Automatic I/O scheduler selection through online workload analysis

Ramon Nou[1], Jacobo Giralt[1], Toni Cortes[1,2]
Barcelona Supercomputing Center (Spain)[1] - Technical University of Catalonia(UPC)[2]
ramon.nou@bsc.es, jacobo.giralt@bsc.es, toni@ac.upc.edu

Abstract

I/O performance is a bottleneck for many workloads. The I/O scheduler plays an important role in it. It is typically configured once by the administrator and there is no selection that suits the system at every time. Every I/O scheduler has a different behavior depending on the workload and the device. We present a method to select automatically the most suitable I/O scheduler for the ongoing workload. This selection is done online, using a workload analysis method with small I/O traces, finding common I/O patterns. Our dynamic mechanism adapts automatically to one of the best schedulers, sometimes achieving improvements on I/O performance for heterogeneous workloads beyond those of any fixed configuration (up to 5%). This technique works with any application and device type (RAID, HDD, SSD), as long as we have a system parameter to tune. It does not need disk simulations or hardware models, which are normally unavailable. We evaluate it in different setups, and with different benchmarks.

***Index Terms*—automatic; I/O Scheduling; pattern matching; optimization;**

I. Introduction

The I/O path comprises a highly developed set of components, to the very least composed by an application, file cache, file system and an I/O scheduler (if we restrict ourselves to the software part). The objective of this architecture is arranging I/O operations in a better way for the underlying storage device. Nonetheless, it is well known that workloads have diverse behavior depending on the storage device [1]. No uniformity can be assumed for a brand and model, not even for a single disk, since aging [2] can produce different results over time. I/O parameters need to change dynamically, even if the workload does not, to maintain performance.

However, many of these I/O parameters either come preconfigured with default values or are tuned only once in the lifetime of a system. In many cases, the default values come from analyses that took place on different setups and it is assumed they will suit the majority.

We studied the I/O scheduler and found out it has a big impact on I/O performance, more with multiple and independent parallel applications, and there is certainly not a single configuration that provides the best performance for every workload nor hardware. A mainline Linux OS implements four different schedulers, with slightly distinct targets: for example, fairness or low latency, and they all underperform for some workloads. The I/O scheduler is assigned statically on a per-disk basis and is normally predefined by the Linux distribution.

On the CPU side, the additional transistors given by the advance of the scale of integration are currently addressed mainly to increase the number of cores in the chip. Performance achieved by CPUs keeps getting higher, and so does the number of CPUs in a system. A similar progression can be found for disk capacity, but not for I/O performance. This performance gap is widening with multicores, since the available storage bandwidth must be shared among several cores. The effects of this performance gap are typically reduced using more storage devices in parallel (like RAID systems), which leads to an increased performance in terms of the number of operations per second (IOPS) and bandwidth.

This paper explores the new opportunity provided by multicores, to add smarter components to the I/O path. Under I/O bound workloads, cores can stand idle during large periods of time, and hence we can get better performance dedicating them to tasks that will alleviate this bottleneck. Therefore, we are proposing a workload analysis mechanism to tune an important I/O parameter: the I/O scheduler. We can use techniques like time-series analysis of I/O traces, to identify previous patterns, unite them to a performance metric, and provide an automatic way to select the best-performing I/O scheduler for the current workload. Patterns are created by the interaction of simultaneous workloads, for example, multiple processes or several Virtual Machines. Some of the samples seem random at first sight, but in fact, they can still be compared and expose a similar behavior to others. The technique we are proposing can link a certain pattern to a performance value, and find if one pattern matches another.

The proposed mechanism should not be used where fairness or low latency is important, specifically real-time systems should avoid using it. In order to link patterns and performance, periodically, we check all different schedulers implying potential inefficiencies for some intervals.

Although the evaluation is done on I/O bound applications, our method runs in low priority threads being able to reduce the impact on CPU bound applications.

In summary, we present the following contributions:

1) A novel method to compare I/O traces and extract behavior patterns online, using an algorithm based on DTW (Dynamic Time Warping [3], FastDTW variant [4]) in multi-cores.
2) The proposal and evaluation of an automatic scheduler

selector implementation for Linux: *IOAnalyzer*.

3) The addition to IOAnalyzer of a simple dynamic probability guided scheduler selector for random workloads, selecting an I/O scheduler when no pattern is detected.

The feasibility of this technique is demonstrated with an evaluation in different setups. We use widely accepted industry benchmarks like TPC-E [5] and TPC-H in a setup with HDD technology. Additionally, we examine multiple workloads that produce a near-random behavior to demonstrate we can obtain good performance where no trivial manual setup is possible. Finally, we evaluate our technique in four Virtual Machines on a 16-SSD RAID-0 running a real application.

II. Motivation

With our method, we assume that similar I/O patterns will be repeated over time. Moreover, a certain pattern is typically followed by another one. The mechanism to manage the dynamic pattern probability chain is further explained in Section III-A.

If we can predict this situation, we can try different system parameters (in our case I/O schedulers). We will be able to see (**learn**) the effects that those system parameters have over the same I/O pattern. Once we have a set of learned values, we can start choosing the most beneficial system parameter for the current workload.

On the next subsections, we will show why clustering I/O patterns is a valid idea and why the I/O scheduler is an important system parameter to tune.

A. I/O Scheduler effects

The I/O scheduler selection affects the performance of the system. Nowadays, Linux distributions implement four different I/O schedulers: Completely Fair Scheduler **(CFQ)**, Deadline (**DL**), Anticipatory (**AS)** and No-Op (**NOOP**). The default one is usually **CFQ** or **DL**. However, we claim no scheduler performs well for every workload and device. In Figure 1, we show two different workloads [1] (row) with two different hardware devices (column), both HDD technology [2]. Each bar is the execution time (normalized to the maximum) when using a static I/O scheduler.

Figure 1 exposes three main ideas: i) looking at the first row (different BENCH, same DISK) we see how the scheduler performance is totally different. Selecting CFQ for BENCH 1 provides us with the best performance, but for BENCH 2 the situation changes and the best scheduler is NOOP. For these workloads on DISK 1, a static selection would not be a good choice. ii) Observing the second column (same BENCH, different DISK) we also see how the best-performing scheduler varies. With the default I/O scheduler selection, be it DL or CFQ, one disk will work in an optimal way while the other will not. iii) Finally, the loss on performance can be significant, for example, at BENCH 1 / DISK 1 we can lose more than an 80% of performance by choosing the wrong scheduler.

[1]workloads are detailed in Section IV-E as **8S** and **512S**

[2]DISK 1 is an 8 HDD RAID-0, and DISK 2 is a single HDD

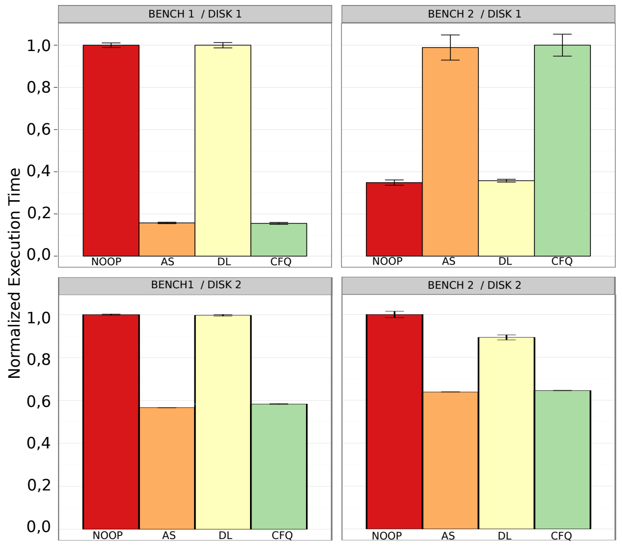


Figure 1. I/O Scheduler comparison between two different benchmarks (row) with two different hardware devices (column). Execution Time is normalized for each test. Each color bar represents a I/O scheduler (NOOP, AS, DL, CFQ).

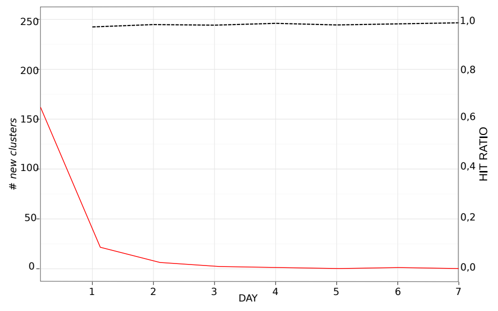
It is worth mentioning that for SSD disks, NOOP scheduler usually performs best. But, eventually a specific one will be created (for example, [6], [7]), which will change this fact. To summarize, I/O Scheduler should be adaptive and not fixed as it is now.

B. I/O clustering inside known traces

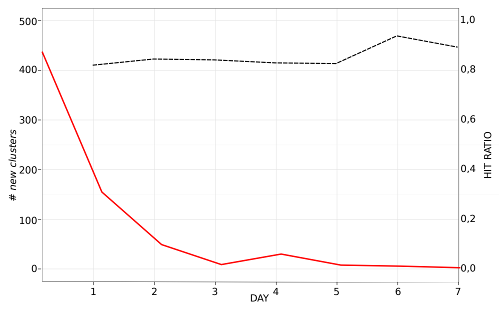
Being able to identify or classify the workload is a difficult task, we cannot use file names, applications or other high level properties. The same application, with identic parameters and files can generate different behavior due to its interaction with other applications or files. Even if we isolate the applications, internal fragmentation or disk aging can convert a simple sequential read in something totally different.

Our methodology uses, mainly, traces obtained from the kernel elevator. These traces include the I/O requests that will be issued to the disks before entering the I/O scheduler. The scheduler will order and merge them, but it is assumed that a similar trace will produce the same result. For our task, we are using Dynamic Time Warping (DTW) [3] (specifically the FastDTW [4] implementation) to compare and get the edit distance between two time-series. In our case, the I/O traces are timeseries. The edit distance ($\mathbb{N}_0$) is the minimal number of changes that need to be applied to a trace to convert it into another one.

We have analyzed two traces, **cello99** [8] and **deasna** [9]. The HP **cello99** trace is taken from the cello server at HP labs in January through December of 1999, is an I/O intensive SCSI-controller-level trace. **deasna** is a trace taken from the Division of Engineering and Applied Sciences at Harvard from 16 October through 22 November of 2002. It is a NFS trace, and includes all the request done to the server. We applied our clustering identification method to see how this



(a) HP cello 99 trace(1 week)



(b) NFS trace (deasna, 1 week)

Figure 2. new clusters (red, solid line) and daily hit ratio (black, dashed line) for 5 seconds traces.

kind of clustering works on those systems. The parameters used for the clustering are as follows: the trace (we only consider traces over 1 IOPS) is cut in 5 seconds pieces. We accept them as similar (hit) if they have a 80% of similarity, and we compare only traces with a difference of a 10% of I/O operations between them. We create a new cluster if the previous conditions are not met (miss). The parameters are the same we are using in the real system.

On Figures 2a, 2b we show the number of new clusters (not random) identified at the traces with a continuous (red) line and the daily hit ratio with a dashed (black) line for one week traces. Figure 2a is from a **cello99** trace. New clusters are found at the beginning of the trace, and they are used over all the period (new clusters do not go higher than 150 and identified clusters, hits, are over 11000 per day). Figure 2b is from **deasna** (NFS). It has a similar behavior; We can find daily values of 400 new clusters and 10000 hits, increasing hits in last trace days. Missing clusters are from periods where no more than one I/O operation per second is done, as they are not analyzed.

To summarize, the I/O patterns found on the first days are used over all the system life, therefore the learning period is small and acceptable.

III. IOAnalyzer

IOAnalyzer is a userspace utility that gathers traces and statistics from different levels of the I/O stack. Traces are compared using DTW, creating a set of clusters with associated performance values from several I/O schedulers. IOAnalyzer then selects the best scheduler (given a metric), for the next probable workload based on historical pattern sequences. IOAnalyzer is designed as a multithreaded low priority process to use the extra cores that we may have idle in the machines and it only runs when I/O is active.

In the next subsections, we will explain in detail all the components of IOAnalyzer (Figure 3 shows a diagram): The workload identification and prediction method, the trace reduction method, the instrumentation needed to achieve our objective, the detailed workflow and finally the parameter selection.

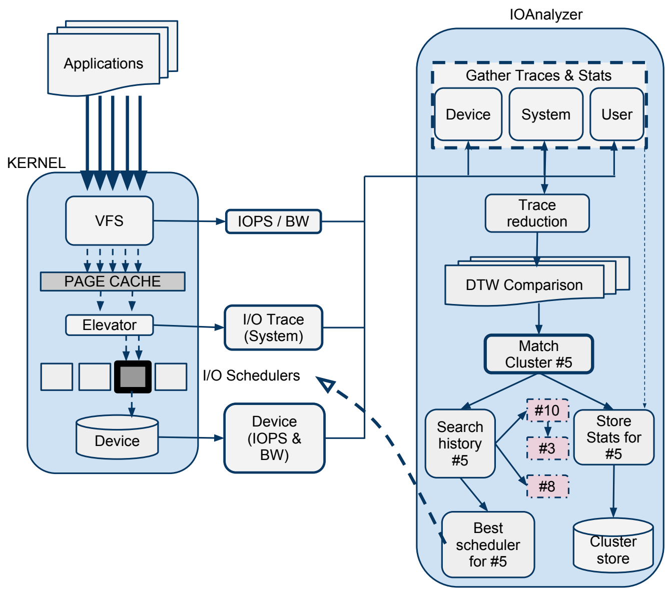


Figure 3. IOAnalyzer and kernel diagram showing a simplified workflow of a cluster match.

A. Workload identification and prediction

The original DTW is $O(N^2)$ (N is the size of the trace), FastDTW provides an $O(N)$ algorithm using a zoom-in recursive process to reduce the matrix to analyze. We are using this linear time multigrid approximation inside our work.

As our technique will be used at runtime, and we need to predict the next workloads, we have to do the clustering more dynamic. Once we have a match of a trace in a stored cluster, we store the found cluster id on the preceding cluster. We use this information to create a ranking of the most beneficial schedulers for the next expected workloads. The ranking is updated continuously to adapt to dynamic workloads. To create this ranking, we define a depth of d clusters and create a dynamic pattern probability chain.

B. Trace reduction

Trace reduction is an optional process. It limits the I/O trace to a specific number of elements. Depending on the target system or device (for example, big RAID system), I/O traces can become very big. Storing them unmodified would result in a huge trace database, increasing the processing time. Reducing the trace moderates the size of the trace database and lowers the CPU usage of the whole process.

Obviously, this reduction needs to maintain the behavior of the original trace. n seconds I/O traces are reduced to x elements or buckets. Each bucket will store the essential information of n/x seconds. We extract the essential information using the maximum and minimum sector jump of the period (Δsector) and randomly[3] select one of them.

C. Kernel instrumentation

Kernel instrumentation is needed to capture I/O operations. We use debugfs/relayfs to get I/O traces before entering the I/O scheduler. The code is added to *elevator.c* modifying the kernel. The I/O traces mainly contain timestamp, sector position, operation (write or read), block size and physical device. We can add an additional instrumentation level at VFS (*read_write.c*) to capture the number of IOPS and bandwidth that the user is committing. We only need the aggregated value for each sample and export it through $/proc$. In userspace, inside the IOAnalyzer, we capture IOPS and bandwidth metrics through the aforementioned interface. This implementation can work in SSD units (16-SSD RAID-0), with workloads up to 150K IOPS. We only had problems with VFS level instrumentation because we were logging every request and aggregating in the analyzer. Therefore we decided to change the implementation to the aggregated values explained beforehand. On general, overhead is not appreciable at tested levels.

D. IOAnalyzer detailed implementation

IOAnalyzer takes traces from the I/O elevator and compares them with DTW using the previous stored clusters (stored traces) and their performance information (for all schedulers, if possible). If a hit is found, we try to guess the next 5–10 seconds of I/O using the created dynamic pattern probability chain. Then we select and change the I/O scheduler. If we have a miss, we store the new cluster and the performance obtained for the actual scheduler in the trace database (this is part of the learning process). In every case, we store in the preceding cluster a pointer to the new found cluster to continue updating the dynamic pattern probability chain. The next part of this subsection describes in detail the implementation.

We have a thread, T_{System}, getting the values from *debugfs* into the user space as $Trace_x$. We also need to classify them per device. Two additional threads, T_{Device} and T_{User}, we also capture the performance (P) in IOPS and bandwidth per second (at system level and user level) obtained at t with the actual scheduler (s) as P_x^s.

In T_{Main}, main process, every t period we get those traces ($Trace_x$) and start reducing them with the algorithm explained at Subsection III-B, this new trace, reduced, is called $CInfo_x$ (now cluster). $CInfo_x$ is compared using FastDTW in the n cores of the system, in low priority, with all $CInfo \in stored$ that have a similar ($\pm 10\%$) number of IOPS. $diff_y$ = minimum (DTW ($CInfo_x$, $CInfo \in stored$)) is found. Finally, $CInfo_x$ and $CInfo_y$ is selected as similar if they have a similar number of IOPS and the $1.0 - diff_y/maxdifference$[4] is higher than a certain value. Our selected value is 0.8, 1.0 means that the clusters need to be 100% equal. In any case, we select always the cluster with minor differences, so this value has effect only on the learning phase when new clusters are being created.

If $CInfo_x$ and $CInfo_y$ are similar we increase the hit count of the detected cluster ($CInfo_y$) and store P_x^s inside it (we hold the last 20 values for each scheduler, to be able to detect changes in performance and overcome a possible overtraining). If they are not, we add a new cluster with the $CInfo_x$ and P_x^s in the $stored$ set.

Once we have a hit, we select the best scheduler that will improve the requested metric for the next expected clusters. The selection is done using information about the most probable following clusters (Section III-A) of the system starting from the detected one. If we do not have enough information about the schedulers of a cluster we try to select one of the remaining untested schedulers. There is a remaining issue, random patterns. A random pattern is detected by our algorithm when we have a miss. In that case, we use a technique called Armed-bandit [10] where the scheduler is selected based on a dynamic probability (for example, NOOP 80%, DL 10%, CFQ 5% and AS 5%). This probability is updated to select more times the schedulers with the better performance while still giving a chance to the other ones to detect environment changes. In this phase, our priority is keeping a good performance during this unknown period. As we will see in Section V, it delivers better results than a static I/O scheduler selection without any cost.

E. General parameter selection

Dynamic probability chain depth d is selectable, but a value of 2 for five seconds traces is enough; it means we are predicting the next 10 seconds. Increasing this value introduces fewer changes in the scheduler, and would make them less responsively. Reducing the value would reduce the prediction capabilities expecting that the next cluster would be similar to the current one. We are using 2 as standard.

The bucket size x is also selectable. We are using 1000 elements for five seconds traces as standard for our evaluation. In order to test the correctness of this reduction, we analyzed the **cello99** traces with and without this feature. We could identify 97% of the clusters for one week trace with the full traces. Enabling the trace reduction, we identify 96% of the clusters. An identification is a hit on the traces' database. Increasing the number of elements increases the hit ratio while decreasing the number of different clusters, but it also increases memory

[3]We inspected several alternative methods (for example, always selecting the maximum jump), but they did not keep as close to the original trace as the proposed method.

[4]*maxdifference* is continuously increased as we cannot know the max difference between clusters of a system. The effect on the first running stages (where we do not know how the system is) is low.

and CPU usage of the matching algorithm. CPU cost (extracted from the evaluation tests, where it uses from 1–2% of CPU) is quadratic respect to the size of the bucket following the next equation: $2.008 \times 10^{-7} \times BucketSize^2 + 8.621 \times 10^{-4} \times BucketSize + 0.25$, memory cost is $(BucketSize \times 8B + 20 * numschedulers \times 4B) \times numClusters$ Bytes.

The I/O analyzer process is executed each t seconds, we are using 5 seconds as standard. Less time will increase CPU consumption each t seconds but we will get a faster scheduler selection boost. Longer periods increase the time needed to test all the schedulers and reduces the benefits as it it more difficult to find a cluster match.

Compared to the IOAnalyzer overhead, a bad scheduler selection could produce a bigger increase on the application execution time. However our mechanism uses low priority threads to not affect CPU bound workloads. One of the most common problems of learning techniques is overfitting (not observed in the evaluation). Anyway, to mantain the system controllable, we have a mechanism to limit the number of clusters (6000 in our evaluation) in the database and remove the old ones (following an LRU algorithm).

Finally, the selection of the values used on our evaluation (traces of five seconds, similarity of 0.8, size of the reduced clusters) has been found experimentaly. Further analysis of more parameters combinations are left out of the paper, as they are related to the available CPU power of the target machine and the expected I/O load.

IV. Evaluation

In this section, we will evaluate our mechanism with different tests in different setups. We will show the results for each scheduler (in an unmodified Linux system) and then the results for our dynamic method.

Our main environment uses a Intel Core 2 Quad CPU Q9300 with 4 GB and a standard SATA drive ST31500341AS. The operating system is a Linux Ubuntu 10.04.1 LTS with a 2.6.32 kernel, without modifications (only the minimal to extract I/O traces). We are targeting bandwidth on the system side (in the elevator) as metric to improve. For subsection IV-D we are using a different machine, an HP ProLiant DL380 G6 Special Rack Server, with 36 GB of memory, 8 cores and a LSI 9260-16i with 16 SDD in RAID-0 to run 4 Virtual Machines and a real application.

Each of the evaluations is done starting with a clean system, i.e., not trained. Page cache is also cleaned. Results are gathered when they are stable, this is when they do not go down the second best scheduler. In the worst case (2LKR+2HD, Section IV-C) this happens on the 15th iteration (10 hours) of the test. In TPC-E (Section IV-A), on the contrary, in the 5th iteration (less than one hour), the results are steady.

A. Industry Benchmarks - TPC-E - Highly predictable workload

This test uses a TPC-E [5] database simulating an eBay/Amazon site. The parameters are the next: 4000 clients, 80 trade days, scale factor 500 receiving the load from 10 clients in 1000 seconds. The test has a ramp up time of 100 seconds. Database is running on MySQL 5.1.41. Figure 4a shows the improvement (%) on transactions per second (TPC-E metric) over the NOOP scheduler when using different schedulers (DL, AS, and CFQ) and with our method (**DYN**). Boxes represent the median, first and third quartiles, and the maximum and minimum value with whiskers. CFQ is the best scheduler in this case, with a performance over an 18% better than NOOP, which is the worst one for this test. Nevertheless, the results obtained with our dynamic implementation are better than those of CFQ (near 2.5%). As we have an heterogeneous workload, dynamically adapting the scheduler used in different phases can improve performance beyond any fixed configuration.

B. Industry Benchmarks - TPC-H with parallel queries

The second test is a TPC-H [5] 10Gb database (over MySQL 5.1.41) running a set of queries[5] in parallel (queries 1, 3, 5, 6, 7, 11, 12 and 19). TPC-H simulates a Decision Support System or Business Intelligence database environment.

In Figure 4b we have the time obtained with the four static schedulers and with the dynamic method. In this case, the best scheduler (DL) is different from that of the previous TPC-E test (CFQ, the second worse in TPC-H). We can make two separate groups of schedulers with a big difference in performance between them, AS and CFQ as the worse ones, and NOOP and DL as the most suitable for the TPC-H test. Finally, we can observe how the dynamic method is able to obtain a performance between those two best schedulers.

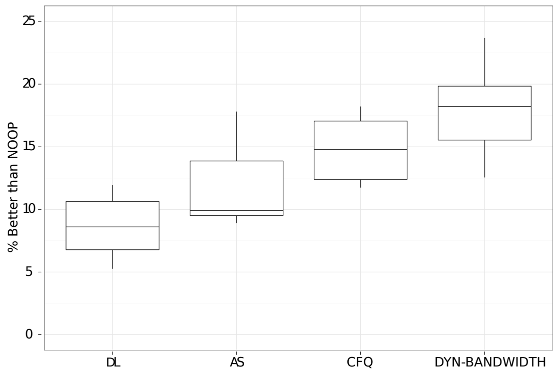
A more detailed analysis of the bandwidth obtained by every pure scheduler shows that the schedulers receiving more bandwidth are not always the better ones on this test, TPC-H is affected by the page cache. This effect is produced by the page cache pollution and trashing, pages useful for the next queries are removed sooner due to the workload. The page cache can impact greatly the results observed by the user (25.74%). This effect is also observed in [11]. In Section V we will propose a solution for this kind of workloads that can improve the results with our dynamic scheduler.

C. Apparently random workload from multiple sequential applications

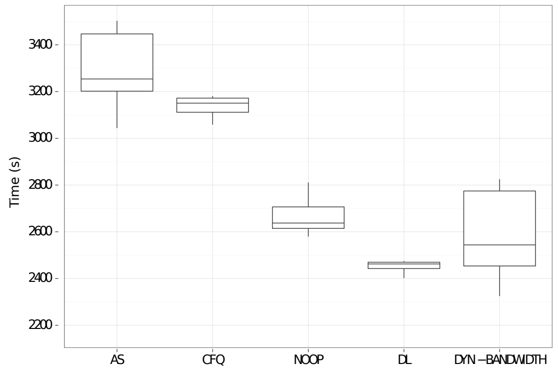
In this third test, we will try an apparently random workload. We are using two components: LKR (Linux Kernel Read) and HD (Hexdump). LKR reads the kernel source files. Each instance of the LKR test traverses the kernel source tree six times. Each of them from a different directory to avoid the cache (device buffer and page cache). The second component, HD, is reading sequentially from a set of files of 1 GB each. An instance of HD reads four randomly selected files of this set. Finally, the test consists on running two instances of LKR and two instances of HD in parallel.

As LKR and HD are running in parallel (In an isolated environment, one LKR runs in 184 seconds, one HD process

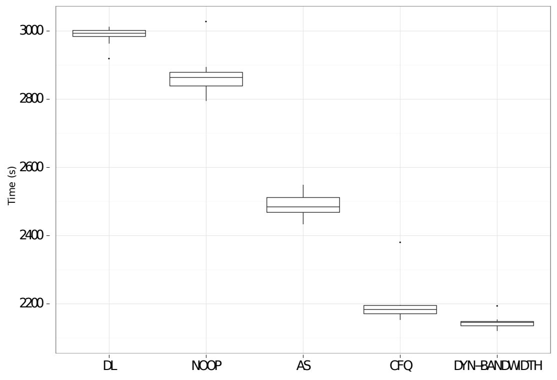
[5]The subset of queries is selected based on the running time in the tested machine to have a 1 hour test



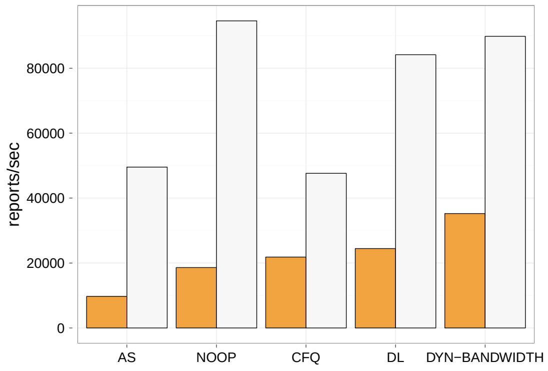
(a) TPC-E : scheduler improvement wrt. NOOP.



(b) TPC-H



(c) 2 LKR+2 HD



(d) Tariff Advisor, QCOW2(filled)/RAW(white) image format

Figure 4. Effect over different workloads of a I/O scheduler selection (static - dynamic).

in 850 seconds), the requests to the disk are apparently randomized. This test, using a fixed policy, gets the minimum time using CFQ. However, the execution time can be improved by nearly a 3% if we apply our dynamic scheduler. Figure 4c has the results for each scheduler. Our dynamic scheduler can get execution times near the best selectable scheduler, CFQ, that was the second worst in the previous TPC-H test. Moreover, DL, the best of that test, is the worst in this one. Nevertheless, we decrease the best time and as a side effect, reduce the variability of the test.

D. 16-SSD RAID-0 with 4 Virtual Machines running a real application

The real application for this test is Tariff Advisor [12] from Neurocom. It is an application based on a fast rating/re-rating engine for telecom operators that is trying to compute and analyze what would be the revenue generated by costumers if they bought a different tariff. It can perform rating and re-rating of call data, for various domains, such as mobile, fixed, and VoIP telecommunications. It has already been deployed to major telecom operators in Greece and other countries. We run 4 Virtual Machines images of 80 Gb, using two different image formats: QCOW2 [13] and RAW. QCOW2 implements copy-on-write and is an image used mainly to static VMs, RAW is an image format representing a real hard disk. We selected this two image formats as we detected that they have a different behaviour under SSD. RAW offers the best performance (twice than QCOW2) and it should be used normally. In Figure 4d we can find the results obtained with the static schedulers and with IOAnalyzer. Results are the accumulated reports per second, the metric that the application uses, over all the VMs. We can see, how in the QCOW2 image format test, 3 static schedulers obtain best performance than AS. RAW image format has a totally different behaviour (with the same application and data sets). We can see how in RAW we have DL, the best scheduler in QCOW2, as the third one in performance. However, IOAnalyzer is able to obtain more performance in QCOW2 and stay closer the first and second best scheduler in RAW.

E. Adaptability - Running different workloads in a row

Our last test shows the results obtained from running a set of benchmarks continuously, without stopping IOAnalyzer. Our objective is to demonstrate that IOAnalyzer adapts the I/O scheduler to the changes in the workload (not only inside the same test) and does not suffer from overfitting. We run the following tests with 4 and 8 processes: 8S and 512S, they read 4KB and jump 8 KB or 512 KB, respectively, of a 2GB file. These tests use a different 2GB file for each process and the page cache is cleaned every time. Finally, between the 4 and 8 processes tests, we run the **TPC-H** benchmark. Figure 5 shows the five consecutive tests and their normalized execution times

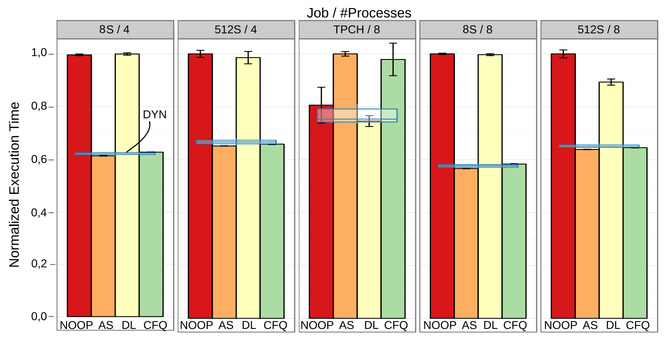


Figure 5. Sequential running test using **8S**, **512S** with different number of processes (4 and 8) and **TPC-H**. Bars are the results from pure schedulers with a confidence interval of 95%. Horizontal box showing the results (max, min, median) using the dynamic scheduler. Executions times for each individual test are normalized to the worst one. One HDD used (ST31500341AS).

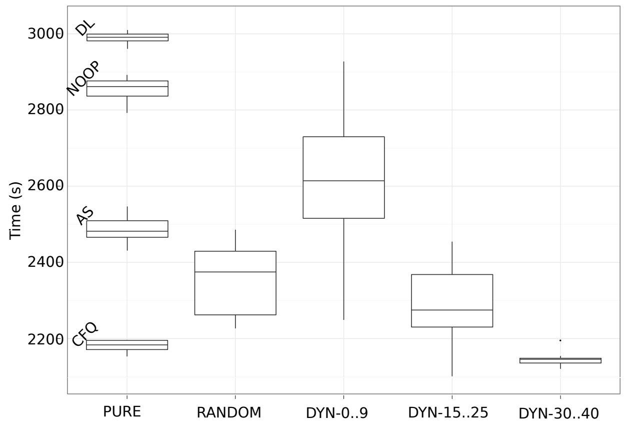


Figure 6. 2 LKR+2 HD execution times with static schedulers and with dynamic changing schedulers. Pure random with dynamic probability, and dynamic selection using several training iterations.

(per test) for each pure scheduler. For every test but TPC-H the best scheduler is AS or CFQ. For TPC-H the best one is DL as we have seen in Section IV-B. Finally, we have a horizontal bar with the result for our dynamic method. Our proposal is closer to the best scheduler in every case. Moreover, we do not have a performance impact due to the change of workload.

V. Improvements

Training

One of the main concerns with this kind of techniques is the learning period, in this subsection we will show how our method can achieve good results with few iterations. We will use the LKR+HD workload from Section IV-C, as it is one of the most random workloads that we have evaluated, and we will see how the training converges fast to optimal values. On the other hand, once the clusters are found (by continuously running IOAnalyzer) they can be used on other workloads.

In Figure 6 we have the previous result from all the pure schedulers, the RANDOM result using only the Armed-Bandit method (Section III-D) and finally, the results of using our dynamic method in different iterations: 0–9, 15–25 and 30–40 (The results improve each iteration, but we plotted those ranges to simplify). We get close to the best scheduler in approximately 15 test iterations ($\approx$ 10 hours). Finally, on more standard workloads, stabilization happens sooner and most of the clusters will be learned beforehand ($\approx$ 1 hour with TPC-E), as it is designed to be continuously running. Being in the learning period, does not mean that the performance will be worse than the worst scheduler. Typically, performance in the learning phase is around the mean of all the involved schedulers.

Finally, the Armed-Bandit method obtains good results without CPU-usage and traces gathering. This happens because we do not force to try the schedulers that are worse than others as our priority is keeping a good performance during this unknown period. On the other hand, we still need to try sometimes the other schedulers to detect changes in the workload, choosing eventually a bad scheduler for a short period. If we combine this armed-bandit method with our dynamic method, when we detect a known cluster (no random), we start again selecting the best scheduler. We get a performance boost for that case, which is better than using Armed bandit during all the workload. However, this armed-bandit technique by itself is simple enough to be a candidate to include it as a user-level application in any Linux distribution. Using the random technique along with the cluster identification surpasses the performance of the previous technique in 15 test iterations (Figure 6).

Extending IOAnalyzer with Application metrics

As we introduced earlier, TPC-H test uses the page cache intensively as many queries request the same data. This means that the best scheduler, not always increases application performance due to trashing. For this case, we can extend the captured metrics to include application level indicators and try to optimize them. In TPC-H, as the problem was that page cache pollution reduced performance, we tested this extension adding the concept of page cache $HITS$ as a metric. We composed a new metric using two already available ones, user side bandwidth minus system bandwidth and tried to optimize it. However, direct performance indicators extracted from the application, if available, could be applied here to drive the scheduler switches. System administrators could also define their own metrics and target them from our method. With this modification, the performance obtained was near the DL values (Figure 4b) (with similar stability).

VI. Related Work

There have been different approaches to automatically select an I/O scheduler. First, we can find a similar technique using machine learning [14]. The paper reviews different methods (from random selection with feedback to machine learning) and provides results using IOMeter [15] synthetic workloads and real ones. However, the workload is stored and compared using simple characteristics like its proportion of reads and writes, average request size or sequential/random ratio, which can be not enough to describe some workloads. For example, their mechanism could not differentiate a pure random workload from an inverse sequential one and change the scheduler if there is any benefit to do it. *Our proposal maps all the disk requests on a 2D space (time and position), detecting those cases where similar quantitative values are different workloads.* Another related mechanism [16] monitors latency and bandwidth. This paper tries to change between deadline and CFQ I/O schedulers to optimize one of those metrics. Our method, on the contrary, is not limited to a subset of schedulers (any old or future ones); we test every present scheduler. Also we allow any metric to be selected as optimization target. However, the proposed method [16] could be applied to a real-time system. Our approach is limited to non-critical systems because of the learning periods. Methods using online simulation as [17] need to build the disk simulator (virtual disk) and modify the I/O schedulers. They do not need learning periods, but their capabilities are reduced as device complexity increases. *Our method will still work unmodified with any future scheduler created for SSD disks (for example, [6], [7]) or any new device (simple or complex).* Also slightly related, C-Miner [18] and DiskSeen [11] look for block correlations and sequences using them for data prefetching. Their proposal can be used in complement with our mechanism as they focus on a different component of the I/O stack. In fact, data prefetching can be faster if the correct I/O scheduler is selected. *Our method can work with any proposal targetting higher layers (over the I/O scheduler) like prefetching.*

VII. Conclusions and Future Work

We have shown that different I/O schedulers result in different performance levels for an application and system, and that there is not an optimal one. We have then proposed a mechanism to detect and compare workload patterns, which we relate to I/O performance with the different schedulers. With that information we constructed IOAnalyzer, a tool to automatically select the best I/O scheduler for the detected workload. The evaluation demonstrates our method always yields **the best or near best performance** for any workload. Moreover, often, we can get better performance than any fixed scheduler for a wide selection of tests (heterogeneous workloads). A key point of this proposal is that it will work on any kind of hardware (SSD, HDD) or organization (RAID, NAS, etc.). It is a safe choice by the system administrator even if the workload is not clear at the point of configuration. This implementation is working in Linux with a modified kernel. Alternatively, we have a naive random approach, which can be easily implemented as it does not need kernel instrumentation, and still will give a better (Figure 6), more adaptive performance to the I/O scheduler than leaving it fixed for many workloads. Future work goes in several directions, mainly adapting the computing power dedicated to I/O with respect to the system load. This will be done using a separate scheduling class which will restrict our analysis to those periods with idle CPUs and not interfere with the CPU intensive workloads. I/O path improvements are targeted to I/O bound applications, but typically we will find different phases in an application. Additionally, we can also *off-load* these tasks or part of them to the GPUs [19] and reduce the CPU usage. A second action is to deeply investigate the metric to optimize. The presented work uses IOPS and bandwidth; other systems may need to optimize, for example, fairness, even application metrics can be used. Finally, each I/O scheduler has a big number of parameters, and they can affect performance. This might be specially interesting when using IOAnalyzer with virtualization ([20]). However, expanding the parameter space, increases the time to test every possible combination.

VIII. Acknowledgments

This work was partially supported by the EU IST program as part of the IOLanes project (contract FP7-248615), by the Spanish Ministry of Science and Technology under the TIN2007-60625 grant, and by the Catalan Government under the 2009-SGR-980 grant. We would also like to thank Neurocom for letting us use their TariffAdvisor application. Additional information about IOLanes project can be found in the web [21].

References

[1] E. Krevat, J. Tucek, and G. R. Ganger, “Disks Are Like Snowflakes: No Two Are Alike,” *13th W. on Hot Topics in Operating Systems*, 2011.

[2] E. Pinheiro, W.-D. Weber, and L. A. Barroso, “Failure trends in a large disk drive population,” in *5th USENIX conference on File and Storage Technologies*, 2007.

[3] R. Bellman and R. Kalaba, “On adaptive control processes,” *Automatic Control, IRE Transactions on, vol. 4, no. 2, 1959.*

[4] S. Salvador and P. Chan, “Toward accurate dynamic time warping in linear time and space,” *Intelligent Data Analysis, 2007.*

[5] TPC - Transaction Processing Performance Council. http://www.tpc.org.

[6] M. Dunn and A. Reddy, “A new I/O scheduler for solid state devices,” *Department of Electrical and Computer Engineering Texas A&M University, Tech. Rep. TAMU-ECE-2009-02-3*, 2009.

[7] S. Li. an IOPS-based I/O scheduler. http://lwn.net/Articles/474268/.

[8] HP Labs. (1999) Tools and Traces. http://www.hpl.hp.com/research/ssp/software.

[9] Harvard. (2002) NFS Traces. http://www.eecs.harvard.edu/sos.

[10] D. A. Berry and B. Fristedt, “Bandit problems: Sequential allocation of experiments,” *Monographs on Statistics and Applied Probability, 1985.*

[11] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, “DiskSeen: exploiting disk layout and access history to enhance I/O prefetch,” in *2007 USENIX Annual Technical Conference*, 2007.

[12] Neurocom. TariffAdvisor / TariffSuite. http://www.tariffsuite.com.

[13] M. McLoughlin. (2008) QCOW2 Image format. http://people.gnome.org/~markmc/qcow-image-format.html.

[14] Y. Zhang and B. Bhargava, “Self-Learning Disk Scheduling,” *IEEE T. on Knowledge and Data Engineering*, vol. 21, pp. 50–65, 2009.

[15] Intel, “Iometer User’s Guide,” http://sourceforge.net/projects/iometer.

[16] S. R. Seelam, J. Babu, and P. J. Teller, “Automatic I/O Scheduler Selection for Latency and Bandwidth Optimization,” *W. on Operating System Interference in High Performance Applications*, September 2005.

[17] P. González-Férez, J. Piernas, and T. Cortes, “Simultaneous Evaluation of Multiple I/O Strategies,” *22nd International Symposium on Computer Architecture and High Performance Computing*, 2010.

[18] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou, “C-Miner: Mining Block Correlations in Storage Systems,” in *3rd USENIX Conference on File and Storage Technologies*, 2004.

[19] G. Poli, J. F. Mari, J. H. Saito, and A. L. M. Levada, “Voice Command Recognition with DTW using GPU with CUDA,” *Computer Architecture and High Performance Computing*, pp. 19–25, 2007.

[20] D. Boutcher and A. Chandra, “Does virtualization make disk scheduling passè?” *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 20–24, March 2010.

[21] FORTH, UPM, BSC, IBM, INTEL, and NEUROCOM. (2010) IOLanes - Advancing the Scalability and Performance of I/O Subsystems in Multicore Platforms. http://www.iolanes.eu.