Improving I/O Performance through an In–Kernel Disk Simulator

PILAR GONZÁLEZ-FÉREZ, JUAN PIERNAS[1] AND TONI CORTES[2]

[1] *Universidad de Murcia, Spain*
[2] *Universitat Politècnica de Catalunya and Barcelona Supercomputing Center, Spain*
Email: pilar@ditec.um.es,piernas@ditec.um.es,toni.cortes@bsc.es

This paper presents two mechanisms that can significantly improve the I/O performance of both hard and solid-state drives for read operations: KDSim and REDCAP. KDSim is an in-kernel disk simulator that provides a framework for simultaneously simulating the performance obtained by different I/O system mechanisms and algorithms, and for dynamically turning them on and off, or selecting between different options or policies, to improve the overall system performance. REDCAP is a RAM-based disk cache that effectively enlarges the built-in cache present in disk drives. By using KDSim, this cache is dynamically activated/deactivated according to the throughput achieved. Results show that, by using KDSim and REDCAP together, a system can improve its I/O performance up to 88% for workloads with some spatial locality on both hard and solid-state drives, while achieves the same performance as a "regular system" for workloads with random or sequential access patterns.

Keywords: Virtual disk; disk modeling; simultaneous evaluation; REDCAP; I/O performance; disk cache

Received 00 January 2009; revised 00 Month 2009

1. MOTIVATION

Over the years, advances in disk technology have been very important. Despite that, the disk I/O subsystem still remains a mayor performance bottleneck in many computers, since both hard disks (HDD) and solid state drives (SSD) are rather slower, regarding latency and bandwidth, than other components, such as CPU, RAM or GPU [1].

There are several mechanisms that can improve the I/O performance, being the cache hierarchy one of the most important parts. This hierarchy consists of several caches that take advantage of the temporal locality, but that usually exploit different spatial localities. For instance, while the page cache can exploit the spatial locality at a file level, the caches built in the disk drives (*disk caches*) exploit that locality at a disk-block level. The former can overlap I/O and computation by prefetching blocks of sequentially-accessed files, while the latter can reduce the I/O time by reading ahead several disk blocks in the same operation.

In this paper, we propose to improve the disk I/O performance by increasing the performance of the disk caches, what we achieve by increasing their effective sizes. The size determines the hit rate of a cache to a large extent [2]. For instance, disk caches are usually split into segments to allow prefetching on multiple sequential streams [3]. Each I/O stream can be treated as having its own cache by assigning it to a segment. I/O performance can be efficiently improved when the number of concurrent streams is smaller than the number of cache segments. But, if the number of streams exceeds the number of segments, there is no benefit. So a bigger cache means more segments or larger segments and thereby a better throughput. System designers generally consider that a disk cache should be 0.1%-1.0% of the total disk capacity to improve I/O performance [4, 5]. But, though manufactures tend to integrate larger caches, sizes are still rather small compared to the disk capacities. For example, a 4 TB disk usually has 128 MB of cache, only 0.003% of its capacity. Current technology also indicates that this imbalance will not change in the short term.

Since operating systems have no control over disk caches, we propose to improve their performance by adding a new level to the cache hierarchy. The new level, called REDCAP (RAM Enhanced Disk Cache Project), is a cache of disk blocks in RAM whose aim is to reduce the I/O time of read requests by mitigating the problem of a premature eviction of blocks [6, 7]. Its essential ideas are: enlarging a disk's cache, emulating its behavior with the purpose of improving the read I/O time, and profiting the disk's read-ahead mechanism by prefetching some disk blocks. Note that a modern computing system usually has a large main memory, and that REDCAP will only use a small percentage of that memory.

A problem of the caches (and many other I/O mechanisms) is that, although they can significantly reduce I/O times, they are not optimal because their benefits depend on the current workload, file system, etc. They even have one or more worst-case scenarios that can downgrade the performance. For instance, for workloads where data is accessed only once, a buffer cache provides no benefit from keeping data in memory, and it can even downgrade the system performance due to the eviction of pages of running processes. REDCAP is not oblivious to this problem. However, we can avoid these worst-case scenarios by activating/deactivating a mechanism, or changing from one to another, depending on the expected performance. To achieve this dynamic behavior, we need a means to simultaneously evaluate several I/O strategies. Obviously, only one would be active at a specific moment, and the rest should be simulated.

To obtain such a general system-wide simulation, we propose KDSim, an in-Kernel Disk Simulator that fulfills the above requirement. KDSim is implemented inside the Linux kernel by creating a *virtual disk* that captures the behavior of HDDs and SSDs, and helps to simulate part of the I/O subsystem. Unlike other simulators [8, 9, 10, 11], which take into account almost all the internal elements, and provide a high-precision simulation, KDSim gives *reliable estimations of I/O times per group of requests without making an exact simulation per request*. By working this way, our simulator is *valid for whatever HDD and SSD drives* and gets rid of two important shortcomings of existing simulators: they are valid only for some specific disk models, and/or they run in user-space.

In this paper, we use KDSim for controlling REDCAP's performance, turning it on/off accordingly. Therefore, we design, implement and evaluate both proposals. We analyze the benefits of REDCAP and KDSim by using two HDDs and two SSDs, several workloads, and different I/O schedulers. Results show that, by using KDSim, REDCAP is correctly turned on/off, depending on the workload, to achieve the maximum performance. For workloads with some spatial locality, REDCAP gets improvements of up to 80% for HDDs and 88% for SSDs. For random or sequential workloads, it roughly has the same performance as a regular system. Results also show that, although our disk model is quite simple, it is "good enough" to allow us to control REDCAP's performance.

This paper is organized as follows. Section 2 describes how a disk cache works and what can be done to improve its performance. Section 3 presents KDSim, our disk simulator. Section 4 describes REDCAP, our RAM-based disk cache, and how we use KDSim to control the performance of REDCAP. Sections 5, 6, and 7 evaluate our proposals. Section 8 discusses related work and Section 9 draws our conclusions.

2. RATIONALE

Many secondary storage devices, such as hard drives, have a *disk cache*. These cache usually acts as a block cache and speed-matching buffer [12, 13], and plays a crucial role in improving the performance of the I/O subsystem [1]. Although the exact working of a disk cache is unknown (manufactures do not disclose many internal details of their drives), several important mechanisms used by these devices are public. Next paragraphs describe some of them.

A disk usually prefetches several sectors into its cache when serving a read request. The goal of this read-ahead of data [3, 4, 8] is to avoid cache misses by anticipating future read requests. Thus, the I/O performance improves because many read requests are served without accessing the disk media. Disk caches exploit spatial locality since prefetched sectors are adjacent on disk. As a consequence, disk caches usually improve sequential access patterns and access patterns with some spatial locality. However, for random access patterns or without spatial locality, disk caches do not provide any improvement.

In addition, a disk cache is usually split into several segments. The number of segments can be static or dynamic, depending on the drive [3]. Ideally, each segment can be assigned to an independent I/O stream, so each stream believes it has its own disk cache. However, when there are more I/O streams than segments, the disk cache could provide no improvement.

Write requests are also affected by disk caches, since they usually use a write-back policy and immediate reporting [3]. Write-back policy means that data is written to the disk cache first, and then to the disk media. Immediate reporting makes a write request be considered "done" as soon as it is in the cache (but not necessarily in the disk media), so the issuing application will not block.

As we can see, in any of the above techniques, the size of the disk cache is important, specially for read requests, because the size determines the hit rate of the cache to a large extent [2]. A larger cache means more prefetched blocks, less evictions, more segments, more immediate reportings, etc. Unfortunately, disk caches are small compared to their disks' capacities, and they have not been as effective as expected.

Since the cache size is important, we aim at improving the read performance of the disk drives by effectively making their caches larger. Since disk caches already existing in the drives cannot be modified, we propose to achieve this by introducing a new large disk cache in main memory. This new cache will prefetch a large number of adjacent disk sectors on every read miss. This way, we will profit the underlying read-ahead performed by the drives, and reduce the overhead imposed by every disk transfer (may small reads will become a single large read). Our new cache will work as long as the current workload presents some spatial locality. This locality is usually common for applications [14, 15] and for file systems that split the disk blocks into several groups [16, 17, 18]. However, since spatial locality can temporally disappear in the current workload, we will need a mechanism to detect these situations and deactivate our cache until the spatial locality comes back again.

3. KDSIM

KDSim is a disk simulator implemented inside the Linux kernel [19]. It creates a *virtual disk* that works as both a driver and a disk device. It has its own I/O scheduler to sort incoming requests, and manages its own request queue. This virtual disk serves *virtual requests*, which are generated from the corresponding real requests by using the same parameters (process id, type, sector and size) but no data.

KDSim works as follows: 1) move some virtual requests from its auxiliary queues to its scheduler queue; 2) fetch the next virtual request from the scheduler queue; 3) get the I/O time to serve the request from a table-based model of the real disk; 4) sleep this time to simulate the disk operation (this sleep simulates the scheduler effect in the request arrival order); and 5) complete the request, and delete it.

3.1. Disk model

We model the behavior of a disk drive with a dynamic-table. Although the time of a disk operation depends on several factors [8, 20], for simplicity reasons, our table-based model only uses three input parameters: *request type* (read or write), *size*, and *inter-request distance* from the previous request.

Request type has been considered since reads and writes have different I/O times [8, 20, 21, 22]. Especially for SSDs, writes are often slower than reads, although SLC models may have a balanced read/write performance.

We have also considered the request size as a parameter for two reasons. Firstly, transfer time of an I/O request is usually proportional to its length, specially in SSDs. Secondly, our model indirectly takes into account disk caches, and I/O time significantly depends on the request length in a cache hit. Results in Section 6 show that considering the size is fundamental for the accuracy of the model.

Finally, the inter-request distance is important for hard disks because the I/O time of a request depends on the logical distance from the previous one [8]. For SSDs, since they do not have seek or rotational delays, this distance could be dismissed as an input parameter. However, according to our tests, inter-request distance is important in these devices too since there are small differences in I/O time between sequential and non-sequential access patterns.

We are aware that, with only these parameters, the estimated I/O time for each individual request is not going to be very precise. However, we are interested in estimating the *total I/O time for a large set of requests*. Section 6 shows that these three parameters, along with the dynamic behavior of the table, are enough to accurately model the real disk.

Our model manages a table for reads and a table for writes. Rows represent request sizes. Each table has thirty two rows for sizes from one block (4 kB) to 32 blocks (128 kB), the minimum and maximum request sizes allowed by the file system, respectively. Although the scheduler can merge requests and produce one larger than 128 kB, these requests are rare and have not been considered (they are treated as 128 kB requests). Columns represent ranges of inter-request distances. Column 0 corresponds to a distance of 0 kB; it captures many disk cache hits. Columns from $n = 1$ to $n = 18$ represent small inter-request distances, from $4 \cdot 2^{n-1}$ kB to less than $4 \cdot 2^n$ kB. The other columns corresponds to large distances: $[1\text{ GB}, 2\text{ GB})$, $[2\text{ GB}, 3\text{ GB})$, etc. Cells store estimated I/O times.

3.1.1. *Dynamic behavior*

To model the disk behavior in a precise way, *to adapt the model to the current workload*, and *to catch the effects of the disk cache*, we make tables dynamic. For each request dispatched by the real disk, its I/O time is calculated, and the corresponding cell (either in the read or write table) is updated. Since the update does not take into account any disk-specific feature, this approach is able to model the behavior of both hard and SSD drives.

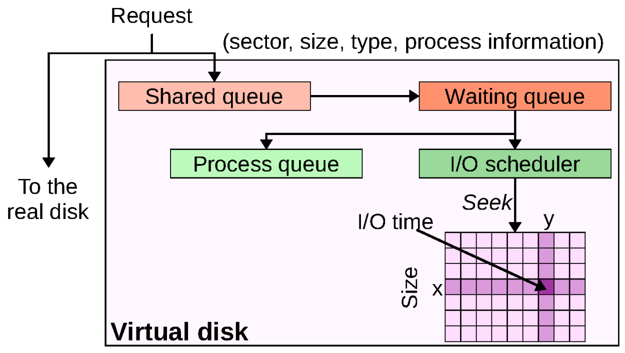


FIGURE 1. Auxiliary and I/O scheduler queues and table of the virtual disk.

Each cell stores the *last S* observed I/O times for the cell, and returns the average of these values. Our model forgets past times that depend on past workloads, and keeps values that depends on the current workloads. We choose $S = 64$ after analyzing the sensitivity to the number of averaged values per cell (see Section 6).

We are aware that our model does not explicitly consider several disk features, such as zoned recording, track/cylinder skew, and bad sector remapping. The impact of these features, however, is indirectly modeled through the I/O times obtained from the real disk during the dynamic update of the tables.

3.2. Request management

For a right scheduling, the virtual disk should know the order in which requests were issued, and whether a request was issued when the previous one was complete or not. The virtual disk has to schedule requests of a process in the same order as the system had scheduled them. To fulfill these requirements, the virtual disk controls the arrival order of requests and dependencies between them.

Read requests are usually synchronous with respect to their corresponding applications (they block the application until they are served). Hence, they introduce dependencies among requests of the same process. Dependencies due to synchronous reads appear among requests of related processes too. For instance, if a parent waits for a child, dependencies among their requests arise. There also exist, however, asynchronous read requests issued by applications or by the Linux kernel itself to support sequential file prefetching [23]. Thus, KDSim distinguishes between two kinds of read requests: *synchronous reads* and *asynchronous reads*.

Write requests are usually asynchronous with respect to applications because the operating system defers them in memory. So, they do not have dependencies.

The virtual disk maintains the arrival order of the requests and their dependencies by managing three queues in addition to the scheduler queue. It also implements a heuristic to decide the insertion of requests into the scheduler queue. Figure 1 presents the flow of requests throughout these queues.

The *shared queue* connects the disk simulator with the operating system. For each request, just before queueing it in the scheduler of the real disk, the system copies its main parameters to this queue. To not introduce overhead in the regular I/O path of a request, the system does not create the virtual request, and just copies its parameters. KDSim itself will create the new virtual request by using these parameters, and it will insert the virtual request into the *waiting queue*.

The *waiting queue* stores virtual requests that cannot be inserted into the scheduler queue due to dependencies. It also maintains the arrival order of requests. A request in this queue is moved to the scheduler iff it has solved its dependencies. These dependencies are controlled via the following heuristic:

- Write requests are queued immediately in the scheduler;
- A *synchronous read* will be inserted into the scheduler queue if no another *synchronous read* of the same process is either ahead in the waiting queue or in the scheduler queue. A *synchronous read* can be inserted into the scheduler even when there is already an *asynchronous read* of the same process in this queue;
- An *asynchronous read* request is enqueued in the scheduler if there is no *synchronous read* request of the same process ahead in the waiting queue;
- The first request of a new process is enqueued in the scheduler when the last request of its parent process, issued before child creation, has been served. If a parent process waits for the completion of its child, none of the new requests of the parent will be enqueued until the child exits.

This heuristic copies the default Linux behavior, and imitates how requests arrive to the scheduler of a regular disk. We are aware that all the dependencies are not controlled, but most of them are captured, so the virtual disk can process requests in the order it had used if been the real disk.

The *process queue* records the requests in the scheduler of KDSim. We need it since Linux handles scheduler queues as black boxes that cannot be scanned. This queue controls dependencies with requests dispatched but not completed.

Requests are moved from the shared to the waiting queue in a FIFO order. Then requests are moved from the waiting queue to the scheduler and process queues following the aforementioned heuristic. Requests in the scheduler queue are served following the order imposed by the I/O scheduler used by KDSim. Finally, a request in the process queue will be deleted only after being completed by KDSim.

3.3. Training the tables

Tables can be initialized on-line or off-line. For the off-line, we run a training program that is run only once for every disk model. The training is usually “fast”; in our system, it took 80 minutes for a 400 GB HDD, and 2 minutes for a 64 GB SSD. The program issues requests in a random way and uniformly distributed over all the cells. The value of each cell is the mean of the samples obtained for it.

With the on-line configuration, tables are first zeroed, and then our model learns the disk behavior as requests are served. For a not-yet-updated cell, the model returns the average of the corresponding column as I/O time, if this value is not zero; otherwise, it uses the nearest column with non-zero cells.

4. A USE CASE: THE RAM ENHANCED DISK CACHE PROJECT

REDCAP introduces a new cache of disk blocks that can reduce the I/O time of read requests. It (a) enlarges a disk cache with a small portion of RAM; (b) prefetches disk blocks; and (c) leverages the read-ahead mechanism of a disk drive. REDCAP consists of a *cache*, a *prefetching* method, and an *activation-deactivation algorithm* (*ADA*).

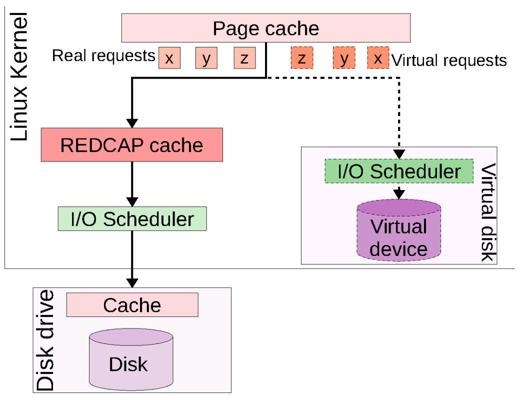
The *REDCAP cache* introduces a new level in the cache hierarchy, between the page and disk caches. This new cache has disk blocks prefetched by reads on every cache miss. It is stored in RAM and has a fixed size of C blocks. As a disk cache, our cache is split into N segments (REDCAP segments) that are managed independently. A segment has S consecutive blocks (where $C = N \times S$), and is the transfer unit of the prefetching technique. The replacement algorithm is Least Recently Used (LRU).

The prefetching technique mimics the read-ahead of a disk cache. Each request is handled depending on its type, and before being inserted into the scheduler. For read requests, REDCAP searches the corresponding segments in its cache. For a hit, the request is serviced from the cache, and no prefetching is done. For a miss, the blocks of the segment are read. Some blocks are those requested by the application, while others (the *prefetched blocks*) are read to complete the segment. Since some prefetched blocks may be located before the requested blocks on disk, REDCAP can leverage the *immediate read*[3] usually made by HDDs.

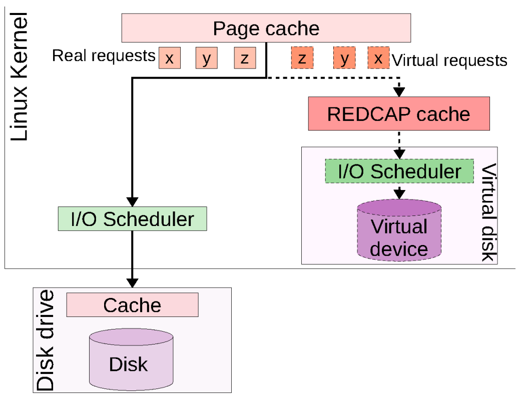
A read request that affects several segments is handled as n small partial requests, one for each segment. All the partial requests are handled as “normal” requests, and when all of them complete, the original one is completed.

No prefetching is done on write requests. On a cache hit, we invalidate the appropriated blocks. We do not update the affected disk blocks, because we do not expect any access to these blocks in the short term. Misses are just ignored. In both cases, the write request is directly issued

[3]When settling the heads, hard disks attempt a read as soon as the heads are near the desired track [8]. So, blocks before the requested ones can be read and stored in the disk cache too. The net effect is similar to a backward prefetching, which drives neither really perform or detect.



(a) REDCAP is active



(b) REDCAP is inactive

FIGURE 2. KDSim used in a REDCAP system. For the sake of clarity, we omit the KDSim queues.

to the real disk, with no modification, in order not to put the consistency of the file system at risk.

Two REDCAP's aims are to overlap computation and I/O, and to be I/O-time efficient. REDCAP converts workloads with thousands of small requests into workloads with disk-optimal large sequential ones, mitigating the overhead that data transfers between the disk controller and the RAM suppose for each request.

Our prefetching policy is completely different from that made by the page cache of Linux. REDCAP prefetches blocks that are adjacent on disk, so it exploits spatial locality at disk level. The page cache usually reads in advance data blocks of the same file, that could be fragmented on disk, what is less efficient (this problem also appears on other prefetching techniques based on file access patterns [14]). The page cache, therefore, exploits spatial locality at file level.

In exploiting the principle of spatial locality, REDCAP takes advantage of the organization in block groups of some file systems, where data blocks and i-nodes of all the regular files in a directory are put together in the same group [24]. As results show, even with a small portion of RAM, REDCAP is able to obtain a performance that is larger than that obtained by the usual policies of the page cache. Moreover, we are not interested in an integration of REDCAP in the page cache, since the page cache could evict blocks that REDCAP could need later on.

4.1. Activation-deactivation algorithm

An activation-deactivation algorithm (ADA) completes REDCAP by controlling its performance to make its prefetching dynamic [6]. ADA continually analyzes the REDCAP performance, and turns its cache on/off according to it.

REDCAP works in two main states. In the *active* state, REDCAP handles requests as explained. But if ADA detects that access time is worse with than without the cache, it moves REDCAP to the *inactive* state. In the inactive state, no requests are processed and no prefetching is done. But ADA studies the possible success of REDCAP by simulating that it is on and recording the hits/misses on each read. When ADA detects that REDCAP could be efficient, turns it on again.

KDSim is used for implementing ADA. As Figure 2 shows, when REDCAP is active, KDSim simulates the behavior of the real disk in a normal system. In the inactive state, it simulates the behavior of the real disk in a REDCAP system. ADA compares the mean time required to serve one block of 4 kB by a REDCAP system and by a normal system. For this, the following information is stored:

- B_{Redcap} is the number of blocks (4 kB) served by REDCAP. For a hit, it includes the requested blocks, and for a miss, it includes the requested and prefetched blocks
- T_{Redcap} is the time that REDCAP needs to serve B_{Redcap} blocks. For each request, it is computed as the time elapsed since the request arrives to REDCAP until it ends. It can be the time of a cache hit, of a cache miss, or even both times.
- B_{Normal_System} denotes the number of requested blocks, also of 4 kB.
- T_{Normal_System} is the time that a regular system needs to serve B_{Normal_System} blocks from disk.

The service time is calculated as the elapsed time since the request is queued in the scheduler until its completion. This time includes the waiting time in the scheduler, and the I/O time of the drive. An interesting point is that ADA uses the table model of KDSim for computing the I/O times of both the real and virtual disk. As our accuracy study shows (see Section 6), the virtual disk's behavior is very similar to that of the real disk, but not identical. So, for the sake of comparison, we always use I/O times obtained from the same source.

By using the above data, ADA says that if condition

$$\sum_{i=1}^{1000} \frac{T_{Redcap_i}}{B_{Redcap_i}} < \sum_{i=1}^{1000} \frac{T_{Normal\_System_i}}{B_{Normal\_System_i}} \qquad (1)$$

is true, REDCAP is improving access time and it has to be active; otherwise, it has to be inactive. The verification considers the last 1000 requests (see Section 6).

In the active state, each original I/O request, without any modification, is inserted into KDSim. The time of the virtual disk corresponds to the time to serve the request in a normal system. For each request, KDSim provides B_{Normal_System} and T_{Normal_System}, and REDCAP calculates B_{Redcap} and T_{Redcap}.

In the inactive state, KDSim simulates the behavior of the real disk as if REDCAP was on. For each cache miss, the corresponding request is issued to the virtual disk. Since REDCAP is simulated, B_{Redcap} is exactly known, and T_{Redcap} is calculated as $T_{Redcap} = T_{Virtual\_Disk} + T_{CHit}$, where $T_{Virtual\_Disk}$ is the service time of the virtual disk, and corresponds to the time that REDCAP needs to read the requested and prefetched blocks from disk; T_{CHit} is the time of the cache hits, and is estimated by using values stored for cache hits during the active state. Simultaneously, the system calculates B_{Normal_System} and T_{Normal_System}.

5. EXPERIMENTS AND METHODOLOGY

We implement REDCAP and KDSim in a Linux kernel 2.6.23 (*REDCAP-VD kernel*), and for comparison we use the same kernel with no modification.

Our experiments are conducted on a 2.67 GHz Intel dual-core Xeon system with 1 GB of RAM and five disks. The first one has a Fedora Core 8 operating system, and collects traces. The other are the test drives, two HDDs and two SSDs, but here we only present results for one HDD and one SSD. Nevertheless, results obtained for the other devices are pretty similar to those provided here [19].

The HDD is a Seagate ST3400620AS disk with a capacity of 400 GB and a cache of 16 MB. The SSD is an Intel X-25M SSDSA2MH160G2C1 that has a capacity of 160 GB and uses MLC flash memories. Both have a clean *Ext3* file system, containing nothing but files used for the tests. We have chosen Ext3 here because it is the predominant file system in Linux 2.6.23. Moreover, since REDCAP has been already tested with five different Linux file systems (Ext2, Ext3, XFS, JFS and ReiserFS), and has proved to provide significant improvement with all of them [7], we think that results obtained with one of those file systems represent our proposal's benefits very well.

We use the following benchmarks, each executed for 1, 2, 4, 8, 16 and 32 processes, that cover different access patterns:

- *Linux Kernel Read (LKR).* It reads the sources of the Linux kernel 2.6.17 by using the command "`find -type f -exec cat {} > /dev/null \;`". Each process uses its own copy of the kernel files.
- *IOR Read (IOR-R).* We test the REDCAP behavior in parallel sequential reads with IOR 2.9.1 [25]. We use the POSIX API, 1 GB file per process, and a transfer unit of 64 kB. Before running the test, we create the files in parallel with IOR too.
- *TAC.* Each process reads a file backward with the command `tac`. Files are the same as those from IOR-R.
- 8 *kB Strided Read (*8*k-SR).* Each process reads 1 GB file with a strided access pattern with small strides. The test reads a first block (4 kB) at offset 0, skips two blocks (8 kB), reads the next block, skips another two blocks, etc. Files are the same as those read by IOR-R and TAC.
- 512 *kB Strided Read (*512*k-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.
- *All in a row (AIR).* The previous tests are run one after another, without rebooting the computer until the last is done. This shows how KDSim adapts to workload changes. Since some tests use the same files, the execution order (TAC; 512k-SR; 8k-SR; LKR; and IOR-R) tries to reduce the effect of the page cache.
- *All in parallel (AIP).* This test runs all the previous ones (but *AIR*) in parallel, and finishes when each one has been run at least once. If a benchmark ends when others are still in their first run, it is launched again. This test is executed for 1, 2 and 4 processes, i.e., each benchmark is run for that number of processes, and each one reading its own files. The aim is to analyze the behavior of our proposals when the workload is a blend of different access patterns.

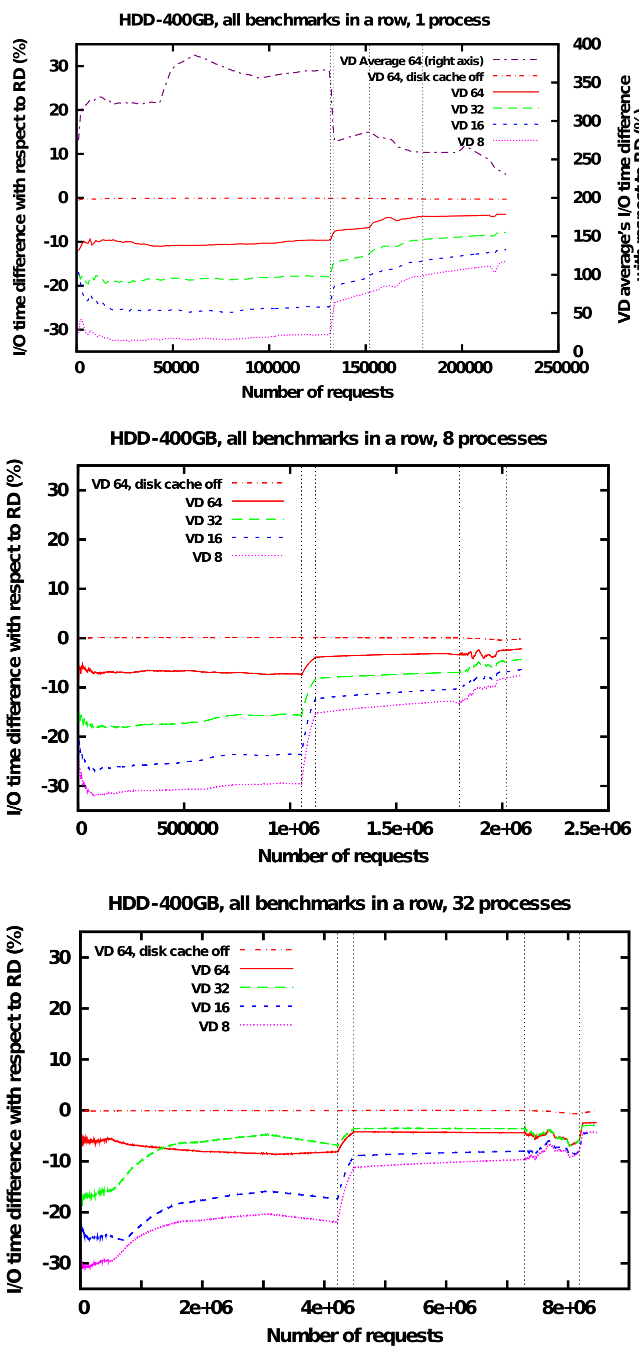
As schedulers, we use CFQ and AS for the HDDs, and CFQ and Noop for the SSDs. CFQ and AS have been the most widely used in Linux [23], and CFQ is the default one as of Linux kernel 2.6.23. Noop usually gets better performance for SSDs than the other available Linux schedulers [26, 27]. For a given test, the real and virtual disks use the same scheduler.

Regarding KDSim, the read and write tables obtained from the off-line training are provided each time the system is booted.

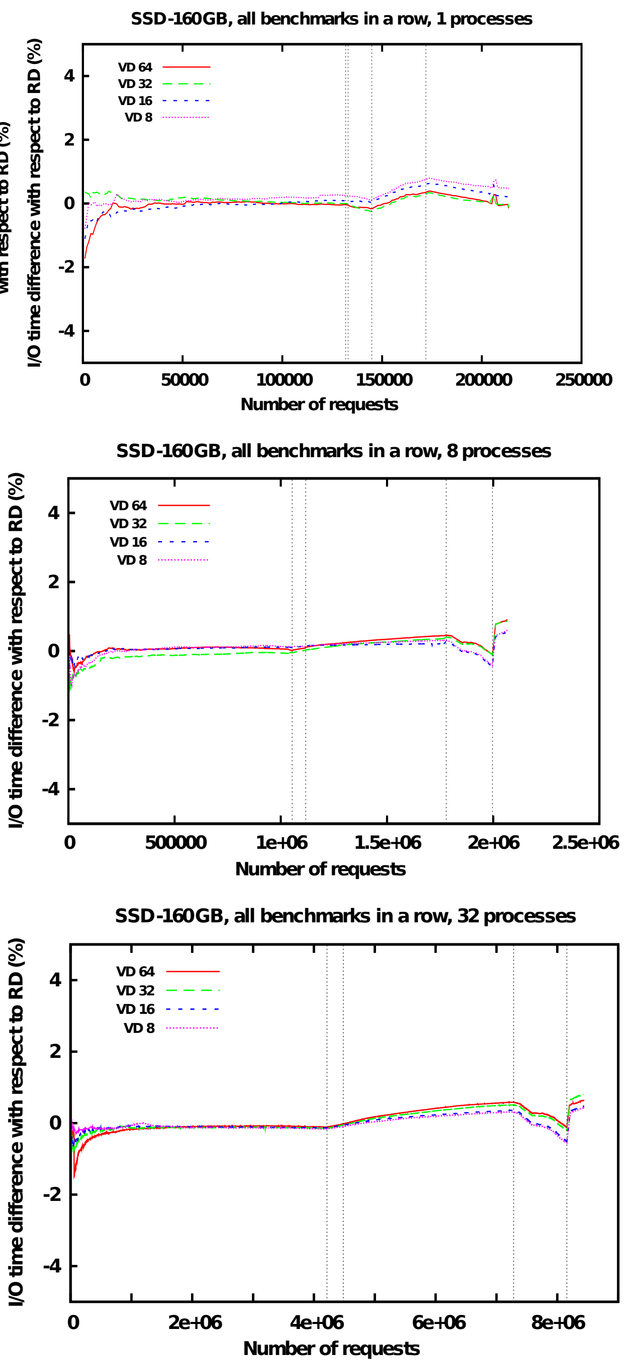
6. ACCURACY OF THE DISK MODEL

This Section analyzes the accuracy of KDSim. Since KDSim does not explicitly simulate the disk cache, which is one of the most important features to simulate a disk behavior request by request, we do not calculate its demerit figure [8]. Moreover, note that it does not make sense to compare performance achieved by both disks on every request, since it would imply a significant overhead, and it would be impossible to take the right decision.

As I/O time comparisons are done for large numbers of requests (1000 in our case), our simulator only needs to give good estimations per set of requests. By using the total I/O time of several requests, many over and under estimations cancel each other out, making the overall estimation more accurate. We are aware that our model does not fully simulate the disk behavior, being just an approximation. Our intent is not to develop the best possible model, but to develop one "alike enough" to allow us to evaluate the performance of I/O mechanisms.



(a) HDD



(b) SSD

FIGURE 3. Difference (% of I/O time) of the virtual disk (VD) with respect to the real disk (RD) in the AIR test for the HDD (a) and SSD (b). Vertical dashed lines mark the end of a benchmark and the beginning of the next one. Benchmarks conforming the *AIR* test are executed in a row, in the order TAC, 512k–SR, 8k–SR, LKR, and IOR. Note that for the SSD the Y-axis range is $[-5,5]$.

The real and virtual disks serve the same requests, and use CFQ as scheduler. I/O times compared are those directly given by the disks. The real disk does not use the table model to estimate I/O times. REDCAP is neither active nor simulated. Although both disks receive the requests in the same order, they could dispatch them in a different order since they are independent from each other.

We use the AIR test because it runs the TAC, 512k-SR, 8k-SR, LKR, and IOR-R benchmarks in a row, that is, not in parallel. Therefore, its execution shows the accuracy of our model in all the benchmarks, and how our model adapts to workload changes. It also shows how the dynamic update of the tables allows KDSim to follow the behavior of the real drive.

6.1. Accuracy for Hard Disk Drives

For the HDD, and 1, 8 and 32 processes, Figure 3(a) shows the evaluation of different configurations based on the number of values averaged per cell. We analyze results of averaging eight ("VD 8" in the figure), sixteen ("VD 16"), thirty two ("VD 32"), and sixty four ("VD 64") values per cell. Each point of the figure represents the difference between the last 1000 requests served by each disk. A negative difference means that the virtual disk is "faster" than the real one.

Major differences are observed at the beginning of the test, when TAC is run. Like a sequential access pattern profits the prefetching of a disk drive, a backward access pattern takes advantage of the *immediate read* of the drive (see Section 4). However, at a cache miss, the time needed to read the requested blocks is larger than in the forward access due to the backward seeks. Although our read table is updated with all these times, the disk cache effect has a noticeable impact, making the virtual disk faster than the real one.

After TAC, difference decreases quickly, being less than 5% on average when each cell stores the mean of the last sixty four values. The reason of this fast adaptation is that, although our tables have many cells, just a small set of cells is used and updated, even when the test is run for 32 processes [19].

With these results, we can claim that the "VD 64" configuration of the virtual disk has a good behavior, and that *the virtual disk closely matches the real one*.

One reason for which the virtual disk's behavior is not the same as the real disk's is the difficulty to simulate the disk cache. To analyze this, we run the same test with the disk cache off, and only the "VD 64" configuration. In Figure 3(a) results appear as "VD 64, disk cache off". Now, the virtual disk matches the real one very accurately, and differences are smaller than 0.2%, even for TAC.

Finally, to show that the request size is important, we run the same test, but without considering that parameter: for a given inter-request distance, we use the average of the values in its column (for all the sizes). This test is run once and for 1 process because the virtual disk becomes very slow and the execution takes a long time. The result appears as "VD 64 average" in Figure 3(a). This single execution is enough to see that, when the size is not considered, differences are significant, and the behavior of the virtual disk does not match that of the real one.

6.2. Accuracy for Solid State Drives

For the SSD, and 1, 8 and 32 processes, Figure 3(b) shows the evaluation of the four configurations based on the number of values averaged per cell. The virtual disk behaves very much like the SSD, and differences between both are less than 0.3% on average. Major differences (up to 1.7%) are seen at the beginning of the test, when TAC is run, because the cells are not updated yet to the current behavior of the device, but these differences decrease quickly. Differences among the four tested configurations are negligible. But, to use a single configuration, our disk model will also store the average of the last 64 values per cell when simulating SSDs.

TABLE 1. Average difference (% of I/O time) between the real and virtual disks in the AIR test (REDCAP on, 64 values/cell).

Disk	1 proc	8 procs	32 procs
HDD-400 GB	2.02%	0.49%	0.62%
SSD-160 GB	0.84%	0.44%	0.60%

6.3. Accuracy with REDCAP active

We evaluate our model's accuracy when REDCAP is always active (ADA is not run). The real and virtual disks serve the requests issued by REDCAP for each cache miss. These requests create a workload composed of 128 kB requests that may not be sequential on disk. Table 1 shows the average difference (% in I/O time) between both disks when 64 values are stored per cell. As we can see, our model gives quite good estimations, and differences are smaller than 2.1% and 0.9% for the HDD and SSD, respectively.

7. REDCAP PERFORMANCE

This section studies how KDSim and REDCAP can improve I/O performance together. For this evaluation, the REDCAP cache size is set to 64 MB, 4× larger than the cache of the test disk, and less than 6.25% of memory utilization. The cache has 512 segments of 128 kB each. This segment size showed the best behavior in our early tests [6].

Results are the average of five runs. Error bars represent confidence intervals for the means, for a 95% confidence level. We run the tests with cold (page and REDCAP) caches. In all the executions, the REDCAP initial state is *active*.

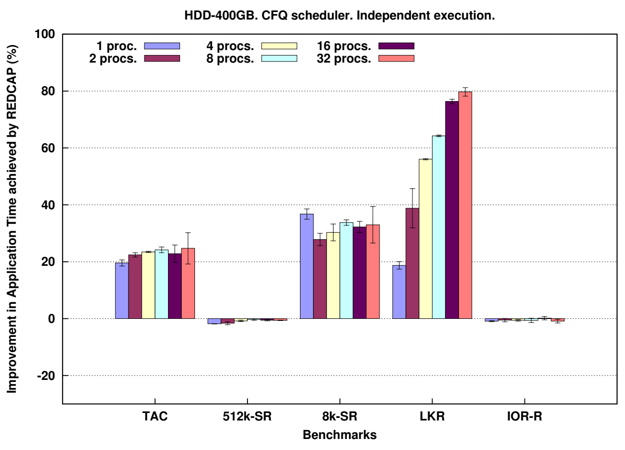
7.1. REDCAP performance with hard disk drives

7.1.1. Benchmarks executed independently

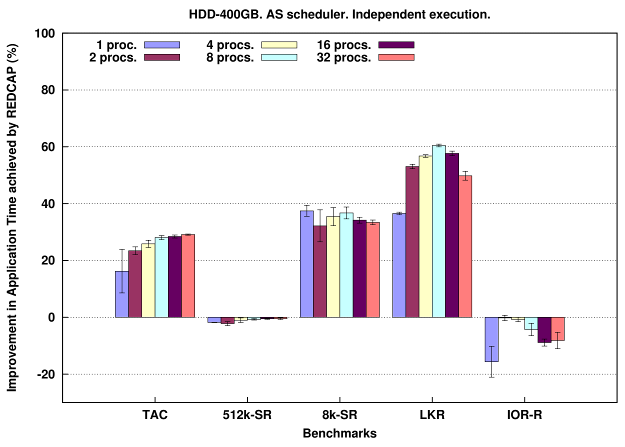
For the tests running independently, Figure 4 shows the improvements in application time achieved by REDCAP with respect to the original kernel, when CFQ/AS and the HDD are used. Tests are sorted in the same order as they are run in *AIR*.

TAC. For this test, REDCAP always performs better than the vanilla kernel with improvements of up to 24.7% for CFQ and 29.1% for AS. The operating system is unable to detect the backward access pattern, and makes no prefetching. REDCAP takes advantage of the immediate read made by the disk drive and of its own prefetching, and its cache is on almost all the time. TAC reads the files with 8 kB requests, so almost fifteen out of every sixteen application requests are cache hits.

512 kB Strided Read. For this pattern, our cache provides no improvement because it is not effective. ADA detects this



(a) CFQ scheduler



(b) AS scheduler

FIGURE 4. Application time improvement achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the hard disk.

fact and keeps REDCAP off almost all the time. Our kernel behaves quite similar to the original one, and, statistically, both have the same performance. The small degradation that appears in a few cases is due to the time initially lost when REDCAP is on at the beginning of the test.

8 kB Strided Read. REDCAP always outperforms the vanilla kernel, and ADA keeps REDCAP always on. The Linux kernel does not detect this access pattern nor does it implement any technique to enhance the performance under this sort of access. With REDCAP, however, most of the requests profit the prefetching done, since almost nine out of every ten application requests are cache hits. Reductions of up to 36.8% and 37.5% are achieved for CFQ and AS, respectively.

Linux Kernel Read. Our proposal always performs better than the original one. REDCAP is active all the time for both schedulers. For CFQ, improvements increase with the number of processes. For 32 processes, the application time is reduced by up to 79.2%. For AS, improvements have roughly the same behavior (except for 1 process), with improvements of up to 60.5%.

Although, REDCAP gets the highest reductions for CFQ, on average the improvement is larger for AS. REDCAP behaves the same for both schedulers, having almost the same hit percentage. Differences are due to the access pattern and the schedulers' behaviors. When traversing a directory tree, AS exploits the spatial locality in disk provided by Ext3 better than CFQ does. AS usually makes fewer disk seeks, since it sorts requests by physical location on disk [23]. CFQ chooses the process to attend with a round robin policy [23, 28]. This implies more disk seeks, increasing application and I/O times. Due to these distinct behaviors, differences between both schedulers are even larger in this test, since a large amount of small files is read.

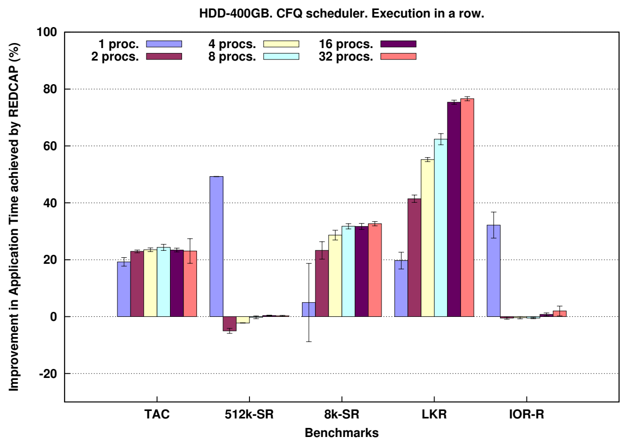
IOR Read. The prefetching techniques of both the operating system and disk caches are optimized for a sequential access pattern. Due to the prefetching of the operating system, most of the read requests issued have a size of 128 kB (maximum request size allowed by the file system), which is the same size as that of the REDCAP requests. Hence, the REDCAP improvement is rather small, and it even adds a copy time to each cache hit. Because I/O times of the real and virtual disks are very similar, sometimes ADA alternates the REDCAP state between active and inactive. But, the right decision would be to keep it inactive.

For CFQ, taking into account confidence intervals, REDCAP behavior is almost equivalent to that of the vanilla kernel.

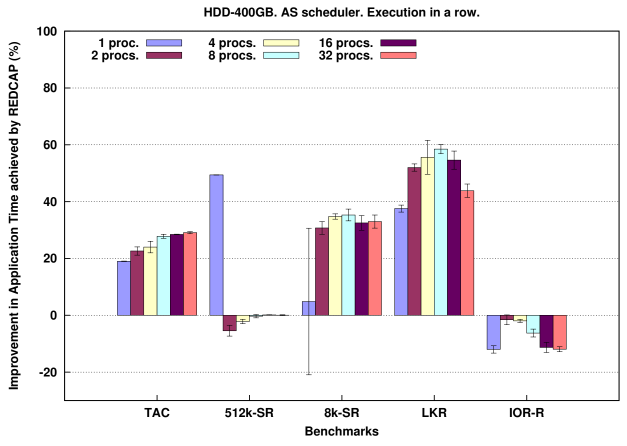
For AS, the situation is slightly different. For 1 process there is a degradation of up to 15.6%. But, if we compare I/O times (not shown due to space restrictions), the degradation is only of 2.6%. To understand these results, we need to explain two aspects of Linux and AS. First, when a new asynchronous request is issued, if there is no pending requests, the kernel plugs the block device, delaying the request that will be done later [29]. The goal is to increase the chances of clustering requests for adjacent blocks. The device usually remains plugged for a small time interval (normally, 3 ms). Second, AS selectively stalls, for a small time interval, the block device right after servicing a request in the hope that a new one for a nearby sector will be soon posted [30]. When a new request is scheduled that meets this condition, if the device is plugged, AS unplugs it.

When REDCAP is on, these two policies produce an odd effect for sequential access patterns. Due to their large sizes, requests usually need two REDCAP segments: one is a cache hit, that copies data, and the other is a miss, that issues a disk read request. Although the copy time is very small, when the disk request arrives to the scheduler, the AS time interval has expired. If there are no more pending requests, the new one is not dispatched until the device is unplugged, what increases application time. The I/O time, however, does not usually increase.

For two or more processes, when the AS time interval expires, a request of a different process is dispatched (because a request of the current process has not been



(a) CFQ scheduler



(b) AS scheduler

FIGURE 5. Application time improvement of REDCAP over a vanilla Linux kernel for the AIR test on the HDD.

issued yet), and the device is rarely plugged. This behavior requires more disk-head movements that can increase seek time, and downgrade the performance of the disk cache because prefetched data can be evicted before being used. For 8, 16 and 32 processes, there is a degradation of 4.3%, 8.9% and 8.2%, respectively. But, for the I/O times, the degradation is only of up to 4.8% for 32 processes.

7.1.2. *All in a row*

Figure 5 shows application time improvements achieved by our technique with respect to the original kernel in this test, for CFQ and AS. REDCAP presents an equivalent behavior to that obtained when the benchmarks are run independently. Improvements of up to 32.7%, 42.2% and 76.6% are achieved for TAC, 8k-SR and LKR, respectively. For 512k-SR and IOR-R, except for two cases, performance is quite similar to that obtained by a vanilla kernel. Hence, according to these results, we can claim that *our proposal adapts very quickly* to the workload changes caused by the execution of the tests in a row. Only minor differences, explained below, are observed due to both the page and REDCAP caches.

1 process. When TAC finishes, a significant amount of file blocks are already in the buffer cache, because the system has 1 GB of RAM, and the file is also 1 GB in size. So, 512k-SR has to read only a small amount of data from the end of the file (due to the backward access pattern of TAC, the last blocks of the file were evicted from memory). As we explained, 512k-SR makes four read series on the file. With the REDCAP-VD kernel, after the first series, almost all the blocks requested by the other three series are either in the buffer cache or in our REDCAP cache. The operating system, however, does not perform any prefetching, and the vanilla kernel has to read all the blocks of the four series by means of independent requests. For this reason, REDCAP unexpectedly gets an improvement of 50%.

When 8k-SR is run, REDCAP reads more blocks than the vanilla kernel and its improvement is reduced to 5%. This is due to the size of the kernel image. The memory needed by the REDCAP-VD kernel image is larger than that needed by the original kernel one. After the execution of the first two tests, with our kernel, there are fewer blocks of the file in memory and more blocks have to be read.

At the end of 8k-SR, 1 out of every 3 blocks (of 4 kB) of the file are in RAM, that is approximately 341 MB of the file. Since the next test, LKR, uses only 344 MB of memory, all the 341 MB of the file are still in RAM when IOR is run. This produces a "strided" access pattern that prevents the operating system from doing large prefetching requests. But, the REDCAP prefetching is used widely, and our method obtains an unexpected improvement of 32% for CFQ.

For AS, IOR-R produces an increase in application time of up to 12%, but an improvement of up to 6.6% in I/O time (not shown). The reason is the same as that given in Section 7.1.1. The effect can also be observed for 8, 16 and 32 processes.

2 processes. At the end of TAC, parts of the files are in RAM, but, due to the size of the kernel images, there are fewer file blocks in RAM with our kernel than with the original one. When 512k-SR is run, REDCAP reads more blocks than the vanilla kernel. Now the blocks to read do not fit in the REDCAP cache, and there is a degradation of up to 5% for both schedulers. For 4 and more processes, the same effect happens, but the degradation is insignificant, since the extra blocks that REDCAP reads represents a small percentage of the total.

7.1.3. *All in parallel*

Results for this test, presented in Table 2, show that REDCAP always outperforms the vanilla kernel, although its improvement depends on the scheduler. REDCAP presents its best behavior for CFQ, reducing the application time by up to 74.3%. In this case, improvements slightly decrease as the number of processes increases, although for 4 processes per benchmark (20 processes altogether) reductions of 60.3% are still achieved. For AS, REDCAP gets improvements of up to 19.3%, and they increase

TABLE 2. Application time improvement of REDCAP over a vanilla kernel for the AIP test on the HDD.

Scheduler	1 proc	2 procs	4 procs
CFQ	74.3%	73.1%	60.3%
AS	3.5%	18.4%	19.3%

with the number of processes. Again, although REDCAP provides the higher reductions for CFQ, its behavior is the same for both schedulers, and the mean percentage of activation is almost equal: 92% for CFQ and 91% for AS. The different reductions are due to the performance of the schedulers in this test. AS already behaves quite well in this test, so REDCAP improvement cannot be higher.

7.1.4. Aged file system

We test our approach in a second HDD with an aged file system. Although results [19] are not showed, they are pretty similar to those provided by the clean file system for all the tests. This means that: i) KDSim is able to simulate HDDs with aged file systems; and ii) aged file systems have no negative impact on REDCAP.

7.2. REDCAP performance with Solid State Drives

Before explaining the results, it is important to clarify two aspects. First, when comparing Linux I/O schedulers on SSDs, Noop and Deadline usually outperform CFQ and AS [26, 27]. The problem is that CFQ and AS insert delays with the hope of minimizing seek times, and the delays can increase application times without reducing I/O times. We have made several tests with the vanilla kernel, CFQ, Noop, and our SSDs, and results confirm that Noop outperforms CFQ with respect to application times, although I/O times are almost identical. The scheduler also affects REDCAP results: improvements in application time are usually larger for Noop than for CFQ, while improvements in I/O time are almost identical.

Second, although the overhead of our proposals is rather small (see Section 7.3), it becomes noticeable with SSDs, and, consequently, application times can be slightly increased, specially in the tests where REDCAP provides no improvement. The “problem” is the high performance of SSDs, and the relative low performance of the computer used, that has a processor with only two cores. But it is important to realize that many current systems have powerful processors with several cores, so we believe that this overhead can be reduced to almost zero in one up-to-date system.

7.2.1. Benchmarks executed independently

Figure 6 shows improvements, in application time, achieved by REDCAP with respect to the vanilla kernel for CFQ and Noop. To explain these results, Figure 6 also depicts improvements in I/O time. The tests are sorted as in the *AIR* test. Results are quite similar to those obtained for the HDD, and only a few differences are noticed, mainly because of the high performance of SSDs.

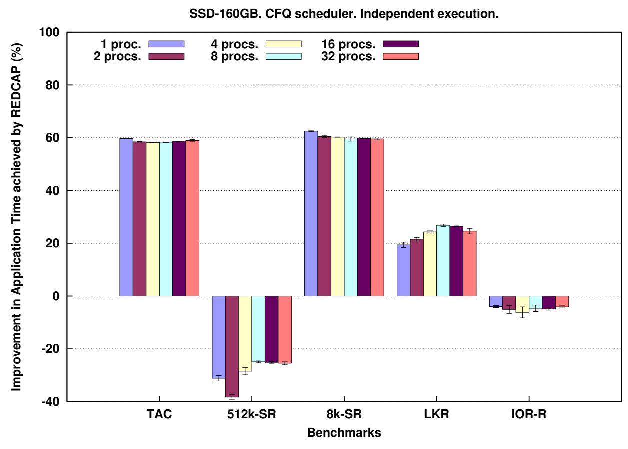
TAC. With this test, REDCAP always performs better than the vanilla kernel, but the improvement depends on the scheduler. The best behavior is achieved for Noop, reducing the application time by up to 80%. Regarding the I/O time, REDCAP gets improvements of up to 80%. Our cache is always active.

512 kB Strided Read. For this test, REDCAP provides no improvement. The algorithm detects this fact and turns it off on the first check, and it is inactive almost all the time. But REDCAP performs worse than a normal system and application time is increased up to 38%, and I/O time up to 22%. This increase is due to two problems. First, it is partially due to the time initially lost by REDCAP while the cache is active at the beginning of the test, and this lost time cannot be recovered in the short duration of the test. Second, the overhead of REDCAP and KDSim, although small, has a negative influence on this benchmark, and the application time is increased. Moreover, with this test, application times are quite small for both the REDCAP and original kernels (bellow 12 s for 1, 2 and 4 processes), and their absolute differences are also small, causing large relative differences. Differences are even smaller when we compare I/O times: the largest one is less than 3 s for 32 processes and CFQ. Table 3 shows average application and I/O times for this test with CFQ. There, we can see the small execution times obtained, the small differences, and that I/O times are almost the same. With Noop, differences are even smaller, but these times are not shown due to space limitations.

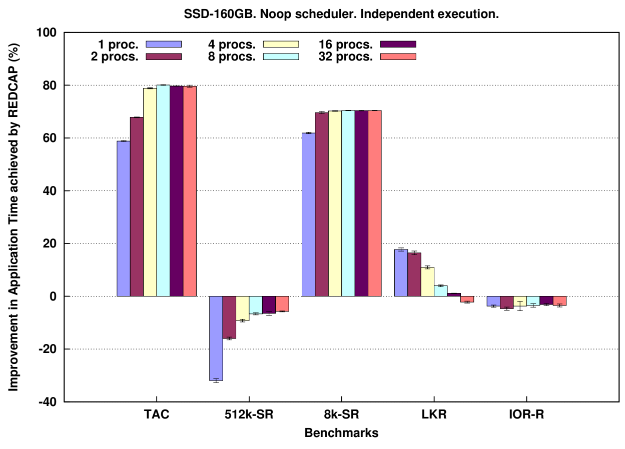
8 kB Strided Read. With this test, REDCAP always performs better than the vanilla kernel. Reductions of up to 62.5% and 70.4% are achieved for CFQ and Noop, respectively. The reason is the same as that given for hard drives.

Linux Kernel Read. When this test is run, the REDCAP cache is almost always active, and it gets almost the maximum possible improvements. Reductions of up to 26.8% and 17.7% are achieved for CFQ and Noop, respectively. But these reductions are not as large as those obtained for hard drives. The problem is the overhead introduced by the creation of the large amount of small processes to read the Linux kernel source, that is relatively larger with respect to the application time on the SSDs than on the HDDs. However, reductions on I/O times are much larger, by up to 74.5%, and they are comparable with those on the hard devices.

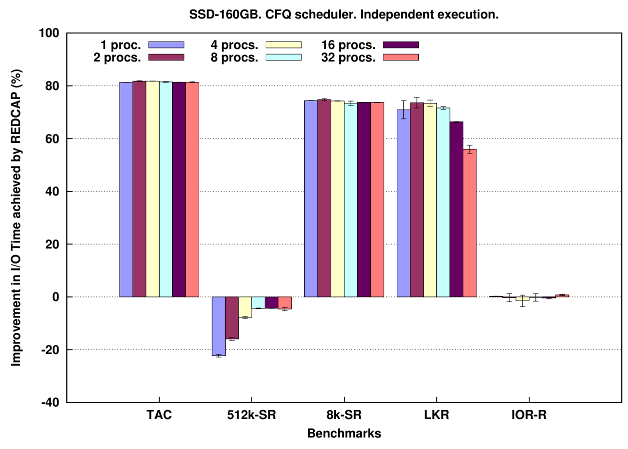
To prove this fact, we modify this test in such a way that a single process now reads all the files of a Linux kernel source tree. We integrate the `cat` command in the `find` command, and, when a regular file is found, the “`cat`” function is called. The reading process is the same, but no processes are created to read the files, and just one makes all the reading. With this test, REDCAP gets improvements in application time of up to 59% and 68% for CFQ and Noop, respectively.



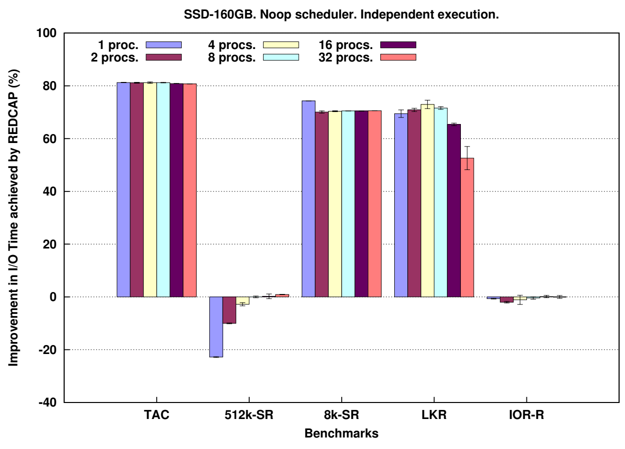
(a) Application time improvement with CFQ



(b) Application time improvement with Noop



(c) I/O time improvement with CFQ



(d) I/O time improvement with Noop

FIGURE 6. Application and I/O time improvement achieved by REDCAP over a vanilla kernel when benchmarks are executed independently on the SSD.

Reductions in I/O time are of up to 77%, quite similar to those of LKR.

IOR Read. With this sequential workload, REDCAP performs worse than a normal system by increasing application time up to 5.6%. The problems are similar to those suffered by 512k-SR: the REDCAP overhead and the increase in the application time produced when REDCAP is active at the beginning of the test. But, since absolute times are larger now, downgrades are relatively smaller than those seen with 512k-SR. For instance, for Noop and 32 processes, the biggest absolute difference is 6.83 s (the times are 163.36 s and 170.19 s with the original and REDCAP-VD kernels, respectively), and the downgrade is just 4.2%. Regarding I/O times, both kernels obtain the same results, and they behave similarly.

7.2.2. *All in a row*

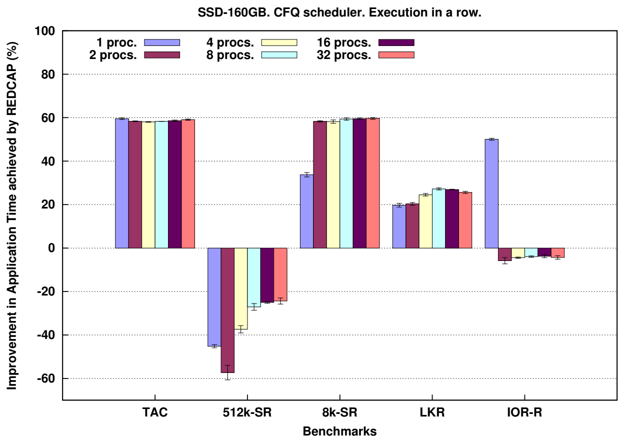
Figure 7 depicts REDCAP improvements in this test. REDCAP presents a similar behavior to that obtained when the tests are run independently. We can claim that KDSim adapts very quickly to the workload changes in this test. Improvements of up to 80%, 71% and 28% are achieved for TAC, 8k-SR and LKR, respectively. With 512k-SR and IOR-R, differences in the application time between REDCAP and a regular system are due to the REDCAP overhead and the small application times, as already explained. There are only minor differences due to the page and REDCAP caches, and the reason is the same as that given for HDDs.

7.2.3. *All in parallel*

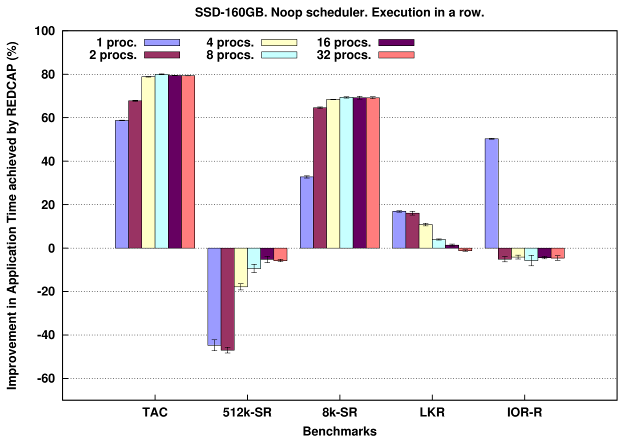
Table 4 presents results for this test. REDCAP always performs better than the regular kernel, although its behavior depends on the scheduler. Its best improvement is achieved for CFQ, reducing the application time by up to 88.7%. In this case, improvements decrease as the number of processes increases, but for 4 processes per benchmark (20 processes altogether) reductions of 63.6% are achieved. For Noop, improvements do not depend on the number of processes, and reach 86.5%.

TABLE 3. Average application time and average I/O time for the 512k-SR test, with the SSD and CFQ.

Average time	Kernel	1 proc	2 procs	4 procs	8 procs	16 procs	32 procs
Application	REDCAP	3.11 s	6.44 s	11.75 s	22.53 s	45.08 s	91.37 s
	Original	2.37 s	4.66 s	9.15 s	18.04 s	36.03 s	72.84 s
I/O	REDCAP	2.31 s	4.42 s	8.21 s	15.84 s	31.62 s	63.44 s
	Original	1.89 s	3.81 s	7.62 s	15.18 s	30.33 s	60.65 s



(a) CFQ scheduler



(b) Noop scheduler

FIGURE 7. Application time improvement of REDCAP over a vanilla kernel for the AIR test on the SSD.

7.3. Overhead of the simulation

We measure our proposal's overhead at two different levels: *whole simulation* and *request copy*. In the former, requests are processed by REDCAP and KDSim as usual, although REDCAP cache is never turned on. In the latter, we only copy regular requests to the disk simulator; we do no run any simulation, although KDSim and REDCAP are loaded. Results for the *whole simulation* show the overhead introduced by both KDSim and REDCAP, i.e., the whole system, whereas results for the *request copy* show the overhead of copying each regular I/O request to our system.

We compute both overheads by comparing the application times with those in the original kernel. We run the AIR test

TABLE 4. Application time improvement of REDCAP over a vanilla kernel for the AIP test on the SSD.

Scheduler	1 proc	2 procs	4 procs
CFQ	88.7%	79.2%	63.6%
Noop	84.3%	83.8%	86.5%

TABLE 5. KDSim and REDCAP overheads.

Overhead	Disk	1 proc	8 procs	32 procs
Whole simulation	HDD	1.6%	2.6%	1.6%
	SSD	7.8%	4.1%	3.5%
Request copy	HDD	0.3%	0.8%	0.4%
	SSD	1.3%	0.2%	0.4%

for 1, 8 and 32 processes. Table 5 presents these 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 2.6% in the worst case). For the SSD, due to its very high performance, this overhead is more noticeable. However, note that, with a more up-to-date hardware, this overhead can also be reduced to almost zero for these drives.

Regarding memory overhead, the KDSim tables need 14 MB for the HDD and only 6 MB for the SSD. The REDCAP cache size is 64 MB (it does not depend on the device). Both mechanisms use only 7.6% (HDD) and 6.8% (SSD) of RAM.

8. RELATED WORK

There have been many studies about disk simulators, and its applications [31, 32, 33]. For brevity, we only highlight those that are representative for our work. The famous DiskSim [9] simulates disks, controllers, drivers, etc. One drawback is that it needs specific parameters of the simulated hard disk, and these parameters cannot be obtained from the disk's technical specification. Another one is that cannot be used for on-line simulation.

Simulators have usually been implemented in user space as standalone applications, or integrated in a more general environment, although, in some cases, they have been implemented in the operating system's kernel. Wang *et al.* [32] implement a disk simulator in the Solaris kernel, but it is specific to a hard disk model, and is not aimed at an on-line performance analysis of a system component.

There are also several simulators for SSDs [10, 11, 21] that usually simulate their internal hardware and software components, and can be used for designing new configurations of SSDs.

Compared to all these simulators, KDSim has the ability to simulate HDDs and SSDs just knowing its capacity. It is integrated inside the Linux kernel and can make on-line simulation and compare different I/O strategies simultaneously.

Regarding disk modeling, we only stand out those that are table-based models. Gotlieb and MacEwen [34] develop a queueing model of disk storage systems to measure performance through mean response times, and use tables to approximate those times. Thornock *et al.* [35] propose a stochastic method that uses tables where each column is a probability distribution of service times for a particular seek distance. For a request, they use the distribution to randomly estimate the corresponding service time. Anderson [36] uses a table-model to measure the performance of disk arrays and to assist in their reconfiguration. By using interpolation, the model returns the maximum estimated throughput available.

Disk Mimic [37] is a table-based disk simulator of HDDs inside the Linux kernel that returns the positioning time of a request. To represent ranges of missing inter-request distances, they use simple linear interpolation. KDSim is similar to Disk Mimic, but with several important differences. Their model predicts positioning time, and only uses the inter-request distance, and the type of the current and previous operations as input parameters. Memory overheads are quite different. They say that, for a disk of 10 GB, the memory needed for their table can exceed 80 MB [37]. So, for a disk of 400 GB, their table needs around 3 GB of memory. Each KDSim table only uses 7 MB of memory. Finally, they propose an I/O scheduler, called shortest-mimicked-time-first, which uses Disk Mimic to select the request with the shortest positioning time, whereas KDSim can evaluate several I/O mechanisms in parallel, and selects that with the highest performance.

The idea of issuing two requests, a "real" and a "virtual", at the same time is somehow similar to the I/O speculation proposed by Chang and Gibson [38], or by Fraser and Chang [39]. But their goal is just to make prefetching more efficient.

With respect to work related to disk caches, or techniques that take advantage of them, we can mention *Disk Caching Disk* (*DCD*), a disk storage architecture that is aimed at optimizing write performance with a small log disk as a secondary disk cache [40]. Their aims are completely different to ours. The log disk buffers modified blocks that will be transfer to disk later, so they optimize writes. Their results are obtained with a trace-driven simulation program.

Several proposals implement storage architectures similar to DCD by using a SSD as a cache for a HDD [41, 42, 43]. These "hybrid" devices improve the performance by usually keeping data associated with random reads and writes on the SSD, and issuing sequential operations on the HDD. Although the main idea (to improve I/O performance) is the same, their aims are completely different to ours. They optimize writes and random workloads, whereas REDCAP optimizes reads and not writes. Furthermore, REDCAP continuously analyzes its performance to decide whether being active or not, while they do not carry out any analysis.

File-Oriented Read-ahead (*FOR*) and *Host-guided Device Caching* (*HDC*) are two management techniques for the disk controller cache [44]. FOR adjusts the number of read-ahead blocks brought into the disk cache according to the file system information. HDC gives the host direct control over part of the disk controller cache. Their drawbacks are: i) their results are based on simulations; ii) to be implemented, the disk controller has to be modified; and iii) they only prefetch blocks that belong to the same file.

Grimsrud *et al.* [45] propose an adaptive prefetching mechanism for disk caches that predicts future access sequences and control the prefetch mechanism by using a table with information about past disk accesses. Zhu *et al.* [46] also use a table with information about the next most probable disk access to implement adaptive prefetching scheme for disk caches. Their results show the effectiveness and efficiency of their proposals. However, the problem of both proposals is that the experiments have been performed in a simulation model, and the proposed algorithms have not been implemented on any machine, whereas REDCAP has been tested and implemented inside the Linux kernel.

Soloviev [47] prefetches disk blocks in RAM by replacing the disk-cache prefetching. Her proposal's drawbacks are: i) it needs to control disk caches, what currently cannot be done; ii) it only tests sequential workloads and prefetching is always active what can downgrade performance in some cases; iii) no control over the achieved improvement is done; and iv) no real implementation exists.

HPCT-IO is an application-based caching and prefetching technique that maintains a file-IO cache in application address space [48]. For a read cache miss, it will issue new requests to read the data; it also tries to make sequential prefetching. Although their cache plays a role similar to that carried out by REDCAP, there are two important differences: i) HPCT-IO is implemented as a user-space library; and ii) it does file prefetching, but no metadata prefetching.

Patterson *et al.* [15] propose that applications inform of future accesses to perform prefetching and caching. They introduce significant improvements, but there are important differences with respect to REDCAP: i) their proposal leverages application hints, so applications should be modified to provide this information; ii) they only prefetch blocks from the same file (and those blocks can be spread across the whole disk); and iii) consequently, they perform file prefetching, but no metadata prefetching.

Operating systems also incorporate some kind of prefetching. Prefetching during sequential access patterns, as Linux does [49, 44], is the most common technique used when files are read. Lei and Duchamp [14] also propose to prefetch entire files by taking into account past file access patterns. Note, however, that both techniques present drawbacks similar to those described for Patterson *et al.*'s proposal.

Smith suggested a disk cache that turned itself on and off depending on the performance [50], but the idea was not tested, and no algorithm was developed to handle the cache state. He also suggested the possibility of using the main memory for the disk cache by mapping files in memory, and *eliminating* the controller cache.

9. CONCLUSIONS AND FUTURE WORK

We have presented the design and implementation of KDSim and REDCAP. KDSim is an in-kernel disk simulator that accurately simulates the behavior of both HDDs and SSDs. Thanks to KDSim, it is possible to simulate different I/O mechanisms, and to dynamically turn them on/off depending on the throughput achieved. REDCAP is a RAM-based disk cache that is able to significantly reduce the I/O time of read requests.

KDSim has interesting features: i) it does not interfere with regular requests; ii) it simulates the service order of the requests in a real disk by considering their dependencies; and iii) unlike other theoretical approaches, *it makes a real self-monitoring and self-adapting I/O subsystem come true*.

REDCAP has also several features that make it unique: i) it is I/O-time efficient; ii) it converts workloads with thousands of small requests into workloads with disk-optimal large sequential requests; iii) its activation-deactivation algorithm makes it dynamic; and iv) it is independent of the device.

By using KDSim, REDCAP decides the right state of its cache to get its maximum improvements. For workloads with some spatial locality, REDCAP achieves improvements up to 80% for HDDs, and up to 88% for SSDs. It presents the same performance as a regular system for random or large sequential workloads.

Our work points out two interesting aspects. First, disk manufacturers should provide specific information of the disk cache behavior to be able to simulate it properly. Second, considering the results achieved by REDCAP, disk drives should include larger caches to enhance their performance.

As future work we want to evaluate our proposals on hardware and software hybrid drives (i.e., HDD+SSD drives), and RAID devices. We also want to improve REDCAP to work more cooperatively with the disk cache by taking into account some of its parameters (number of segments, segment sizes, etc.), and adapting REDCAP to them. Finally, evaluation of our proposal with other file systems, such as Ext4 and Btrfs, is something to be considered too.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] Ousterhout, J. K. (1990) Why Aren't Operating Systems Getting Faster As Fast as Hardware? *Proceedings of the USENIX Summer*, Anaheim, California, USA, June, pp. 247 – 256. USENIX Association, Berkeley.

[2] Bhatia, S., Varki, E., and Merchant, A. (2010) Sequential prefetch cache sizing for maximal hit rate. *Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer Systems*, Miami Beach, FL, USA, 17-19 August, pp. 89 – 98. IEEE Computer Society, Washington.

[3] Shriver, E. (1997) Performance modeling for realistic storage devices. PhD thesis.

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

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

[6] González-Férez, P., Piernas, J., and Cortes, T. (2007) The RAM Enhanced Disk Cache Project (REDCAP). *Proceedings of the IEEE Conference on Mass Storage Systems and Technologies (MSST)*, San Diego, CA, USA, 24-27 September, pp. 251 – 256. IEEE Computer Society, Washington.

[7] González-Férez, P., Piernas, J., and Cortes, T. (2008) Evaluating the Effectiveness of REDCAP to Recover the Locality Missed by Today's Linux Systems. *Proceedings Annual Meeting of the IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems (MASCOTS)*, Baltimore, DF, USA, 8-10 September, pp. 1 – 4. IEEE Computer Society, Washington.

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

[9] (2015). The DiskSim Simulation Environment. http://www.pdl.cmu.edu/DiskSim/. Last accessed: 09-27-2015.

[10] Lee, J., Byun, E., Park, H., Choi, J., Lee, D., and Noh, S. H. (2009) CPS-SIM: configurable and accurate clock precision solid state drive simulator. *Proceedings of the ACM Symposium on Applied Computing*, Honolulu, Hawaii,USA, 9-12 March, pp. 318–325. ACM, New York.

[11] Kim, Y., Tauras, B., Gupta, A., and Urgaonkar, B. (2009) Flashsim: A simulator for nand flash-based solid-state drives. *Proceedings of the 2009 First International Conference on Advances in System Simulation*, Porto, Portugal, 20-25 September, pp. 125–131. IEEE Computer Society, Washington.

[12] Seagate. http://www.seagate.com. Last accessed: 09-27-2015.

[13] Western Digital. http://www.wdc.com. Last accessed: 09-27-2015.

[14] Lei, H. and Duchamp, D. (1997) An Analytical Approach to File Prefetching. *Proceedings of the USENIX Annual Technical Conference*, Anaheim, CA, USA, 6-10 January, pp. 275–288. USENIX Association, Berkeley.

[15] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. (1995) Informed prefetching and caching. *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, USA, 3-6 December, pp. 79–95. ACM, New York.

[16] Card, R., Ts'o, T., and Tweedie, S. (1994) Design and Implementation of the Second Extended Filesystem. *In Proceedings of the First Dutch International Symposium on Linux, also available in http://e2fsprogs.sourceforge.net/ext2intro.html*, Amsterdam, Holland, 8-9 December. State University of Groningen.

[17] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. (1996) Scalability in the XFS File System. *In Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, CA, USA, 22-26 January. USENIX Association, Berkeley.

[18] (2008). JFS for Linux. http://jfs.sourceforge.net/. Last accessed: 09-27-2015.

[19] González-Férez, P., Piernas, J., and Cortes, T. (2010) Simultaneous evaluation of multiple I/O strategies. *Proceedings of the 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC–PAD)*, Petropolis, Brazil, 27-30 October, pp. 183 – 190. IEEE Computer Society, Washington.

[20] Worthington, B. L., Ganger, G. R., Patt, Y. N., and Wilkes, J. (1997) On-Line Extraction of SCSI DISK Drive Parameters (HPL-97-02). Technical report. Hewlett-Packerd Laboratories.

[21] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R. (2008) Design tradeoffs for SSD performance. *Proceedings of the USENIX Annual Technical Conference on Annual Technical Conference*, Boston, Massachusetts, USA, 22-27 June, pp. 57 – 70. USENIX Association, Berkeley.

[22] Kim, J.-H., Jung, D., Kim, J.-S., and Huh, J. (2009) A Methodology for Extracting Performance Parameters in Solid State Disks (SSDs). *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, London, UK, 21-23 September, pp. 1 – 10. IEEE Computer Society, Washington.

[23] Love, R. (2005) *Linux Kernel Development. 2nd edition*. Sams Publishing, Indiana.

[24] Tweedie, S. (1998) Journaling the Linux ext2fs Filesystem. *Proceedings of the 4th Annual Linux Expo, LinuxExpo'98*, Durham, NC, USA, 28-30 June.

[25] IOR Benchmark. http://ior-sio.sourceforge.net. Last accessed: 09-27-2015.

[26] Kim, J., Oh, Y., Kim, E., Choi, J., Lee, D., and Noh, S. H. (2009) Disk Schedulers for Solid State Drivers. *Proceedings of the 7th ACM international conference on Embedded software*, Grenoble, France, 11-16 October, pp. 295 – 304. ACM, New York.

[27] Dunn, M. and Reddy, A. L. N. (2009) A new I/O scheduler for solid state devices (TAMU-ECE-2009-02-3). Technical report. Department of Electrical and Computer Engineering Texas A&M University.

[28] Rodrigues, G. (2008). Variations on fair I/O schedulers. http://lwn.net/Articles/309400/. Last accessed: 09-27-2015.

[29] Bovet, D. P. and Cesati, M. (2005) *Understanding the Linux Kernel, 3rd edition*. O'Really.

[30] Iyer, S. and Druschel, P. (2001) Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. *Proceedings of the Symposium on Operating Systems Principles*, Banff, Canada, 21-24 October, pp. 117 – 130. ACM, New York.

[31] Elford, C. L. and Reed, D. A. (1997) Technology trends and disk array performance. *Parallel and Distributed Computing*, **46**, 136–147.

[32] Wang, R. Y., Anderson, T. E., and Patterson, D. A. (1999) Virtual log based file systems for a programmable disk. *Proceedings of the USENIX third Symposium on Operating systems design and implementation (OSDI)*, New Orleans, LA, USA, 22-25 February, pp. 29 – 43. USENIX Association, Berkeley.

[33] Jiang, S., Ding, X., Chen, F., Tan, E., and Zhang, X. (2005) DULO: An effective buffer cache management scheme to exploit both temporal and spatial locality. *Proceeding of the 4th USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, USA, 13-16 December, pp. 101 – 114. USENIX Association, Berkeley.

[34] Gotlieb, C. C. and MacEwen, G. H. (1973) Performance of movable-head disk storage devices. *Journal of the ACM*, **20**, 604–623.

[35] Thornock, N. C., Tu, X., and Flanagan, J. K. (1997) A stochastic disk I/O simulation technique. *Proceedings of the 29th conference on Winter Simulation*, Atlanta, GA, USA, 7-10 December, pp. 1079 – 1086. IEEE Computer Society, Washington.

[36] Anderson, E. (2001) Simple table-based modeling of storage devices (HPL-SSP-2001-4). Technical report.

[37] Popovici, F. I., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2003) Robust, Portable I/O Scheduling with the Disk Mimic. *Proceedings of the USENIX Annual Technical Conference*, San Antonio, Texas, USA, 9-14 June, pp. 297 – 310. USENIX Association, Berkeley.

[38] Chang, F. and Gibson, G. A. (1999) Automatic I/O hint generation through speculative execution. *Proceedings of the USENIX Symposium on Operating systems design and implementation (OSDI)*, New Orleans, LA, USA, 22-25 February, pp. 1 – 14. USENIX Association, Berkeley.

[39] Fraser, K. and Chang, F. (2003) Operating System I/O Speculation: How two invocations are faster than one. *Proceedings of the USENIX Annual Technical Conference*, San Antonio, Texas, USA, 9-14 June, pp. 325–338. USENIX Association, Berkeley.

[40] Hu, Y. and Yang, Q. (1998) A New Hierarchical Disk Architecture. *IEEE Micro*, **18**, 64–76.

[41] dm-cache. https://www.kernel.org/doc/Documentation/device-mapper/cache.txt. Last accessed: 09-27-2015.

[42] (2015). Flash cache. https://github.com/facebook/flashcache. Last accessed: 09-27-2015.

[43] (2015). Bcache. http://bcache.evilpiepirate.org/. Last accessed: 09-27-2015.

[44] Carrera, E. V. and Bianchini, R. (2004) Improving Disk Throughput in Data-Intensive Servers. *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, Madrid, Spain, 14-18 February, pp. 130 – 141. IEEE Computer Society, Washington.

[45] Grimsrud, K. S., Archibald, J. K., and Nelson, B. E. (1993) Multiple Prefetch Adaptive Disk Caching. *IEEE Transactions Knowledge and Data Engineering*, **5**, 88–103.

[46] Zhu, Q., Gelenbe, E., and Qiao, Y. (2008) Adaptive prefetching algorithm in disk controllers. *Performance Evaluation*, **65**, 382–395.

[47] Soloviev, V. (1996) Prefetching in Segmented Disk Cache for Multi-Disk Systems. *Proceedings of the fourth workshop on I/O in parallel and distributed systems: part of the federated computing research conference*, Philadelphia, PA, USA, 27-27 May, pp. 69–82. ACM, New York.

[48] Seelam, S., Chung, I.-H., Bauer, J., and Wen, H.-F. (2010) Masking I/O latency using application level I/O caching and prefetching on Blue Gene systems. *Proceedings of the IEEE International Symposium on Parallel and Distributed*

Processing (IPDPS), Atlanta, GA, USA, 19-23 April, pp. 1 – 12. IEEE Computer Society, Washington.

[49] The Linux Kernel Archives. http://www.kernel.org/. Last accessed: 09-27-2015.

[50] Smith, A. J. (1985) Disk cache–miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, **3**, 161–203.