# A QoS Aware Non-work-conserving Disk Scheduler

Pedro Eugênio Rocha
*Federal University of Paraná, Brazil*
*pedro@inf.ufpr.br*

Luis C. E. Bona
*Federal University of Paraná, Brazil*
*bona@inf.ufpr.br*

***Abstract*—Disk schedulers should provide QoS guarantees to applications, thus sharing proportionally the storage resource and enforcing performance isolation. Disk schedulers must execute requests in an efficient order though, preventing poor disk usage. Non-work-conserving disk schedulers help to increase disk throughput by predicting future requests' arrival and therefore exploiting disk spatial locality. Previous work are limited to either provide QoS guarantees or exploit disk spatial locality. In this paper, we propose a new non-work-conserving disk scheduler called High-throughput Token Bucket Scheduler (HTBS), which can provide both QoS guarantees and high throughput by (a) assigning tags to requests in a fair queuing-like fashion and (b) predicting future requests' arrival. We show through experiments with our Linux Kernel implementation that HTBS outperforms previous QoS aware work-conserving disk schedulers throughput as well as provides tight QoS guarantees, unlike other non-work-conserving algorithms.**

## I. Introduction

Storage consolidation in dedicated servers is a growing approach in organizational and departmental data management. This consolidation brings several benefits, such as ease of management and backup, optimized hardware usage and flexibility in the storage capacity allocation. In order to share a centralized storage resource, a *virtual disk* abstraction must be provided, which can be expressed in terms of capacity and QoS guarantees. This approach is specially interesting in virtualized environments, where the underlying hardware must be multiplexed among VMs while providing performance guarantees.

A very common approach when providing QoS guarantees is to modify the disk scheduler [4], [6], [8], [11]. QoS aware disk schedulers are usually based on fair queuing algorithms, initially used for packet scheduling and then adapted to the disk scheduling context. In a fair queuing algorithm, each request receives one or more *tags*[1] according to its QoS guarantees, defined in terms of bandwidth, latency and bursts. In fact, providing QoS guarantees enforces the so-called *performance isolation*, in which the performance experienced by an application should not suffer due to variations in the workload from other applications [10].

Nevertheless, is it also necessary that the scheduler executes requests in an efficient order, thus increasing disk throughput. Non-work-conserving disk schedulers, which are widely used in current systems, like Anticipatory [7] and Completely Fair Queuing (CFQ) [1], are intended to increase disk performance through future request prediction [12]. The key idea behind these schedulers is that a request that is soon to arrive might be closer to the current disk head position than other pending requests. If the seek time needed to serve other pending requests is greater then the cost of keeping the disk idle while waiting for future requests (assuming that such request *does* arrive), than the idle waiting is justified.

[1]Tags are timestamps assigned per-request (or per-packet) based on either virtual or real time.

This performance gain is commonly observed when the scheduler must deal with concurrent applications issuing synchronous requests. In the meanwhile between synchronous requests, a work-conserving scheduler would serve a pending request from another application, thus losing spatial locality. This behavior, commonly known as *deceptive idleness* [7], causes unnecessary seek time and harms disk performance. As non-work-conserving schedulers consider both pending and future requests when taking schedule decisions, then several synchronous requests issued by the same application can be dispatched sequentially, increasing disk throughput.

This paper presents a new non-work-conserving disk scheduler algorithm called High-throughput Token Bucket Scheduler (HTBS). As far as we know, previous work were focused on either providing QoS guarantees [4], [6] or high throughput [1], [7], [11]. HTBS aims to ensure both. Our algorithm can provide QoS guarantees, defined in terms of bandwidth, latency and bursts, by assigning tags to requests in a fair queuing-like fashion. In addition, HTBS schedules future requests as well, ensuring high performance.

We implemented the HTBS scheduler as a module for Linux Kernel 2.6.38. Through experiments, we show that HTBS outperforms previous work in two dimensions: (a) achieving higher performance than former QoS aware work-conserving schedulers, and (b) still providing QoS guarantees to applications with different attributes even in the presence of synchronous requests.

The rest of this paper is organized as follows. Section II points to related work. Section III describes the HTBS disk scheduler algorithm. Our Linux implementation and experiments are presented in Section IV. Finally, Section V concludes this paper.

## II. Related Work

Prior effort in disk scheduler development can be classified into two major groups: high-throughput schedulers and QoS aware schedulers. High-throughput schedulers are generally non-work-conserving, i.e., they do predict future requests, in order to increase the disk performance. On the other hand, QoS aware disk schedulers do not implement future request

prediction as they are usually work-conserving, what can lead to poor performance.

Iyer *et al.* [7] introduced the concept of future requests scheduling. In their work, they proposed an Anticipation framework that can be placed on top of others disk schedulers. However, their analysis covers only proportional-share schedulers (YFQ [4], in particular). Proportional-share schedulers are limited since one cannot configure bandwidth and latency independently. Therefore, if a flow has higher bandwidth guarantees, it will necessarily have lower latency than lower bandwidth flows, differently from our algorithm.

Another algorithm that implements future request prediction is the Completely Fair Queuing (CFQ) [1], which is the default disk scheduler used by most Linux distributions. CFQ distributes time slices to applications, similarly to CPU scheduling, based on processes' I/O priority. CFQ is not able to provide QoS guarantees though.

The Budget Fair Queuing (BFQ) [11] predicts future requests in a similar fashion as the Anticipatory Scheduler, using a proportional-share algorithm to assign tags and to control the virtual clock. The main differences between our work and BFQ are: (a) BFQ assigns tags per-application, not per-request (or per-packet), like modern fair queuing algorithms; (b) BFQ does not provide an explicit way to configure per-application bandwidth, e.g., in kilobytes per second or I/Os per second, but just a weight that relies on disk performance, which is very difficult to predict; and (c) lack of support for bursts and delay configuration per-application.

pClock [6] is a disk scheduler based on former fair queuing algorithms [2], [3], [5], [9]. In pClock, tag assignment policies were extended thereby allowing the configuration of QoS guarantees in terms of bandwidth, delay and bursts. However, request dispatching order in pClock is solely based on per-request tags, resulting in low spatial locality and consequently poor disk performance. Besides, pClock is a work-conserving scheduler since it does not predict future requests.

## III. High-throughput Token Bucket Scheduler

In this Section, we detail the High-throughput Token Bucket Scheduler (HTBS). HTBS uses a tag assignment policy that is similar to others fair queuing-based schedulers [6], since they provide good capacity allocation, with a modified dispatch order. The modified dispatch order from HTBS prevents deceptive idleness by scheduling future requests, like other non-work-conserving algorithms.

### A. Algorithm Specification

Like any fair-queuing algorithm, HTBS relies on timestamps, also called *tags* [4], [6], [11]. For each request two tags are assigned: start tag $S_i^j$ and finish tag $F_i^j$, where $i$ represents the application and $j$ is the request identifier. Moreover, there is one per-application tag, named $MaxS_i$, which represents the biggest start tag between the pending requests from $i$, used to compute new requests' tags.

An application (which could represent a real application, process groups, threads or even entire virtual machines) consists of a request queue and three performance attributes, $\sigma_i$, $\rho_i$ and $\delta_i$, representing bursts, bandwidth and delay, respectively. We also define the *backlog* of an application $i$, namely $B_i$, as the number of pending requests in a given queue. An application is said *backlogged* if $B_i > 0$. Finally, the application that is currently receiving service from the disk is named *active application*.

There are also two parameters in HTBS: $B_{max}$ and $T_{wait}$. $B_{max}$ controls the maximum number of requests from the same application the scheduler can issue consecutively. This parameter avoids starvation of requests issued by other applications. Moreover, $T_{wait}$ limits the time the disk can be kept idle when waiting for future requests. $T_{wait}$ must be equal to the smallest period of time necessary for an application to handle the completion of a prior request, process it and then issue the next one.

The HTBS main algorithm is shown in pseudo-code in Figure 1. The function *dispatch_request* (line 7) returns the next request to be served. Basically, it performs two actions: selecting the active application (the application whose requests will be served), and dispatching one of its requests. When the active application is not set, the function will search for the application which issued the request with the smaller finish tag (lines 10 and 11). In fact, the active application is only changed in three cases:

- *Last active application already dispatched $B_{max}$ consecutive requests (line 9).* To prevent starvation, we limit the maximum number of requests an application can dispatch consecutively. Reaching that limit, the active application is changed anyhow.
- *$T_{wait}$ expired and none requests arrived (line 21).* $T_{wait}$ limits the time the disk can be kept idle, when waiting for future requests. Thus, if $T_{wait}$ expires and none requests arrives, it means that future request prediction has failed, and the scheduler must serve another backlogged application.
- *The application disk pattern is random.* The whole point of predicting future requests is to minimize the seek time overhead by dispatching requests with strong spatial locality. If the application access pattern is random, there is no gain in executing its requests consecutively.

After selecting the active application, its request with minimum finish tag will be dispatched (line 14) if the active application is *backlogged* (if there are pending requests). Otherwise, the disk will be kept idle up to $T_{wait}$ milliseconds (line 16), waiting for future requests to arrive, thus avoiding deceptive idleness. In short, either a pending or a future request is scheduled.

When a new request arrives the scheduler, through the function *add_request* (line 1), there are two possible scenarios. If the disk was waiting for future requests and the request belongs to the active application (lines 2 and 3), it should be dispatched immediately. Therefore, *unset_timer* (line 4) is called, preventing the execution of *timer_expired*, and the new request will be dispatched in the next execution of *dispatch_request*. Furthermore, if the disk was not waiting for future requests, the request is queued among the others. In

```
1   add_request (i, r)
2     if active_app == i and
3     i is waiting for the next request then
4         unset_timer ()
5     update_num_tokens (i)
6     compute_tags (i, r)

7   dispatch_request ()
8     if active_app == nil or
9     active_app dispatched more than B_max then
10        w = request with minimum finish tag F_j^w
11        active_app = application j which issued w
12    else
13        if active_app is backlogged then
14            w = request with minimum finish tag
                    F_j^w from active_app
15        else
16            set_timer (T_wait)
17            return nil
18    return w

19 timer_expired ()
20    active_app = nil
21    dispatch_request ()
```

Fig. 1. HTBS main functions.

both cases, two functions are called: *update_num_tokens* and *compute_tags*.

Figure 2 shows these two functions, called whenever a request arrives the scheduler. *Update_num_tokens* (line 1) updates the number of tokens of a given application. The new tokens available are proportional to the time elapsed since the last update, as well as the bandwidth $\rho$ assigned to the application (line 3). Tokens bound the number of requests an applications can dispatch. This function also controls bursts, through the attribute $\sigma$, by limiting the maximum amount of tokens a bucket can store (lines 4 and 5).

```
1   update_num_tokens (i)
2     Let Δ be the time interval since last request
3     numtokens_i += Δ x ρ_i
4     if numtokens_i > σ_i then
5         numtokens_i = σ_i

6   compute_tags (i, r)
7     if numtokens_i < 1 then
8         S_i^r = max {MaxS_i , t}
9         MaxS_i = S_i^r + 1 / ρ_i
10    else
11        S_i^r = t
12    F_i^r = S_i^r + δ_i
13    numtokens_i −= 1
```

Fig. 2. Fair queuing-like functions.

Finally, tags are assigned to requests through the function *compute_tags*. Start tags are set to current time (line 11), unless the application had exceeded its guarantees. In such cases, the scheduler assigns a greater value to start tags (line 8). In practice, assigning a time in the future for the start tag tries to approximate the value that this tag would have if the application had not exceeded its guarantees. Finish tags are always equal to the sum between start tag and the delay attribute, $\delta_i$ (line 12).

### *B. Parameters Discussion*

HTBS has two parameters: $T_{wait}$, which limits the maximum amount of time the disk can be kept idle when waiting for the next request from the active application, thus preventing deceptive idleness, and $B_{max}$, which bounds the number of consecutive requests one application can issue.

A reasonable value for $T_{wait}$ heavily depends on applications' and systems' characteristics. If the average processing time between consecutive synchronous requests in one system is higher, this parameter should be increased. Although, assigning a large value to $T_{wait}$ could decrease the system overall performance, since the disk can be kept idle unnecessarily when future request prediction fails.

Finally, $B_{max}$ limits the number of requests one application can issue consecutively. The greater $B_{max}$ is, greater is the number of requests dispatched with locality, whereas locality increases the system performance. However, a high value to $B_{max}$ can starve requests issued by other applications, as well as deadline guarantees can be missed.

## IV. Experimental Results

We have implemented the algorithm HTBS as a disk scheduler module for Linux Kernel 2.6.38. In order to compare HTBS with related work, we also implemented pClock, another QoS-aware disk scheduler which uses a similar tag assignment policy as HTBS, but without predicting future requests. All tests were executed in a AMD Athlon X2 240 2800 MHz dual-core processor PC, with 4 GB of DDR3 memory. The disk used is a Samsung HD080HJ SATA, 80 GB, 7200 rpm and 8 MB of onboard cache without NCQ (Native Command Queuing) support.

We used two benchmarking tools in this work: *fio*, which allows micro-benchmarking of very specific I/O workloads, and *dd*, a Linux application used to test the behavior of the scheduler along filesystems. $T_{wait}$ was set to 10 milliseconds in our experiments, which is the default waiting time used by CFQ in most Linux distributions, and $B_{max}$ was set to 20 requests.

The experiments presented in this paper are organized as follows. Firstly, we ran several synchronous workloads, using fio and dd, in order to measure the throughput increase that future request prediction can achieve. In the second experiment, we showed that in the presence of synchronous workloads, even QoS guarantees can be missed if the algorithm does not implement future request prediction. In such cases, we argue

that a non-work-conserving scheduler could still providing QoS guarantees, unlike previous work.

In the first experiment, several synchronous and sequential *fio* jobs were executed against each scheduler: pClock (work-conserving) and HTBS (non-work-conserving). All applications accessed the block device directly (bypassing the filesystem layer) and read distinct positions on the disk surface.

Figure 3 presents the results. In the first bar set, there is no significant difference between the results as just one sequential job was executed, causing no seek time. As the number of concurrent jobs grows, the aggregated bandwidth decreases due to the lack of spatial locality. However, regardless of the number of concurrent jobs, HTBS reached higher bandwidth than pClock, because HTBS was able to execute consecutively requests issued by the same application (up to $B_{max}$ requests), increasing spatial locality. As pClock' schedule decisions are exclusively based on pending requests, no more than one request per-application is dispatched consecutively due to the synchronous nature of the requests, causing pathological seek time.

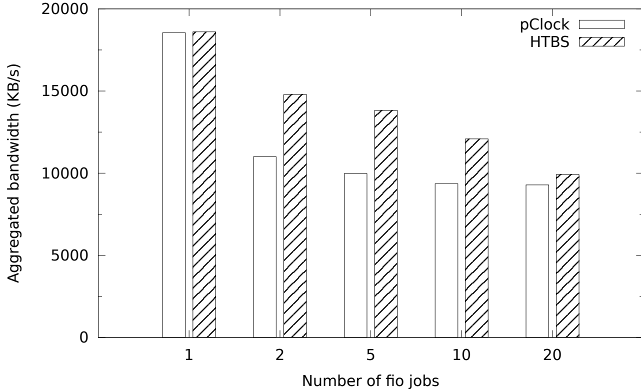


Fig. 3. Aggregated bandwidth achieved by pClock and HTBS using the fio benchmark.

Then, continuing the first experiment, we created several files within an ext3 filesystem and executed an increasingly number of concurrent *dd* threads, with the objective to measure how HTBS performs along filesystems. We also compared HTBS to our pClock implementation to check whether our future prediction scheme increases total throughput in a more realistic scenario. Every file read by *dd* processes was 100 MB sized and the request size was 4 KB.

Figure 4 shows the results. It is possible to verify that the behavior of the last experiment remains, even along filesystems. Indeed, due to future request prediction, HTBS was able to take better schedule choices when compared to the work-conserving scheduler pClock, resulting in higher performance.

After showing that HTBS provides best performance than previous work-conserving schedulers, we must also test whether HTBS can still enforce QoS guarantees to synchronous requests. To this end, in the second experiment we created four *fio* synchronous jobs: *app1* with bandwidth 8800 KB/s; *app2* with bandwidth 4000 KB/s; *app3* with bandwidth

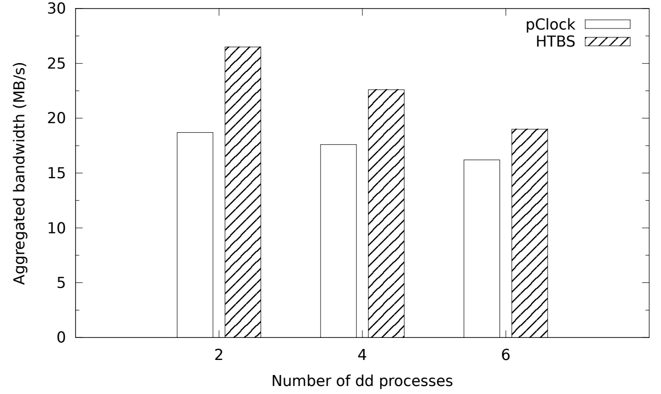


Fig. 4. Aggregated bandwidth achieved by pClock and HTBS using dd processes.

2000 KB/s and *app4* with bandwidth 800 KB/s. Deadlines were set to 100 milliseconds for all jobs and the bursts were disabled. Total execution time was 300 seconds.

Figure 5 shows the results obtained by the pClock algorithm whilst Figure 6 shows the results for HTBS. As pClock does not predict future requests, its schedule decisions are based on the fact that an application is always idle (*deceptively idle*) after issuing synchronous requests. Therefore, pClock never dispatches consecutive synchronous requests issued by the same application, even if its bandwidth guarantee is much higher than other applications.

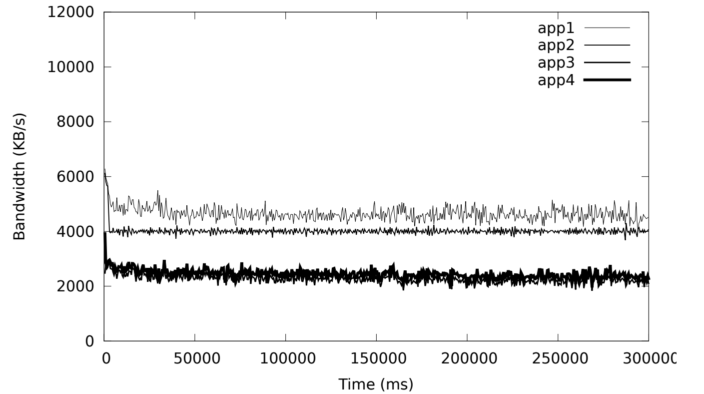


Fig. 5. pClock: four synchronous jobs with different bandwidth attributes.

HTBS, on the other hand, is able to take better scheduling choices and fulfill established QoS guarantees, as it waits for upcoming synchronous requests. On average, HTBS met bandwidth guarantees for all applications created in the experiment. Besides, through this experiment we showed empirically that some work-conserving disk schedulers can fail to provide QoS guarantees to synchronous workloads.

## V. Conclusions

This paper presented the HTBS, a new non-work-conserving disk scheduler algorithm that assigns tags to requests — similarly to fair queuing algorithms — and predicts future requests' arrival in order to provide both QoS guarantees and

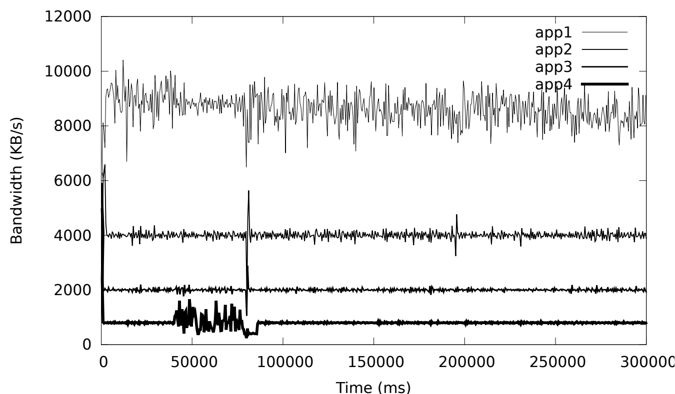


Fig. 6. HTBS: four synchronous jobs with different bandwidth attributes.

high throughput. Scheduling future requests reduces seek time by increasing spatial locality, and avoids *deceptive idleness* in the presence of synchronous requests. Through experiments performed with our Linux implementation, we showed that HTBS can increase disk performance, when compared to other QoS aware work-conserving algorithms. We also showed that previous fair queuing based disk schedulers can fail to provide QoS guarantees when synchronous requests are issued. By being non-work-conserving, HTBS can enforce applications' QoS guarantees to both synchronous and asynchronous requests.

For future work, we intend to integrate HTBS with VMMs and check whether future request prediction can still increase throughput and fulfill QoS guarantees in those systems. In addition, we also intend to continue testing the scheduler using other well-known benchmarks, such as TPC, DVDStore and filebench to simulate more realistic workloads. We believe that fair disk throughput allocation is not a closed chapter, and that our scheduler is just another step toward fair disk allocation.

## REFERENCES

[1] J. Axboe, "Linux block I/O - present and future," in *Proceedings of the Ottawa Linux Symposium*, 2004, pp. 51–61.
[2] J. Bennet and H. Zhang, "WF$^2$Q: Worst-case fair weighted fair queueing," in *Proceedings of IEEE INFOCOM*, vol. 1. IEEE, 1996, pp. 120–128.
[3] ——, "Hierarchical packet fair queueing algorithms," in *IEEE/ACM Transactions on Networkig*, vol. 5. IEEE Press, 1997, pp. 675–689.
[4] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Disk scheduling with quality of service guarantees," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*. IEEE Computer Society, 1999, pp. 400–405.
[5] P. Goyal, H. Vin, and H. Cheng, "Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks," in *IEEE/ACM/ Transactions on Networks*, 1997, pp. 690–704.
[6] A. Gulati, A. Merchant, and P. Varman, "pClock: An arrival curve based approach for QoS in shared storage systems," in *In Proceedings of ACM SIGMETRICS*. ACM, 2007, pp. 13–24.
[7] S. Iyer and P. Druschel, "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O," in *18th ACM Symposium on Operating Systems Principles*, 2001, pp. 117–130.
[8] J. Ke, X. Zhu, W. Na, and L. Xu, "AVSS: An adaptable virtual storage system," in *IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, 2009, pp. 292–299.
[9] H. Sariowan, R. Cruz, and G. Polyzos, "Scheduling for quality of service guarantees via service curves," in *In Proceedings of the International Conference on Computer Communications and Networks*, 1995, pp. 512–524.
[10] S. Seelam and P. Teller, "Fairness and performance isolation: an analysis of disk scheduling algorithms," in *IEEE International Conference on Cluster Computing*. IEEE, 2006, pp. 1–10.
[11] P. Valente and F. Checconi, "High throughput disk scheduling with fair bandwidth distribution," in *IEEE Transactions on Computing*, vol. 59. IEEE Computer Society, 2010, pp. 1172–1186.
[12] Y. Xu and S. Jiang, "A scheduling framework that makes any disk schedulers non-work-conserving solely based on request characteristics," in *Proceedings of the 9th USENIX conference on File and storage technologies*. USENIX Association, 2011, pp. 119–132.