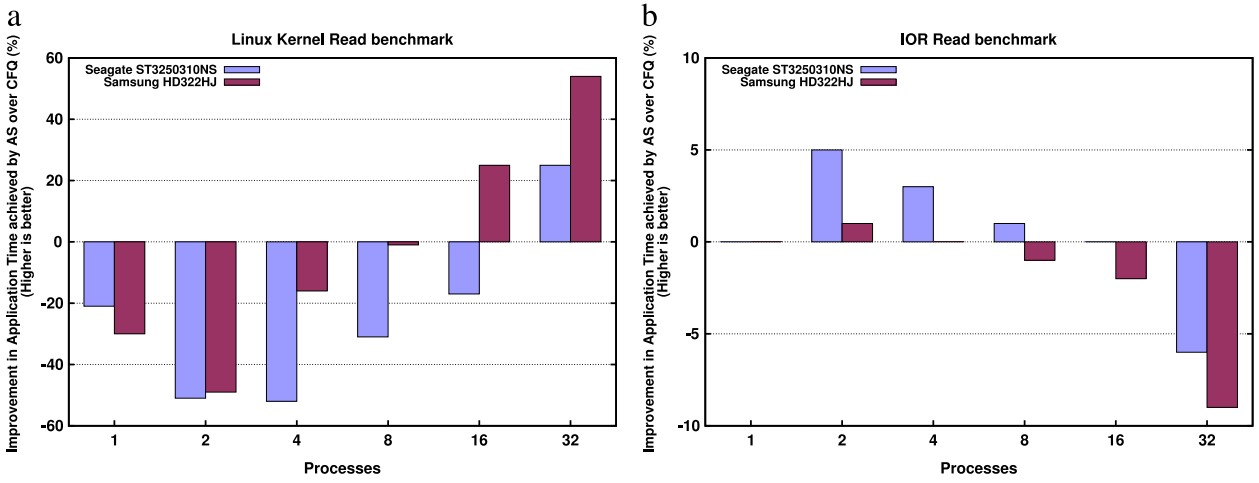


Fig. 1. Application time improvement, in percentage, achieved by AS over CFQ for the (a) *Linux Kernel Read* and (b) *IOR Read* benchmarks, when a Seagate ST3250310NS and a Samsung HD322HJ disk are used.

with small files, the latter is a sequential read of large files. For the Seagate disk and Linux Kernel Read test (Fig. 1(a)), CFQ achieves better performance for any number of processes but 32, and differences between them are significant. However, for the Samsung disk, CFQ only behaves clearly better than AS for 1, 2 and 4 processes. For the IOR Read test and the Seagate disk (Fig. 1(b)), CFQ only gets better performance for 32 processes, while it is better than AS for 8, 16 and 32 processes for the Samsung disk. Therefore, the best scheduler depends on the *workload*, *the number of processes in the workload*, and *even the drive*.

Another example is the Noop scheduler: its FIFO policy usually produces the worst performance on hard disks. But for flash memory cards or Solid State Drives (SSDs) [12,4], Noop usually gets better performance than other policies, because it does not add delays between requests that can increase the total service time.

Since there is no optimal I/O scheduler, some operating systems provide several. In Linux 2.6.23 (used in our experiments), there exist four: AS, CFQ (the default one), Deadline, and Noop. System administrators can select one of them for each disk. But choosing the scheduler that provides the best performance on each disk is not an easy task. Most of the times, administrators make no selection, and the default one is used. However, a different scheduler could improve the system's throughput. Therefore, a mechanism that automatically selects a scheduler, depending on the expected performance, could achieve the highest throughput.

Motivated by these ideas, we present the design and implementation of a general Dynamic and Automatic Disk Scheduling framework (*DADS*) for hard and solid-state drives that is able to select the scheduler that provides the highest I/O bandwidth at any moment, among those available in Linux.

To select the best scheduler, DADS uses an in-kernel disk simulator [7] that allows us to compare, in real time, different I/O mechanisms, and to dynamically turn them on and off, depending on the expected performance. We have adapted this simulator to DADS, and the enhanced version is also described in this paper.

DADS has been analyzed by using different workloads, six different disks (four hard disks and two SSDs), an Ext3 file system, and the four I/O schedulers available in Linux 2.6.23. Results show that, for workloads that do not change during the test execution, DADS usually achieves the same results as the best scheduler. However, when workload changes occur, it can even outperform the best scheduler in a vanilla Linux kernel. Therefore, DADS not only improves the I/O performance, but also, and more importantly, exempts system administrators from selecting a suboptimal scheduler that can hurt the performance. We have also analyzed the overhead of our proposal, and results show that it is almost irrelevant.

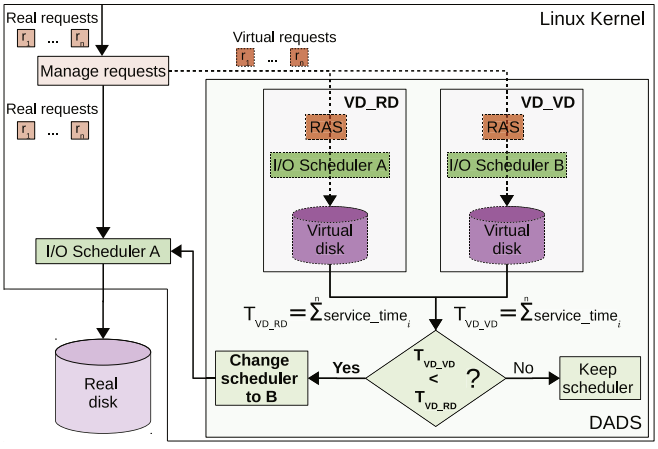


Fig. 2. Overview of DADS.

2. DADS overview

The availability of a disk simulator inside the kernel of an operating system enables the possibility of simulating several I/O strategies in parallel with the strategy enforced by the system [7]. One of the subsystem suitable for such simulation is the I/O scheduling: it is possible to compare several disk schedulers at the same time, and choose among them, according to the performance of each one.

For simplicity, the implementation of DADS presented in this paper only compares two I/O schedulers, and chooses the one that obtains the greatest performance. However, it can easily be improved to support three or more schedulers.

DADS evaluates I/O performance of each scheduler by using an enhanced version of our in-kernel disk simulator [7]. Two instances of the simulator are run inside the kernel (see Fig. 2). Both have the same configuration, except for the scheduler, and attend the same requests. We call them the *Virtual Disk of the Real Disk* (*VD_RD*), with the same scheduler as the target disk; and the *Virtual Disk of the Virtual Disk* (*VD_VD*), with the scheduler the comparison is made with.

By using two instances, and not the real disk and an instance of the simulator, the comparison is fairer, and allows us to know that the differences are due to the I/O performance of the schedulers and not to the simulation itself. Moreover, if our model has prediction errors, both instances will share the same errors.

To decide whether the real disk should change its scheduler or not, schedulers' I/O performances are calculated and compared. I/O performance can be defined by means of different metrics: average transfer rate of a disk, average response time per request, etc. We define it as the service time provided to applications. For each request, its service time is calculated as the elapsed time since the request is queued in the scheduler until its completion. A scheduler's performance is the sum of the service times of all the requests served. DADS selects the scheduler that minimizes this total service time. Since both instances transfer the same amount of bytes, minimizing service time implies maximizing I/O bandwidth. Our metric is effectively the same as the average response time per request.

3. The in-kernel virtual disk

The first version of our in-kernel disk simulator, implemented inside the Linux kernel, is able to simulate both hard disk and solid state drives [7]. Our simulator models a disk drive by using a dynamic table of I/O times, controls dependencies among requests to determine its arrival order, and has an I/O scheduler. However, it lacks two features that are important to fairly compare I/O schedulers.

First, it does not directly simulate the built-in cache of a disk (that we call *disk cache*). However, the cache-hit ratio of a scheduler significantly determine the disk performance. Second, it does not consider the *thinking time* of the requests (time elapsed between the completion of a request and the arrival of the next request of the same application) to control their inter-arrival times. Without this control, all the requests of a process arrives in a row and almost at the same time, so the process's "real" I/O behavior is not properly simulated. To take into account these aspects, our disk simulator now consists of two subsystems: a *virtual disk*; and a *request arrival simulator* (*RAS*).

The virtual disk mimics the behavior of a real disk (including its disk cache), and has associated its own I/O scheduler. It is a kernel thread that continuously runs the following routine: (1) fetch the next request from the scheduler queue; (2) get, from a disk model, the estimated I/O time needed to serve the request; (3) sleep this time to simulate the disk operation; (4) complete the request, and inform to RAS that the request has been finished. The disk model used in step 2 implements the simulated disk cache.

RAS (also a kernel thread) controls the arrival order of the requests, their thinking times and dependencies among them, to allow the virtual disk to serve the requests in the "right" order. The virtual requests of a process have to be served in the same order as they were produced. This order control is also done among requests of related processes (a parent and its children). Hence, RAS simulates the arrival of requests; it also inserts them into the scheduler of the virtual disk.

There are important differences between the first version of our simulator and that presented here. In the first version, RAS was not implemented, so the virtual disk itself inserted the requests into the scheduler, and controlled their dependencies. In the current one, RAS handles the arrival of requests and the virtual disk only simulates the disk drive. This simulation includes the disk cache, so the disk model is also different. Thanks to RAS, the simulation of the arrival of requests is more accurate (a request can be queued in the scheduler while a different one is being served), and the thinking times of the requests are taken into account too.

The following subsections discuss the features of our disk simulator, but focusing the attention only on the differences with respect to the first implementation.

3.1. Disk model

We model a disk drive by means of a dynamic table of I/O times addressable by seek distance (inter-request distance with a previous request), request size and type. The dynamic approach allows the disk model to adapt to workload changes.

Since read and write requests take different times [19,28,1,11], our initial table-based disk model used two tables, one for each operation type. However, since read operations also take different times on hard drives depending on whether they are cache hits or misses, now we use two read tables, one for each read type. The three tables of our model are: a cache-miss read table; a cache-hit read table; and a write table. The new disk model also needs a disk-cache model to determine whether a read operation is a cache hit or miss, and then decide the read table to use. This cache model and its management are described in Section 3.2.

If we used a single read table, the estimated read times would not be as exact as we need, because a cell's value would be the average of cache-hit and cache-miss times, and differences between them are usually very large. For instance, for the HDDs used in our experiments, a read request of 4 kB takes around 200 μs for a hit, and around 8000 μs for a miss. In our previous proposal, differences among hit and miss times were not important. But, for DADS, the disk cache and the corresponding hit and miss times have to be directly simulated, because each instance has a different scheduler, and the disk performance is significantly determined by the cache-hit ratio of the scheduler.

We assume that only one write table is needed because write requests are usually either sporadic or bursty. In the former case, due to the write-back policy and immediate reporting normally used in disk caches [26], writes are considered "done" as soon as they are in the cache, and I/O times are small. In the latter case, since the disk cache is saturated, a request usually has to wait for the previous one to reach the media surface. The net effect is like there does not exist disk cache, and I/O times are large. Consequently, cells will be updated with either short or large times, and the write table will adapt to both cases.

The three tables have the same structure: rows are request sizes, and columns are inter-request distances. There are thirty two rows, corresponding to sizes from 1 (4 kB) to 32 blocks (128 kB). Columns represent ranges of inter-request distances, due to the huge number of possible inter-request distances in a modern disk, and their number is determined by the disk size. The first one represents a distance of 0 kB, and it catches some cache hits. For columns from the 2nd to the 19th, the table stores values for small inter-request distances (from $4 \cdot 2^{n-2}$ kB to less than $4 \cdot 2^{n-1}$ kB; n from 2 to 19). Larger distances are represented by the other columns: the 20th from 1 to less than 2 GB, the 21st from 2 to less than 3 GB, etc.

We may think that, for a cache-hit read operation, the inter-request distance from a previous request could have no influence on its I/O time. But, for two of our test disks, we have realized that I/O times of cache-hit reads also depend on the number of cache segments used simultaneously (disk caches are split into segments) and, hence, on the distance between segments. For instance, for a 4 kB request that is a hit, I/O time is, on average, 177 μs if there is a single read stream, and only one cache segment is used. But, it is, on average, 323 μs if hits occur at different segments due to concurrent read streams. We think that this time increase is due to the selection of a different segment to serve the current request from the cache. Thus, the inter-request distance seems important for cache-hit reads too.

To predict I/O times, our new disk model uses four parameters: operation type, request size, inter-request distance from the previous request, and, for read operations, cache hit/miss information. Given a request, its type and cache information determine the table, its size specifies the row, and its inter-request distance the column. The corresponding cell gives the I/O time to attend the request.

As in the original model, tables are dynamically updated through the I/O times provided by the real disk, so cell values are

adapted to the workload characteristics. The value of each cell is the average of the last sixty four corresponding samples. The number of sixty four has been chosen after performing an analysis of the sensitivity of the disk model to the number of averaged values per cell [7].

In the case of the read tables, if a read request is a hit, its I/O time updates the cache-hit read table; if it is a miss, the cache-miss read table is updated. For hard drives, we have considered that a miss is a read operation that takes more than 1000 μs; otherwise, it is a hit. For SSDs, as explained in Section 3.2, we do not simulate a disk cache, and we only use and update the cache-miss read table.

Tables can be initialized on-line or off-line. For the off-line initialization, the cache-miss read and write tables are initialized by means of a training program [7]. There exists another training program for the cache-hit read table. This second program produces several access patterns that take advantage of the disk cache, like, for instance, a sequential pattern, or a "strided" pattern with small strides.

Both training programs have to be executed only once for every disk model. The total training process is usually "fast". In our system, it took 100 min to built the three tables for a 400 GB disk, and only 2 min for a 64 GB SSD disk. Note that the dynamic update of the tables will allow them to catch the potential increase in service time that SSD drives will suffer over time due to their use.

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

3.2. Simulation of the disk cache

There are many features, most of them considered a trade secret, that specify a disk cache's behavior [19,26,10]. Size is usually the only publicly available information. Consequently, it is not easy to simulate a disk cache.

To reduce the number of possibilities, we have only modeled a disk cache that uses both read-ahead and immediate reporting, it is split into segments of equal size, and uses LRU as replacement algorithm. Some of the segments are used for read operations and other for write operations. The number of segments does not change, since we have not considered a dynamic division of the cache either.

Our simulated disk cache only performs read-aheads on cache misses. We have also considered an adaptive read-ahead policy that uses two different read-ahead sizes: one for sequential accesses (the *maximum read-ahead size*), and another one for random accesses (the *minimum read-ahead size*).

Since modern drives hide their physical geometry to the operating system, the exact block layout is unknown, and it is impossible, for instance, to simulate that the read-ahead is made for track-aligned disk blocks. So, we have set that the first time a segment cache is used, the requested blocks, that produce the miss, are in the middle of the read-ahead blocks. Other subsequent read-ahead blocks will be behind or in front of the requested blocks, depending on the access direction.

Given a disk, the number of segments of its cache and the read-ahead sizes are calculated by using a capturing program that uses the instructions given in [28,21]. The cache size is obtained from the manufacturer's specification.

We are aware that our cache model does not fully simulate a disk cache, and it is just an approximation. Our intent is not to develop the best possible cache model, but to develop one "alike enough" that allows us to study system performance with different I/O schedulers. Since disk performance is greatly determined by the cache hit ratio of a scheduler, we will consider that the cache of our virtual disk is "alike enough" if it achieves a hit ratio similar to that obtained by the real disk. Experimental results will show that our cache model meets this requirement.

For SSDs, although some have a cache [17,20], to the best of our knowledge, it is used for internal operations, and not for read-aheads. We consider that SSDs do not have a disk cache, and only the cache-miss read table is used.

3.3. I/O schedulers for the virtual disk

Our virtual disk behaves quite similar to a "regular one". Consequently, it can use any of the four schedulers of Linux, and it can change its scheduler on the fly without rebooting, after pending requests have been completely drained. While Noop and Deadline can be used directly, AS and CFQ need a small modification.

The problem with AS and CFQ is that both take into account the process that issues each request to sort their queues. Since virtual requests are issued by RAS, all of them belong to the same process (actually, the same kernel thread in this case). Thus, we have changed the way both schedulers use for retrieving a process's information, so they can "see" several processes submitting I/O operations. The new CFQ-VD and AS-VD schedulers use the same code as their original counterparts, and behave like them, but work with our virtual disk [7]. Lu and Shen already pointed out the problem of losing the I/O context in AS. They analyzed remote accesses in parallel I/O systems, and showed that AS is only effective when it is aware of request-issuing processes' contexts [14].

3.4. Thinking time

Each application usually spends a thinking time before submitting its next I/O request. This thinking time is measured as the time elapsed since one request is completed until the next one is inserted into the scheduler queue.

To make a right simulation, thinking times of requests have to be considered because some Linux I/O schedulers, such as CFQ and AS, take these times into account during the scheduling. Moreover, if thinking time was not simulated, all the requests of a process would arrive at the same time to the scheduler of a virtual disk, without simulating the "right" arrival of the requests (although, due to the control of dependencies, they all would not be queued at the same time).

To compute the thinking time of a request, its arrival and completion times are recorded. The thinking time of a request is calculated by subtracting the completion time of the previous request of the same process from its arrival times.

Once the dependencies of a request have been solved, the thinking time is counted. Then, RAS sleeps this time to simulate the application's operation. Finally, the request is queued into the scheduler when its thinking time has elapsed.

4. DADS implementation

As we have said, DADS selects one between two I/O schedulers by using two instances of the disk simulator, *VD_RD* and *VD_VD*, each with a different scheduler. The selection is done by comparing the total service time of the simulations.

Both instances use the same disk model, i.e., same tables of I/O times and same configuration for the simulated disk cache. They also serve the same requests. Obviously, the orders in which requests are served are different since each I/O scheduler establishes its own dispatching order. Cache contents also differ.

4.1. Simulation phases

Since there are two simulators, we have to decide the request arrival order for each one. An approach is to use the same arrival

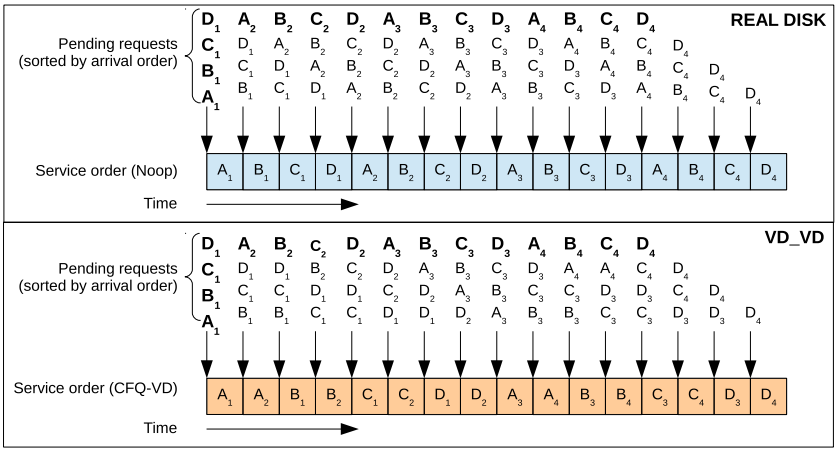


Fig. 3. Mimicry problem. The real disk has Noop, and *VD_VD* has CFQ-VD. There are 16 requests, 4 per process A, B, C and D. Bold and larger letters mark just arrived requests. The arrival order of these requests is the same in both cases.

order, captured from the real disk, for both virtual disks. But this approach will not work because each simulator has its own scheduler, and different schedulers lead to different application progresses: when a scheduler serves a request, only its corresponding application can progress and issue a new request, whereas other applications remain blocked waiting on their requests to be served. Since the scheduler of the real disk establishes the order in which processes issue new requests, *VD_VD* tends to behave quite similar to *VD_RD*, and a *mimicry problem* arises.

This issue is better understood with an example. Let us assume that the comparison is done between CFQ and Noop. The former gives to each process exclusive access to disk for a period of time by serving a few requests of the same process in a row. Let us establish that CFQ consecutively serves, at least, four requests of a process. Let us also assume that there are 4 processes (A, B, C and D) issuing requests to the real disk, that each one issues four synchronous requests (A_i, B_i, C_i and D_i respectively, with $i = 1, \ldots, 4$ indicating the number of request), that the time to serve a request is always the same, that all the processes start at the same time, and that they have quite similar thinking times.

Then, if the real disk and the virtual disk of the real disk (*VD_RD*) have Noop, and the other virtual disk (*VD_VD*) has CFQ-VD, the service order of the requests in the former will be: A_1, B_1, C_1, D_1, A_2, B_2, C_2, D_2, A_3, B_3, C_3, D_3, A_4, B_4, C_4, and D_4. While the service order in *VD_VD* will be: A_1, A_2, B_1, B_2, C_1, C_2, D_1, D_2, A_3, A_4, B_3, B_4, C_3, C_4, D_3, and D_4. Fig. 3 shows both service orders.

Due to the FIFO order imposed by Noop, in the better case, *VD_VD* consecutively serves only two requests of the same process, while, in a "normal situation", it could attend the four consecutively. When the real disk serves A_1, the request A_2 is issued, but there are already three requests (B_1, C_1 and D_1) waiting in the scheduler. So, the real disk, following the Noop policy, serves B_1 instead of A_2, and the A process cannot issue its third request. Fig. 3 depicts these problems.

Similarly, if the real disk and *VD_RD* have CFQ, and *VD_VD* has Noop, CFQ establishes the FIFO order to Noop. Consequently, *VD_VD* serves three requests of each process, when it should serve a request of each process according to Noop.

It is important to note that the mimicry problem does not depend either on the number of processes or the number of requests. We have chosen the number four (four processes issuing four requests each) to make easy and short the explanation.

To avoid this mimicry problem, the simulation process has *three phases*. The *initial phase* collects requests from the real disk, but no simulation is done. The system copies the submitted requests to RAS, and RAS creates the corresponding virtual requests, establishes their dependencies and calculates their thinking times. No request is inserted into the scheduler queue, and the virtual disk is just blocked. The duration of this phase can be configured using the */proc* virtual file system.

The *second phase* runs the simulation itself. RAS queues requests in the scheduler, by controlling their dependencies and thinking times. Both instances of the virtual disk serve requests and calculate the service time of each request. This phase finishes when the instances have served all the collected requests.

Finally, the *last phase* controls the performance and decides whether a scheduler change is needed. Then, the process starts over by collecting new requests. Note that the duration of the whole process depends on the first two phases.

4.2. Change of scheduler

DADS decides to change the scheduler of the real disk if the performance achieved by *VD_VD* improves the performance obtained by *VD_RD*, because it is expected an improvement in the performance of the real disk too. In other words, if T_{VD_RD} denotes the total service time of *VD_RD*, and T_{VD_VD} specifies the total service time of *VD_VD*, a scheduler switch is done if $T_{VD\_VD} < T_{VD\_RD}$ is true.

As we are aware that an I/O-scheduler switch is a time-consuming process, we take into account an estimation of the time needed to do the change. Moreover, since the simulation is very precise, but not exact, a scheduler change is only done if the improvement achieved by *VD_VD* is larger than 5%, thereby allowing a certain margin of error. Consequently, the previous equation is modified to

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

where T_{Change} is the time estimated to carry out the scheduler change, and is calculated by using the number of pending requests in the scheduler of the real disk and an average I/O time per request calculated in the previous scheduler change.

Since I/O schedulers may present a similar behavior for some access patterns, time differences can be slightly greater than 5% many times. This could produce a frequent scheduler change, and hurt performance. To avoid this, we have implemented a mechanism that, when 5 changes in a row are detected, permanently assigns the *default* scheduler to the real disk. We have considered that the default scheduler is set by the system administrator based on its experience, expected performance, etc., and not the default one imposed by the operating system. Both instances, however, keep comparing the service times, and the scheduler exchange is reactivated when no change has been predicted in the last 7 checks.

Table 1
Main hardware features of Hera and Hecate and their test disks.

Test disk	Model	Tech.	Capacity	Cache	Nick name
Hera: 2.67 GHz Intel dual-core Xeon and 1 GB of RAM					
1st	Seagate ST3250310NS	HDD	250 GB	32 MB	HD-250-32
2nd	Samsung HD322HJ	HDD	320 GB	16 MB	HD-320-16
3rd	Intel X-25M SSDSA2MH160G2C1	SSD	160 GB	–	SSD-160
4th	Intel X-25E SSDSA2SH064G1GC	SSD	64 GB	–	SSD-64
Hecate: 1.86 GHz Intel dual-core and 2 GB of RAM					
1st	Seagate ST3500630AS	HDD	500 GB	16 MB	HD-500-16
2nd	Seagate ST3500320NS	HDD	500 GB	32 MB	HD-500-32

Table 2
Parameters of the simulated disk caches.

Disk model	Cache size (MB)	# seg	Read-ahead size	
			Sequential	Non sequential
HD-250-32 (ST3250310NS)	32	63	256 sectors	256 sectors
HD-320-16 (HD322HJ)	16	64	256 sectors	96 sectors
HD-500-16 (ST3500630AS)	16	20	256 sectors	32 sectors
HD-500-32 (ST3500320NS)	32	128	256 sectors	256 sectors

5. Experiments and methodology

DADS and the in-kernel disk simulator have been implemented in a Linux kernel 2.6.23 (this is called the *DADS kernel*). We have carried out several experiments by comparing two by two the Linux I/O schedulers. Results are compared with those achieved by an unmodified vanilla Linux kernel 2.6.23 (the *original kernel*). We have also analyzed the overhead introduced by our proposal.

Two computers (called Hera and Hecate), with several disks each, have been used in our study. In each computer, one disk is the system disk that contains the operating system, and it is used for collecting traces to evaluate the proposal. The other disks are the test drives: four hard drives and two SSD drives. The main features of both computers and test disks are presented in Table 1. The "Nick name" column contains the names used for referring to the test disks during the explanation of the results. For hard disks, the nickname format is "HD-*capacity*-*cache*", where *capacity* is the disk capacity, and *cache* is the size of its cache. For instance, HD-250-32 represents the first test disk, a Seagate ST3250310NS drive of 250 GB in size and 32 MB of cache. For SSDs, the nickname is "SSD-*capacity*", where *capacity* is also the drive capacity.

All the test disks but HD-320-16 have a clean *Ext3* file system, containing nothing but the files used for the tests. The HD-320-16 disk 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, was in use for several years, and, at the time of the copy, was 84% full. Files for carrying out the benchmarks have been created also in this file system. The benchmarks use these new files and not the files already present on the aged file system.

Regarding the configuration of the simulated disk caches, we have executed our capturing program (see Section 3.2) to calculate the corresponding values. Table 2 summarizes these values for the four test hard disks. The size of the simulated disk cache has been set to the same size as the original one. Note that, for the SSDs, as previously explained, no disk cache has been simulated.

We have analyzed DADS performance by running a test that executes several microbenchmarks in a row, one after another, without restarting the computer until the last is done. In this way, we show how DADS switches the I/O scheduler and adapts itself to changes on the workload. The selected microbenchmarks are:

Linux Kernel Read (LKR). It reads the source of Linux kernel 2.6.17 by executing the command "`find -type f -exec cat {} >/dev/null \;`". In the test disks, there are up to 32 copies of the kernel files, one for each process.

IOR Read (IOR). The IOR test [8], version 2.9.1, is used for testing the behavior of DADS in parallel sequential reads. It has been configured with the POSIX API, one file of 1 GB per process, and a transfer unit of 64 kB. Before running the test, files were created in parallel with IOR too.

TAC. Each process reads a file backward with the command `tac`. Files are the same as those from IOR. For these files, `tac` uses a transfer unit of 8 kB.

8 kB *Strided Read (8k-SR)*. Each process reads 1 GB file through an access pattern with small strides. The test reads a first block (4 kB) at offset 0, skips two blocks (8 kB), reads the next 4 kB block, skips another two blocks, etc. Files are the same as those reads by IOR and TAC.

512 kB *Strided Read (512k-SR)*. This test is similar to the previous one, but has a larger stride. Now, every process reads 4 kB, skips 512 kB, reads 4 kB, skips 512 kB, etc. 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 at offsets 0, 4 kB, 8 kB, and 12 kB. The same files of the previous tests are used.

To establish the execution order of the tests, we have taken into account the performance of the I/O schedulers on each test. So, we have sorted them trying to cause a scheduler change. This order is: IOR; LKR; 512k-SR; TAC; and 8k-SR.

We have run the test for 1, 2, 4, 8, 16, and 32 processes. To reduce the effect of the buffer cache, since some benchmarks use the same files, we have set that, until 16 processes, each one, except LKR, uses files that have not been used by the previous. For 32 processes, as there are only 32 files, it is not possible to meet this restriction. But then, the dataset is large, and the buffer cache has a small impact.

By using Filebench [5], we have run two additional workloads: one that simulates a webserver (*Webserver*), and another one that simulates the same webserver plus a videoserver (*Webserver + Video*). Both workloads run for 10 min. In the latter, a videoserver workload is added every 3 min during 1 min, causing workload changes that allow us to show how DADS adapts to the changes.

We have compared two by two all the Linux I/O schedulers (AS, CFQ, Deadline and Noop), except for SSDs, where CFQ and AS present the worst behavior, and no comparison between them has been done. Due to space constraints, in this paper we only include the comparisons AS vs. Deadline, and CFQ vs. Deadline, for hard drives, and AS vs. Noop, and CFQ vs. Noop, for SSDs. Although not showed, results for the other comparisons are similar to those presented here.

6. Results

For all the experiments, the duration of the first phase, where requests are collected, has been set to 5 s. We have also performed several tests using an interval of 10 s, and the sole difference is that scheduler changes take longer to occur. These results have not been included in the present paper.

Tests have been run with all the possible configurations of the DADS kernel, depending on the I/O schedulers to compare and the scheduler initially assigned to the real disk. Note that the scheduler initially used by the real disk is also the *default* scheduler, and is the one selected when a high rate of scheduler changes occur (see Section 4). For example, when comparing AS and Deadline, *AS–Deadline* means that, initially, the real disk has AS (the default scheduler), *VD_RD* has AS-VD, and *VD_VD* has Deadline, whereas *Deadline–AS* means that, at the beginning, the real disk and *VD_RD* use Deadline (now the default one), and *VD_VD* has AS-VD. All the tests have also been carried out with the original kernel and the corresponding I/O schedulers.

For each disk, we compare the performance of two schedulers A and B by means of the following four configurations: DADS kernel with A–B; DADS kernel with B–A; original kernel with scheduler A; and original kernel with B. Figures show the improvement achieved by each configuration over the worst one. That is, if T is the application time, figures show:

$$\frac{T_{conf}}{T_{worst} = Max(T_{A-B}, T_{B-A}, T_A, T_B)}, \tag{2}$$

where T_{A-B} and T_{B-A} are application times of DADS for A–B and B–A, respectively; T_A and T_B are application times of the original kernel for schedulers A and B, respectively; and T_{conf} is the application time of the configuration analyzed (either T_{A-B}, T_{B-A}, T_A or T_B). Hence, figures also show how DADS adapts to the best of the two schedulers. Moreover, to facilitate the comparison, lines are used for showing results of the original kernel, and histograms for results of DADS.

In the figures, besides results for individual benchmarks, the total application time of the test, calculated as the sum of the application times of the benchmarks, is also showed inside a gray rectangle. This data summarizes the DADS behavior during the whole execution, and shows how it can reduce the overall I/O time. Furthermore, results are grouped by the number of processes, what allows us to observe how the scheduler changes, if required, from one test to the following.

Results shown for every configuration are the average of five runs. Confidence intervals, for a 95% confidence level, have also been calculated, and are less than 5%. However, for the sake of clarity in the figures, they have been omitted.

All the tests have been done with a cold page cache. Tables obtained from the off-line training are given to the virtual disk each time the system is initialized.

Before explaining the results, remember that DADS can outperform the best scheduler for a given test on a vanilla Linux kernel if the workload changes during the test execution, since DADS will select the best scheduler at any moment. Otherwise, without workload changes, DADS will simply adapt to the best scheduler.

6.1. Hard disk drives

Fig. 4 shows results for the four hard disks, and schedulers AS and Deadline, while Fig. 5 depicts results for CFQ and Deadline. For a given number of processes, figures show how a change in the benchmark (due to the execution of them in a row) can produce a scheduler change depending on the performance of each scheduler. The next paragraphs describe some interesting details, firstly, for the global execution of the test, and then, for each benchmark independently.

6.1.1. Global execution

Results for the total application time, depicted inside gray rectangles, show the global behavior of DADS. As we can see, DADS follows the best scheduler, changing the scheduler, if necessary, when the number of processes also changes. Adaptation can easily be seen, specially for HD-250-32 and HD-500-32. Furthermore, our proposal even *outperforms* the scheduler that presents the best behavior for the original kernel in several cases. For instance, when AS and Deadline are compared (Fig. 4), DADS outperforms the best scheduler (usually AS), for 2, 4 and 8 processes when using the HD-250-32 and HD-500-32 disks, achieving improvements of up to 7.4% and 3.4%, respectively, over the performance of AS (see Fig. 4(a) and (d)). When CFQ and Deadline are compared, and disks HD-500-16 and HD-500-32 are used, our method improves the performance of CFQ by 3.5% for 32 processes, being CFQ the scheduler that achieves the largest throughput for both disks in a vanilla Linux kernel (see Fig. 5(c) and (d)).

6.1.2. IOR Read

For the sequential access pattern of IOR, DADS works as expected, and it adapts to the best scheduler. A detail is that, for *Deadline–AS* and *Deadline–CFQ* (Figs. 4 and 5, respectively), sometimes there is a small degradation with respect to the best scheduler, because the worst of the two compared is initially used (Deadline). DADS changes the scheduler at the first check, but the time lost in this first interval cannot be recovered later due to the short duration of the test.

In some cases, DADS's performance is slightly worse than the one of the best scheduler in the regular kernel. Even then, without being perfect, DADS is able to bring its performance near to the best one, quite far from the worst scheduler.

6.1.3. Linux Kernel Read

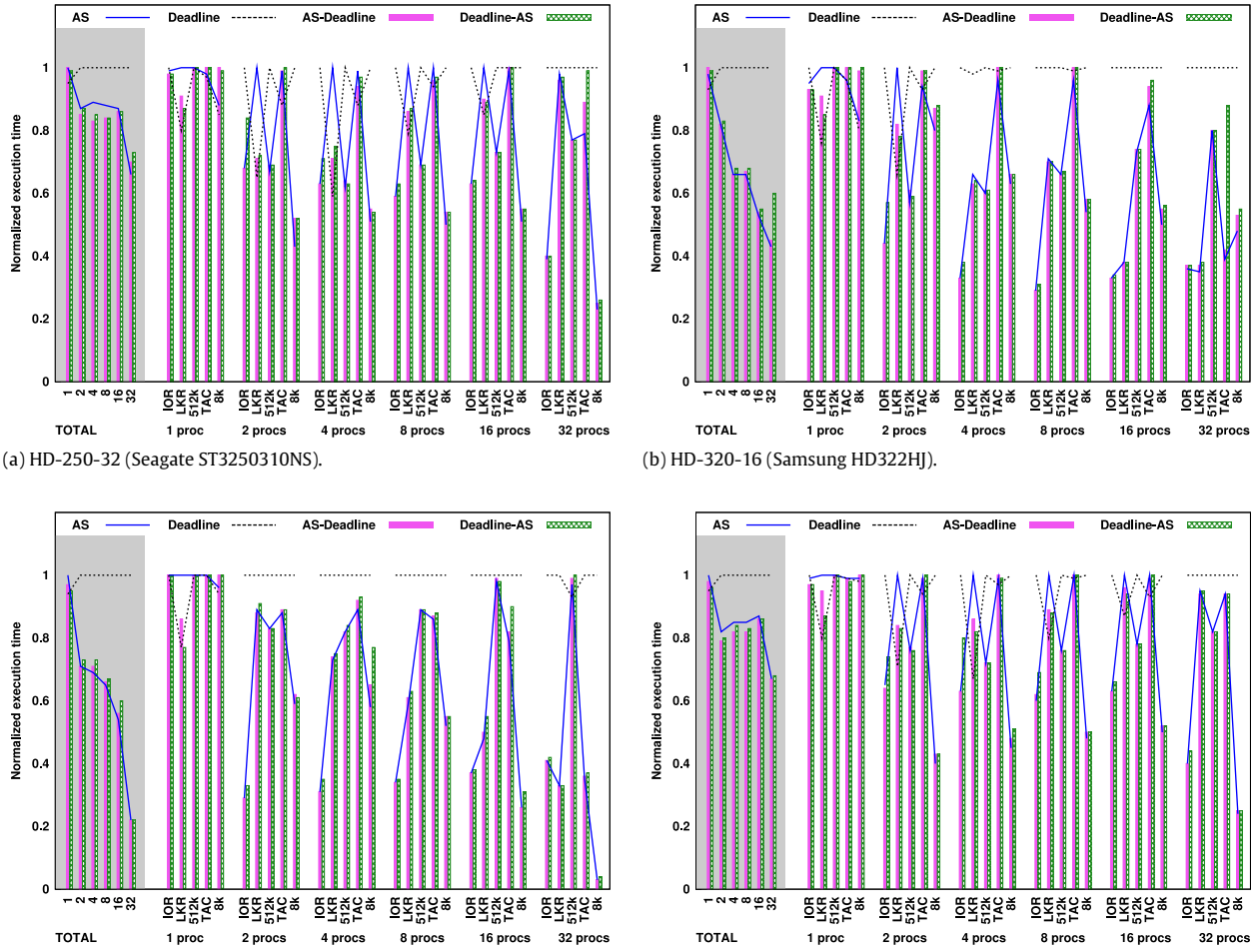
DADS adapts to the best scheduler, although, in some cases, there is a small degradation. The reason, as in IOR, is that the worst scheduler is initially used for this access pattern, because the scheduler that provides the best result for this test is different from the one that presents the highest performance for IOR (executed previously). Although the scheduler is changed at the first check, the I/O time increase, initially produced by the worst scheduler, hurts the final result. This behavior appears for all the disks except HD-500-16.

6.1.4. 512 kB Strided Read

The DADS kernel presents, with all the configurations, the same behavior as the best scheduler in the original kernel, and only one case is remarkable. For HD-500-16 and 32 processes, DADS does not select the best scheduler (Deadline), and spends all the time with AS or CFQ, respectively (see Figs. 4(c) and 5(c)). The mechanism to avoid frequent changes is even put into effect, because many and frequent changes are made. The problem is that, in the vanilla kernel, application time differences between the compared schedulers are quite small, less than 3.8% between AS and Deadline, and 5.7% between CFQ and Deadline. Differences in I/O time, also in the original kernel, are even smaller, less than 1.5% and 3.5%, respectively. DADS does not detect such small differences, and wrongly selects, most of the times, AS or CFQ, in each case. However, despite this, DADS's results are quite similar to those obtained by a vanilla kernel.

6.1.5. TAC

With the backward access pattern, DADS behavior depends on the disk. For HD-500-16, the best scheduler is selected and the same performance as the original kernel with that scheduler is



(a) HD-250-32 (Seagate ST3250310NS).

(b) HD-320-16 (Samsung HD322HJ).

(c) HD-500-16 (Seagate ST3500630AS).

(d) HD-500-32 (Seagate ST3500320NS).

Fig. 4. Configurations *AS–Deadline* and *Deadline–AS* for hard disk drives.

achieved. For the other three disks, DADS does not make the right selection, and slightly increases the service time with respect to the best achievable result. For any number of processes but 32, the problem is that the four schedulers obtain almost the same performance, and DADS is not able to catch such small difference. In these cases, our kernel introduces a small overhead that slightly increases the application time, although I/O time remains the same. This small overhead is due to DADS operation, and is noticeable in this test, and also in 8k-SR, because the number of requests served per second by both is much higher than in the other tests. For 32 processes, DADS frequently alternates from one scheduler to another, and the mechanism to avoid frequent changes is activated. Thus, performance of each configuration is close to its default scheduler's.

6.1.6. 8 kB Strided Read

Now, results depends on the configuration. For all of them, DADS always selects the scheduler that achieves the best performance. However, in some cases, it introduces a small overhead that slightly increases application times with respect to the original kernel. This degradation is more noticeable for 1 process because application times are quite small. In this case, the overhead is due to the DADS operation, but also to delays in the request arrivals (see below).

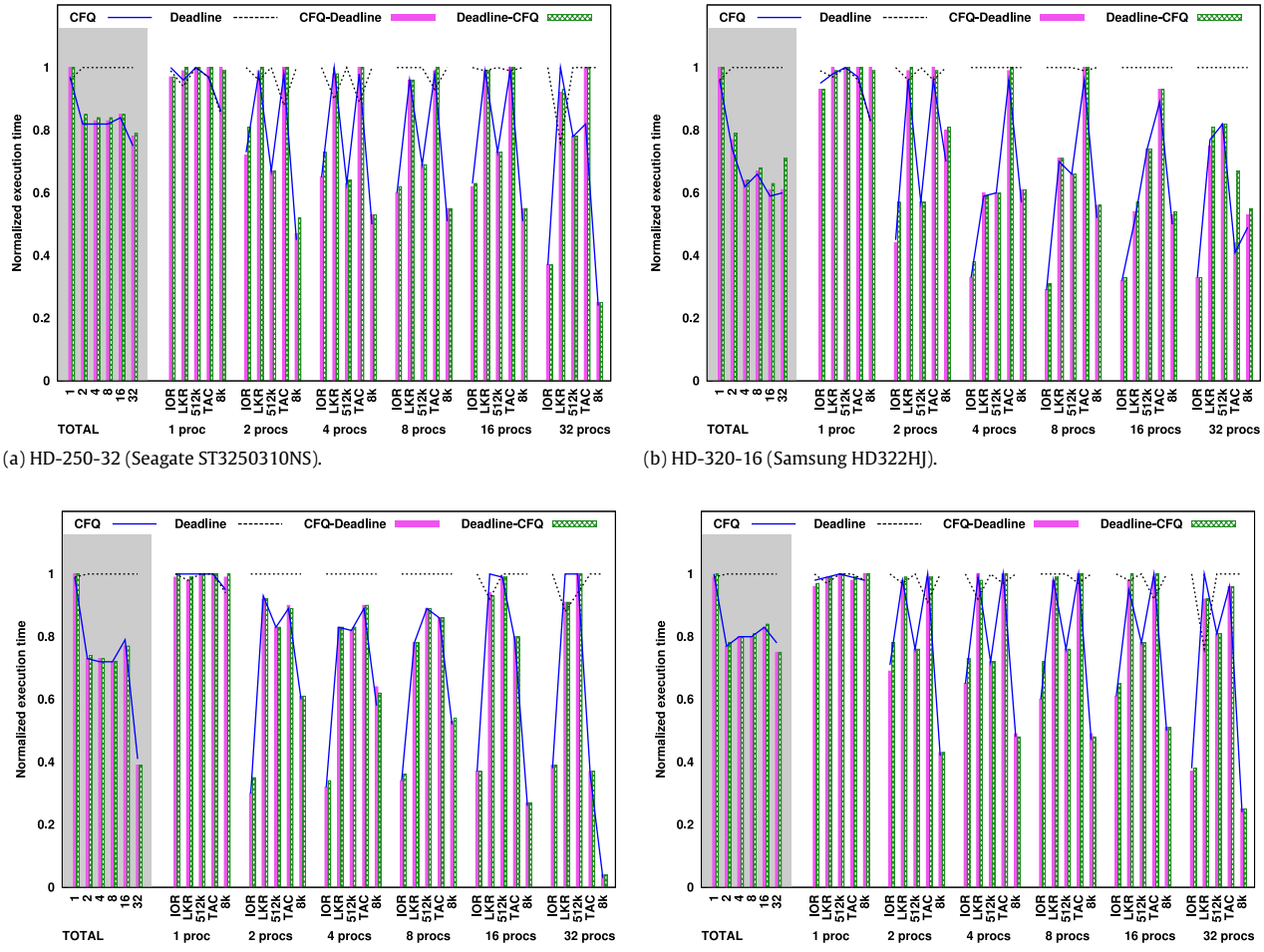
An unexpected result is that the DADS kernel slightly increases I/O times (and application times) with respect to the original kernel when both use the same scheduler. The cause is the small overhead that DADS adds when it copies a real request to the disk simulator. This small overhead delays the arrival of requests to the scheduler of the real disk. The delay is quite small, but big enough to make the disk spin almost a full rotation because the requested sectors have just passed.

Table 3 shows average I/O time per request for the DADS kernel and *CFQ–Deadline*, and for the original kernel with CFQ, for the four disks tested. For HD-320-16, differences are larger than for the other three disks, and the small degradation is more noticeable. Time differences are larger for 1 and 2 processes, because, to attend the requests, the disk has to make less seeks, which implies a small seek time. So, rotational delay, that is increased by the delay of requests, has a greater impact in I/O times. For simplicity, we have omitted these times for AS, albeit differences are quite similar to those provided for CFQ.

This problem only affects requests that 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 drives can have different sector layouts). The problem does not appear in the other tests and is almost negligible for HD-500-16 and HD-500-32. Moreover, we believe that in up-to-date systems, with powerful processors with several cores, this small delay can be reduced to almost zero.

6.2. SSD drives

Fig. 6 shows results for the SSD disks and AS vs. Noop, and CFQ vs. Noop. Before explaining the results, it is important to clarify two key aspects. Firstly, Noop and Deadline usually outperform CFQ and AS [12,4] for SSDs with respect to the application time,



(a) HD-250-32 (Seagate ST3250310NS).

(b) HD-320-16 (Samsung HD322HJ).

(c) HD-500-16 (Seagate ST3500630AS).

(d) HD-500-32 (Seagate ST3500320NS).

Fig. 5. Configurations *CFQ–Deadline* and *Deadline–CFQ* for hard disk drives.

Table 3
Average I/O time per request measured during the execution of the 8 kB *Strided Read* test, for the CFQ scheduler with the original kernel, and the configuration *CFQ–Deadline* of the DADS kernel, for 1, 2, 4, 8, 16, and 32 processes.

Disk	Kernel: scheduler	Processes					
		1 (μs)	2 (μs)	4 (μs)	8 (μs)	16 (μs)	32 (μs)
HD-250-32	Original: CFQ	147	200	203	190	182	187
	DADS: *CFQ–Deadline*	159	205	209	205	194	195
HD-320-16	Original: CFQ	221	234	232	233	237	247
	DADS: *CFQ–Deadline*	254	260	239	239	243	253
HD-500-16	Original: CFQ	155	168	170	174	180	187
	DADS: *CFQ–Deadline*	155	169	188	179	183	190
HD-500-32	Original: CFQ	159	200	200	184	183	187
	DADS: *CFQ–Deadline*	159	203	202	187	186	190

although all achieves almost identical I/O times. The problem is that CFQ and AS introduce delays with the hope of minimizing seek times, and these delays increase application times.

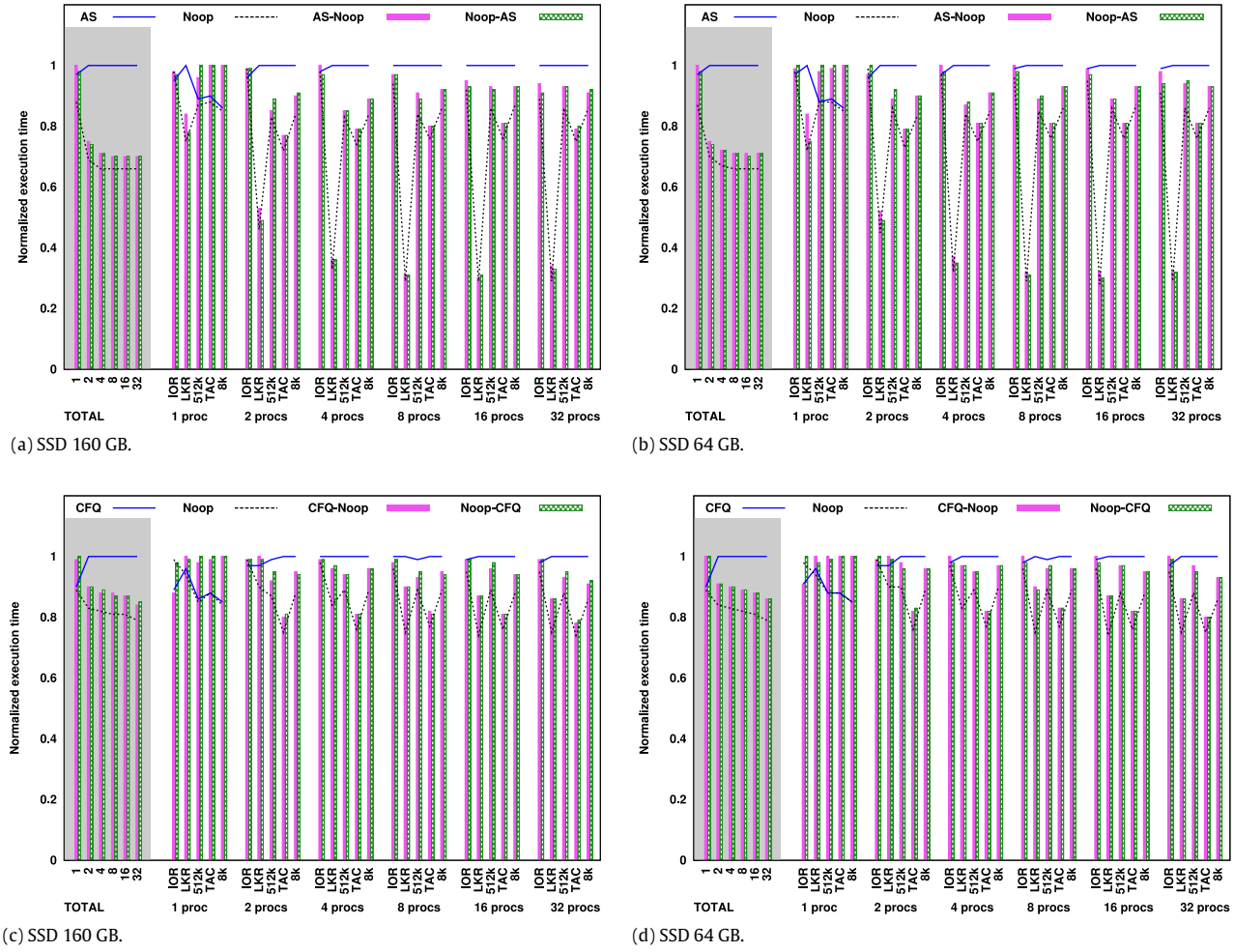
Secondly, the small overhead introduced by the disk simulator and DADS is more noticeable for SSDs, due to the very high performance offered by these devices. Consequently, application times are slightly increased. We have also realized that the overhead depends on the number of requests per seconds issued by the tests, because the instances of the simulator have to process more requests in the same amount of time. However, as we have said, many current systems have powerful processors with several cores, so we believe that the DADS overhead can also be reduced to almost zero for SSD drives in those up-to-date systems.

Despite the overhead, results will show that DADS adapts to the best scheduler, which means that our proposal and its virtual disks are also valid for SSDs.

6.2.1. Global execution

The first histogram in the figures shows the global behavior of DADS; it depicts the results for the overall application time of the test. As we can see, DADS follows the best scheduler, changing the scheduler, if necessary, when the number of processes also changes. Another interesting aspect is that our approach behaves the same for both disks, and achieves a quite similar performance.

It is worth emphasizing that, for 1 process, although it seems that DADS does not choose the best scheduler, it does. Noop is



(a) SSD 160 GB.

(b) SSD 64 GB.

(c) SSD 160 GB.

(d) SSD 64 GB.

Fig. 6. Configurations *AS–Noop* and *Noop–AS*, and *CFQ–Noop* and *Noop–CFQ* for SSD disks.

used during all the execution of the test, or after the first check DADS makes. The problem is that the overhead introduced by our mechanism is more noticeable for 1 process, due to the small application times achieved by the SSD disks.

6.2.2. IOR Read

For this benchmark, in the original kernel, the application time achieved by the four schedulers is quite similar. The highest difference between CFQ and Noop is less than 5%, and between AS and Noop is less than 12%. Therefore, sometimes, our technique is not able to decide which scheduler achieves the greatest performance, and spends all the time with the initial scheduler. Note that the mechanism to avoid a high rate of changes is not activated in this case, since DADS does not perform many changes, and it just keeps one of the schedulers most of the time.

The overhead is less noticeable with this test. The reason is that the number of requests per second issued by IOR is also less than the number issued by the other tests, because the IOR requests are larger and take longer to be served.

6.2.3. Linux Kernel Read

DADS adapts itself to the scheduler with the highest throughput (Noop), and spends all the time with it. If the best scheduler has not been selected during the previous test (IOR), a change is done at the first check. Our kernel presents, with all the configurations, the same behavior as the best scheduler in the original one.

For this test, DADS introduces the highest overhead. Now, the overhead is due not only to the number of requests per second issued, but also to the creation of the large amount of small processes to read the Linux kernel source. For each process, the simulator creates a control structure [7], and, although this creation process is optimized, it implies an increase in the application time.

6.2.4. 512 kB Strided Read, TAC, and 8 kB Strided Read

For 512k-SR, TAC, and 8k-SR, DADS presents a quite similar behavior, and gets the same performance as the vanilla kernel with the best scheduler. The best scheduler is selected in LKR, and no changes are done during these tests.

Regarding the overhead, in these tests, the number of requests per second is roughly the same, and is one of the highest. The overhead is quite similar in all the cases, and is larger than that introduced during the IOR execution. The overhead is more noticeable for 512k-SR and 8k-SR than for TAC, because application times achieved are smaller in the latter than in the former in the original kernel.

6.3. Webserver + Video

The "real-world" workloads have also been run for the six drives we have, but, due to space constraints, we only include results for the HD-250-32 and SSD-60 disks. Moreover, since DADS achieves for the Webserver test results similar to those obtained for the microbenchmarks (i.e., it always adapts to the best scheduler), these results have not been included either.

The situation is different for the Webserver + Video test due to the workload changes. Table 4 shows the results. For

Table 4
Average operations per second during the *Webserver* + *Video* test.

Disk	Original kernel		DADS	
HD-250-32	Deadline: 373 Deadline: 373	AS: 391 CFQ: 287	Deadline–AS: 398 Deadline–CFQ: 373	AS–Deadline: 403 CFQ–Deadline: 378
SSD-60	Noop: 2125 Noop: 2125	AS: 2155 CFQ: 2000	Noop–AS: 1950 Noop–CFQ: 1957	AS–Noop: 1940 CFQ–Noop: 1881

Table 5
DADS overheads.

Overhead	Disk	Processes		
		1	8	32
Whole simulation	HDD-250-32	2.4%	1.2%	0.6%
	SSD-60	11.4%	6.5%	4.6%
Request copy	HDD-250-32	0.3%	0.8%	0.5%
	SSD-60	1.4%	0.4%	0.5%

HD-250-32, DADS always follows the best scheduler and, when AS and Deadline are compared, our proposal even slightly outperforms the best scheduler in the vanilla kernel (AS) by up to 3%. However, for SSD-60, DADS changes several times from one scheduler to the other, since both get a similar performance. These changes, in addition to the DADS overhead, slightly decrease performance, up to 11% in the worst case; this result is consistent with those obtained for the microbenchmarks.

6.4. Overhead of DADS and the disk simulator

We have measured the overheads introduced by DADS and the disk simulator at two different levels: *whole simulation* and *request copy*. In the former, a regular simulation is done, but a scheduler change is never made. In the latter, we have calculated the overhead introduced when the system just copies the regular requests to the disk simulator. We use a fake version of our kernel where only the request copies are done, DADS is loaded, but no simulation is run.

We have computed both overheads by comparing the obtained application times with that of the original kernel. We have run all the microbenchmarks in a row (as described in Section 5) for 1, 8 and 32 processes, and for the HDD-250-32 and SSD-60 devices. Table 5 presents the results. As we can see, the request copy overhead is negligible for both devices. The whole simulation overhead is also irrelevant for the hard drive (it is 2.4% in the worst case). For the SSD, this overhead is more noticeable due to its very high performance. As we have already said, with a more up-to-date hardware, we believe that the DADS overhead can also be reduced to almost zero for SSD drives.

7. Related work

The idea of self-tuning systems is not new. A good survey and taxonomy of existing approaches are given by Denys et al. [3]. VINO is an example of such self-monitoring and self-adapting systems [24]. A problem of all these proposals is that they explore, but *do not implement*, an automatic adaptation. However, DADS has been implemented and tested inside the Linux kernel. An initial version of our disk simulator has been successfully used in REDCAP too [7].

Several authors present the idea that no one scheduler can provide the best possible I/O performance, and introduce proposal that manage different policies. ADIO [22] is an Automatic and Dynamic I/O scheduler selection algorithm that chooses between CFQ and Deadline. Deadline is selected when there are two or more requests with expired deadlines; otherwise, CFQ is chosen. A shortcoming of this approach is that, when there are no expired deadlines, CFQ is always used, while Deadline could provide a better performance for the current workload. Moreover, for a large number of processes, Deadline will be always active because most of the requests will exceed their deadline bounds, while CFQ could improve the performance in this case. DADS does not suffer this problem and, since DADS optimizes service times, it always selects the scheduler that provides the best performance. Another problem of ADIO is that it only selects between CFQ and Deadline, and, although can be modified to use AS and Noop, the comparison always has to be done with Deadline. However, in our method, the system administrator can choose any two I/O schedulers to compare.

Another option is the two-layer learning scheme [29], that automates the scheduling policy selection by combining workload-level and request-level learning algorithms and by using machine learning techniques. The proposed technique needs a training phase; according to the authors, their best results are obtained after running this phase during 24 h. In contrast, the off-line initialization of our time tables only took 100 min. Another problem is that, for a new given workload, their mechanism could not choose the best scheduler because it has not learned any information about the access pattern. However, since DADS takes a decision based on the current service times, this problem does not arise, and it can always select the scheduler that achieves the best performance.

Shenoy and Vin state that different applications need different I/O schedulers, and present Cello, a framework for meeting the diverse service requirements of applications [25]. Cello is a two-level disk scheduling architecture, that separate application-independent from application-specific scheduling policies, and facilitate the co-existence of multiple class-specific schedulers. Lund and Goebel present a similar approach for multimedia databases [16]. Although both use several schedulers, there is no dynamic scheduling. DADS, in contrast, dynamically selects the scheduler with the highest performance for any request and workload.

Regarding SSDs, specific I/O schedulers have been proposed for them [12,4,9,18], but, to the best of our knowledge, none about dynamic scheduling. If new I/O schedulers, providing different performances for different workloads, were identified for SSDs, DADS could help to dynamically select the best one.

8. Conclusions and future work

DADS is a framework that automatically and dynamically selects the best Linux I/O scheduler *at any moment for any workload* by comparing the performance achieved by each available scheduler. The implementation discussed here compares two schedulers, although it can easily be improved to support more. DADS has important features: (i) it can be used in any storage drive because the in-kernel disk simulator is able to simulate *any disk* [7]; (ii) it does not usually interfere with regular I/O requests since simulation is made out of the I/O path; and (iii) it is able to select the best scheduler at any time without previous knowledge.

To implement DADS, we have enhanced our in-kernel disk simulator. This new version simulates: (a) a disk cache; and (b) the inter-arrival times of requests and the "real" I/O behavior of the processes by considering their thinking times.

Results show that, at any moment, DADS selects the scheduler, of the two compared, that presents the best throughput. Even, for HDDs, when considering the total time of the test, it outperforms a "regular system" in several cases.

To sum up, we can claim that, by using DADS, performance achieved is always close to the best one, and system administrators are exempted from selecting a suboptimal I/O scheduler that can provide a good performance for some workloads, but may downgrade the system throughput when the workloads change.

As future work, we want to extend DADS to compare all the available schedulers simultaneously, and to select the best values for the tunable parameters that some schedulers have. We want also to evaluate DADS on hybrid hard disk drives.

Acknowledgments

This work was supported by Spanish MEC/MICINN/MINECO, and European Commission FEDER funds, under Grants TIN2012-38341-C04-03, TIN2009-14475-C04, TIN2012-34557, and CSD2006-00046.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, R. Panigrahy, Design tradeoffs for SSD performance, in: USENIX 2008 Annual Technical Conference on Annual Technical Conference, USENIX Association, 2008.
[2] M. Andrews, M.A. Bender, L. Shang, New algorithms for disk scheduling, Algorithmica 32 (2002) 277–301.
[3] G. Denys, F. Piessens, F. Matthijs, A survey of customizability in operating systems research, ACM Comput. Surv. (CSUR) 34 (2002) 450–468.
[4] M. Dunn, A.L.N. Reddy, A new I/O scheduler for solid state devices, Technical Report, Department of Electrical and Computer Engineering Texas A&M University, 2009.
[5] Filebench Benchmark, http://www.sourceforge.net/projects/filebench, 2013.
[6] R. Geist, S. Daniel, A continuum of disk scheduling algorithms, in: ACM Transactions on Computer System, Vol. 5, ACM Press, 1987, pp. 77–92.
[7] P. González-Férez, J. Piernas, T. Cortes, Simultaneous evaluation of multiple I/O strategies, in: Proceedings of the 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC–PAD), 2010.
[8] IOR Benchmark, http://www.ior-sio.sourceforge.net, 2012.
[9] S. Kang, H. Park, C. Yoo, Performance enhancement of I/O scheduler for solid state devices, in: IEEE International Conference on Consumer Electronics (ICCE), 2011.
[10] R. Karedla, J.S. Love, B.G. Wherry, Caching strategies to improve disk system performance, Computer 27 (1994) 38–46.
[11] J.-H. Kim, D. Jung, J.-S. Kim, J. Huh, A methodology for extracting performance parameters in solid state disks (SSDs), in: 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2009.
[12] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, S.H. Noh, Disk schedulers for solid state drivers, in: Proceedings of the 7th ACM International Conference on Embedded Software, 2009.
[13] R. Love, Linux Kernel Development, second ed., Novell, 2005.
[14] P. Lu, K. Shen, Multi-layer event trace analysis for parallel I/O performance tuning, in: Proceedings of the 2007 International Conference on Parallel Processing (ICPP-07), 2007.
[15] C.R. Lumb, J. Schindler, G.R. Ganger, D.F. Nagle, Towards higher disk head utilization: extracting free bandwidth from busy disk drives, in: Proccedings of the 4th Symposium on Operating Systems Design and Implementation, 2000.
[16] K. Lund, V. Goebel, Adaptive disk scheduling in a multimedia DBMS, in: Proceedings of the Eleventh ACM International Conference on Multimedia, 2003.
[17] OCZ, OCZ Octane SATA III 2.5" SSD, http://www.ocztechnology.com/ocz-octane-sata-iii-2-5-ssd.html, 2011.
[18] S. Park, K. Shen, FIOS: a fair, efficient flash I/O scheduler, in: Proceedings of the 10th USENIX Conference on File and Storage Technologies, 2012.
[19] C. Ruemmler, J. Wilkes, An introduction to disk drive modeling, Computer 27 (1994) 17–28.
[20] Samsung, Samsung SSD MZ-5PA256, http://www.samsung.com/es/consumer/pc-peripherals-printer/memory-storage/ssd/MZ-5PA256/EU/index.idx?pagetype=prd_detail&tab=specification#s275_TableView, 2012.
[21] J. Schindler, G.R. Ganger, Automated Disk Drive Characterization, CMU SCS, Technical Report CMU-CS-99-176, 1999.
[22] S. Seelam, J.S. Babu, P. Teller, Automatic I/O scheduler selection for latency and bandwidth optimization, in: Proceedings of the Workshop on Operating System Interface on High Performance Applications, in Conjunction with the 14th International Conferences on Parallel Architectures and Compilation Techniques, IEEE y ACM, 2005.
[23] M. Seltzer, P. Chen, J. Ousterhout, Disk scheduling revisited, in: Proceedings of the USENIX Winter 1990 Technical Conference, USENIX Association, 1990.
[24] M. Seltzer, C. Small, Self-monitoring and self-adapting operating systems, in: Proceedings of the 6th Workshop on Hot Topics in Operating Systems, 1997.
[25] P.J. Shenoy, H.M. Vin, Cello: A disk scheduling framework for next generation operating systems, ACM SIGMETRICS Perform. Eval. Rev. 26 (1998) 44–55.
[26] E. Shriver, Performance modeling for realistic storage devices, Ph.D. thesis, 1997.
[27] T.J. Teorey, T.B. Pinkerton, A comparative analysis of disk scheduling policies, in: Communications of the ACM, Vol. 15, ACM Press, 1972, pp. 177–184.
[28] B.L. Worthington, G.R. Ganger, Y.N. Patt, J. Wilkes, On-Line Extraction of SCSI DISK Drive Parameters, Technical Report, Hewlett-Packard Laboratories, 1997.
[29] Y. Zhang, B. Bhargava, Self-learning disk scheduling, IEEE Trans. Knowl. Data Eng. 21 (2009) 50–65.

Pilar González-Férez holds M.S. (1996) and Ph.D. (2012) degrees in computer science from the Universidad de Murcia in Spain. She is an assistant professor at the Universidad de Murcia. Currently, she has a postdoc position at the Institute of Computer Science (ICS) of the Foundation for Research and Technology — Hellas (FORTH). Her research is focused on operating systems, storage systems, and distributed systems. She has presented her results in different International conferences, where she has also published them. She has also participated in several Spanish funded projects.

Juan Piernas received the M.S. degree in computer science in 1994 and the Ph.D. degree, also in computer science, in 2004 (both from the Universidad de Murcia). He got a postdoc position at the Pacific Northwest National Laboratory (USA) in 2006. Currently, he is an associate professor at the Universidad de Murcia (Spain). His main research concentrates on operating systems, storage systems, and distributed systems. He has published more than 20 papers in different journals and conferences in these fields. He has also participated in several Spanish funded projects.

Toni Cortes is the manager of the storage-system group at the BSC and is an associate professor at Universitat Politècnica de Catalunya. He received his M.S. in computer science in 1992 and his Ph.D. also in computer science in 1997 (both at Universitat Politècnica de Catalunya). His research concentrates in storage systems, programming models for scalable distributed systems and operating systems. He has published 76 papers, 2 book chapters, and has co-edited one book. He has also advised 8 Ph.D. theses. Dr. Cortes has been involved in several EU projects and has also participated in cooperation with IBM (TJW research lab).