# Implementation and Test of a Weighted Fair Queuing (WFQ) I/O Request Scheduler

Alessa Mayer, Luan Teylo, Francieli Boito

informatics mathematics

# Implementation and Test of a Weighted Fair Queuing (WFQ) I/O Request Scheduler

Alessa Mayer, Luan Teylo, Francieli Boito

RESEARCH REPORT
N° 9480
August 2022
Project-Team TADaaM

ISSN 0249-6399    ISRN INRIA/RR--9480--FR+ENG

# Implementation and Test of a Weighted Fair Queuing (WFQ) I/O Request Scheduler

Alessa Mayer[*], Luan Teylo[*], Francieli Boito[*]

Project-Team TADaaM

Research Report n° 9480 — August 2022 — 12 pages

**Abstract:** This report describes the work conducted by Alessa Mayer during a two-month internship in the Inria center of the University of Bordeaux, as a member of the Tadaam team. During her internship, her advisors were Luan Teylo and Francieli Boito. The goal of the internship was to implement and test the weighted fair queuing scheduling algorithm applied to I/O requests, and to integrate it into the AGIOS I/O scheduling library. That scheduler will be used to implement the I/O Sets method, proposed by members of the team in a recent paper.

**Key-words:** high performance computing, parallel I/O, I/O scheduling

[*] Univ. Bordeaux, CNRS, Bordeaux INP, INRIA, LaBRI, UMR 5800, F-33400 Talence, France

**RESEARCH CENTRE**
**BORDEAUX – SUD-OUEST**

200 avenue de la Vieille Tour
33405 Talence Cedex

# Implémentation et test d'un ordonnanceur Weighted Fair Queuing pour des requêtes d'E/S

**Résumé :** Ce rapport porte sur le travail mené par Alessa Mayer lors d'un stage de deux mois au centre Inria de l'université de Bordeaux, au sein de l'équipe Tadaam. Son stage a été encadré par Luan Teylo et Francieli Boito. L'objectif du stage était d'implémenter et de tester l'algorithme d'ordonnancement Weighted Fair Queuing (WFQ) appliqué aux requêtes d'E/S, et de l'intégrer dans la bibliothèque d'ordonnancement d'E/S AGIOS. Cet ordonnanceur sera utilisé pour implémenter la méthode I/O Sets, proposée par les membres de l'équipe dans un article récent.

**Mots-clés :** calcul hautes performances, E/S parallèles, ordonnancement d'E/S

# Contents

# 1 Introduction

In the field of High-Performance Computing (HPC), supercomputers with hundreds to thousands of computing nodes are used. When applications are executed, they need to have access to files stored in a parallel file system (PFS). The PFS is shared among several applications and is responsible for handling all the I/O requests defined by different I/O operations like reading and writing. During the execution, distinct applications might try to perform I/O operations at the same time which can cause interference, slowing down the applications. To improve the I/O performance, several I/O schedulers have been proposed in the related literature [1]. These schedulers try to avoid I/O interference by defining when which request will be executed.

One method to handle simultaneous access to the PFS is called *exclusive*: The scheduler allows access to only one application at a time. Another method, called *fair-share*, allows that different applications access the PFS at the same time dividing the bandwidth evenly between them. Both approaches can have advantages in some situations and be unfavorable in others. In [4] Boito *et al.* proposed a scheduling approach called *I/O Sets*, which separates the applications into sets and ensures that only applications from different sets access the PFS at the same time. Thus the idea is to use both concepts, the exclusive access between jobs of the same set and shared access between the ones from different sets.

Moreover, in the *I/O Sets* method, each set has a priority that determines the amount of bandwidth each application will receive during an I/O operation. To the best of our knowledge, current PFSs, such as BeeGFS, do not have support for priority-based bandwidth partitioning. Lustre, on the other hand, has some QoS support where the bandwidth can be partitioned among jobs. Nonetheless, jobs only get the configured portion of the PFS bandwidth even if they are the only job doing I/O operations. In the *I/O Sets* method, that is not desirable: in that situation the job should use the full bandwidth of the system. Therefore, in order to implement the *I/O Sets* method, which so far has only been validated through simulations, we need a way of assigning priorities to jobs and of partitioning the available bandwidth among the jobs that are doing I/O while respecting their priorities.

The objective of this internship was to implement the Weighted Fair Queuing (WFQ) algorithm [5] and to integrate it in AGIOS, an I/O-scheduling library developed by Boito *et al.* [2]. The library is meant to be easily integrated into existing file systems, I/O nodes, etc., and will then be used to implement the bandwidth partitioning part of *I/O Sets*.

The remaining of this report is organized as follows. In Section 2 a WFQ algorithm implementation is presented. Next, in Section 3, this first implementation is evaluated. In Section 4 the integration of WFQ in AGIOS is explained, then in Section 5 we evaluate the performance of the WFQ algorithm when run in AGIOS. Finally, in Section 6, we present the conclusions and final remarks.

# 2 Weighted Fair Queuing (WFQ)

The Weighted Fair Queuing algorithm [5] was proposed by Parekh and Gallager in 1993 and is an advanced form of Fair Queuing [3]. When applying Fair Queuing, the resource being scheduled is equally shared between the tasks/jobs/etc. However, using WFQ they will receive a share of the resource that depends on their priorities. For instance, if we are scheduling jobs to run in a processor, we could classify them into two groups and assign them priorities 1 and 2. Consequently, every time one job of the first group is executed, it will be followed by two jobs of the group with the higher priority. In our case we are scheduling I/O requests, which are read/write operations to files, starting at a certain offset and for a certain size in bytes, and the resource being shared is the available bandwidth. Those requests are assigned to different sets that are given a certain priority. The algorithm will therefore calculate the number of requests that can be executed per set while respecting the priority and this priority is measured by comparing the amount of bytes one set receives with the total amount of bytes.

The WFQ function will be called as long as there are requests in the queue that need to be executed. As parameters it receives an array that contains the queue of each set, the head of a linked list that represents the execution and the number of sets. The execution list is used to store the requests in the order in which they are processed.

```c
/* WFQ
 *
 * function takes requests from the queue and puts them in execute */
void wfq(queue_t queue[], queue_t * execute, int nbr_sets){

    // goes through one queue per function call
    request_t * ptr;
    request_t * req;

    int amount = queue[current_queue].weight + queue[current_queue].credit;
    ptr = queue[current_queue].head;

    while(ptr != NULL && amount - ptr->size >= 0) {
        ptr = ptr -> next;

        // remove first request from current queue
        req = remove_request(queue, current_queue);

        // add request to execute
        add_to_execute(execute, req);

        amount -= req->size;
        counter_execute ++;
    }

    if(queue[current_queue].head != NULL){
        queue[current_queue].credit = amount;
    }else{
        queue[current_queue].credit = 0;
    }

    current_queue = (current_queue + 1) % nbr_sets;
}
```

In a first step, the algorithm calculates the amount of bytes the queue was given depending on the according priority. The credit indicates if the weight of the set was fully used in earlier executions. If there is credit left from earlier executions, it will be added to the amount. During the next step there is a pointer going through the queue as long as requests can be executed without surpassing the amount of bandwidth the queue was given. The requests are removed from the queue through a remove_request function which returns the the removed request. After that, the add_to_execute function will add the returned request at the end of the execution list. To keep track of the amount of bytes that were served, the amount of left bandwidth is reduced by the size of the treated request. We will get outside the loop that handles the requests either because there is a request in the queue whose size is bigger than the amount the queue has left or because the queue is empty. In the first case the remaining bandwidth of the queue will be added to the credit of the queue. In the latter case the credit will be set to 0. We do that because, otherwise, a queue that is empty would accumulate too much credit over time. In both cases the current_queue variable will be updated so that the algorithm can move on to the next set.

## 3 Evaluation of WFQ

We developed a test code that generate random requests to fill the queues, and then calls the wfq function while there are requests to be processed. The function, whose code was presented in the previous section, fills a list of requests in processing order. After its execution, we can go over that list to see if the bandwidth was shared according to the proportions. However, if we simply calculate the amount of data processed from each set, what we will find is the amount of data accessed by all requests from that set. Therefore, in order to test our WFQ implementation, we devised a sliding window approach, where we calculate the proportions used by different sets in a window (defined as a number of requests) and slide the window until we reach the end of the list.

This algorithm uses a window size that is a parameter (specified in the setup file of our test code). The chosen window size will influence the output of the test. In our case, we might want to choose a window size that is large enough to make it possible to measure the shares of each set properly. At the same time the result will not be accurate if too many requests are included in the calculation. When we realised our tests, we used a window size of 300 requests, which we set after comparing the accuracy of the results with different sizes.

```c
/* test_priorities
 *
 *
 * to test the share of the bandwidth of each set over a certain window
 * this function receives an array test_results of size nbr_sets and fills the array
 * with the share for the corresponding set over the window
 *
 * Input: #sets, pointer to the first request of the window, the size of the window and the array to store
 * the test results*/
void test_priorities(char * cmd, setup_t setup, request_t * current_req, int * sets_execution_counter, bool verbose_flag){

    FILE * output = fopen(cmd, "w");

    double test_results[setup.nbr_sets];
    int remaining_requests[setup.nbr_sets];



    int j = 0;

    // writing the header
    for(int i = 0; i < setup.nbr_sets; i++){
        if(i == setup.nbr_sets - 1)
            fprintf(output,"set_%d\n", i+1);
        else
            fprintf(output, "set_%d,", i+1);
    }

    while(j < setup.N - setup.window_size + 1){ //going through all requests

        // Cleaning or starting the test_results array
        for(int i = 0; i<setup.nbr_sets; i++)
            test_results[i] = 0;


        request_t * ptr = current_req;
        int total_size = 0; //total nbr of bytes to calculate the shares

        //calculating the sum of the bytes of each set within the window
        int window_counter = setup.window_size;
        while(ptr != NULL && window_counter > 0){
            test_results[ptr->set -1] += ptr->size;
            //count the number of requests per set in the window

            total_size += ptr->size;
            ptr = ptr->next;
            window_counter -= 1;
        }

        //calculating the share of the set and writing it in the csv file
        for(int i = 0; i < setup.nbr_sets; i++) {
            test_results[i] = test_results[i] / total_size;
            if(i == setup.nbr_sets-1)
                fprintf(output,"%lf\n", test_results[i]);
            else
                fprintf(output, "%lf,", test_results[i]);
        }

        // update the counter so that we know how many requests remain in the execution list
        sets_execution_counter[current_req->set -1] -= 1;

        current_req = current_req->next;

        j ++;
    }

    fclose(output);
}
```

The function receives a file in which it will document the results of the calculations. To measure the share of the bandwidth per set the functions goes through the execution list and calculates the shares using a sliding window. The result of the calculation will temporarily be stored in an array of a size equal to the number of sets and will be written in the output file afterwards. Hence, the file contains the share of the sets for each measurement. In the end, the produced file can be used to plot the results over time.

Setting different parameters in the setup file allows us to compare different scenarios. We initialized three tests with requests of random sizes between 1 and 1000 bytes and a total number of 1000 requests. The window size in each test is of 300 requests. The requests are randomly attributed to the sets. In Figure 1 we can see the WFQ algorithm applied on 4 sets with weights of 500, 1000, 1500 and 2000. Therefore, these sets should use 10%, 20%, 30%, and 40% of the bandwidth, respectively. If all sets access a similar amount of data, the requests of Set 4, which has the highest priority, will be finished first. After that, Set 3 will have the highest priority and finish next and so on. The graph shows us that the proportion of every set will change over time while still respecting the ratio of bandwidth they were initially given. When one of the sets is no longer present, the bandwidth is redistributed among the others while respecting their priorities, which is what we wanted.

In Figure 2 10,000 requests were initialized, randomly attributed to one set in Figure 2a and to two sets with priorities of 1% and 99% in Figure 2b. Those scenarios allow us to see that the WFQ algorithm ensures maximal bandwidth. If only one set is having requests, it will get all of the bandwidth no matter its initial priority. Treating two sets where one has a priority of almost 100%, this set will get almost all of the bandwidth until it is fully executed.

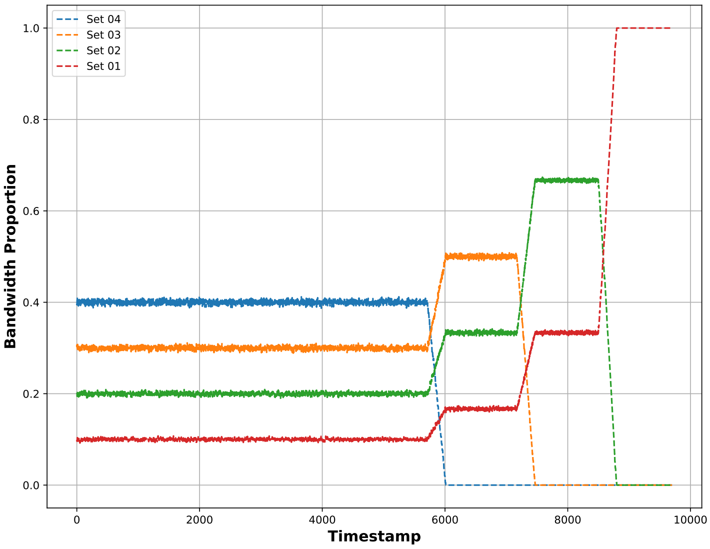


Figure 1: WFQ with 4 queues of weight 500 (0.1), 1000 (0.2), 1500 (0.3) and 2000 (0.4), respectively.

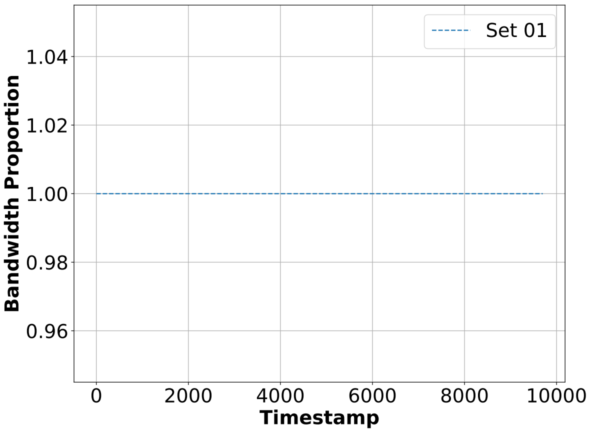
# 4 Implementing WFQ in AGIOS

The AGIOS library is designed to easily integrate new scheduling algorithms. It suffices to create a new file that contains the code of the algorithms and add it to certain files that do the initialization. We mostly used the code that is described in Section 2 and only did minor modifications so that we can use the structures of AGIOS.

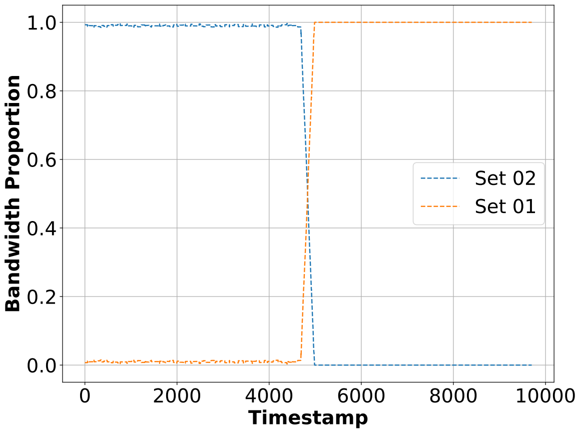
First, we created the .c and the .h files that contain the code for the WFQ algorithm. AGIOS expects a main function for the scheduler. To use our WFQ function, we needed to adjust the structures that are used in our code and added another structure that keeps the weights and the credits of the queues. To remove requests from the queues and add them to the execution list we used functions provided by AGIOS because the library uses macros. Moreover, we added a initialization function to the file that initializes the queue with the given weights. To set the latter we integrated a WFQ configuration file where the user can set the weights of the different sets, the path to that file itself is given to the library through the agios.conf configuration file, which is already used by the library to choose parameters such as the scheduling algorithm to use.

As a next step, we adapted a testing code provided with the library (the agios_test program) to measure the performance of the WFQ algorithm. To do so, we created a function that stores the time each request takes to be treated in AGIOS. We measure the starting time which is just before the request is given to AGIOS, the end time which is when the request is given back to the application to be executed, and the elapsed time in between and store them in a CSV file.

Figure 3 presents the results, calculated using our sliding window method, obtained when running the agios_test code using 10 threads with 100 requests (1000 requests in total), 5 files and 4 sets. We set the requests' size to 1024 bytes and the time between the requests and the



(a) Case of test where only one queue of weight 500 (1) is used.



(b) Case of test where two queues weights of 100 (0.01) and 9900 (0.99) are used.

Figure 2: WFQ executions where one queue has a very high priority.

time to process requests to 10 ns. The probability of sequential access was set to 100% and the weight of the four sets to 1024, 2048, 4096 and 8192.

We can see that the priorities are respected. At the beginning, all sets have approximately the bandwidth proportion we expect: 7% for set 1, 13% for set 2, 27% for set 3 and 53% for set 4. After set 4 finishes, the bandwidth partitioning is set again between the three remaining sets according to the weight and so on.

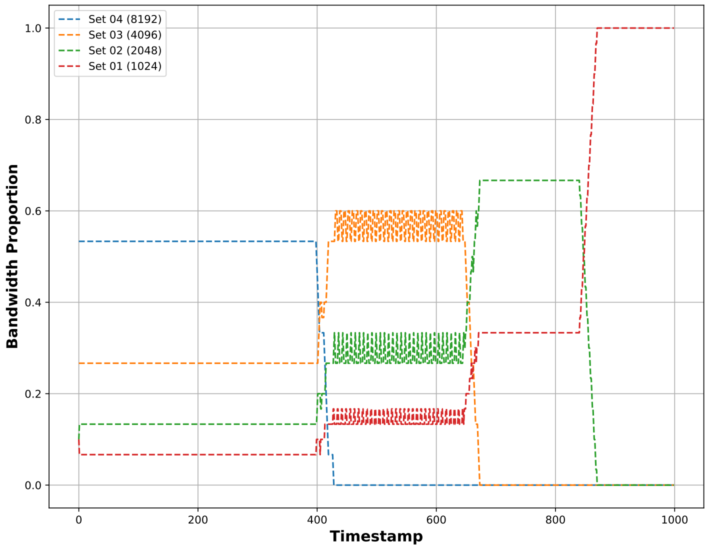


Figure 3: Proportion of the sets when AGIOS uses WFQ scheduling. The four sets are of weight 1024 (0.07), 2048 (0.13), 4096 (0.27) and 8192 (0.53).

## 5 Performance Evaluation

Even if a scheduling algorithm works as expected, we still need to verify that it takes a reasonable time to process the requests. Hence, we compared the WFQ to other schedulers that are provided in AGIOS.

To evaluate the performance of the WFQ algorithm, we used the results we generated using the test provided by the library that we modified as described in Section 4. Likewise, we measured the time AGIOS takes to treat requests using two other algorithms it offers: Timeorder (TO) and No-operation (NOOP). The latter does not do anything: requests are sent back to the application as soon as they arrive to the library (they are not even added to the internal queue), while the former treats them in FCFS (first come, first served) order (they go through the internal queue).

We present overhead which is the average time spent in the scheduling library for all requests treated during the agios_test code, varying the number of threads used to generate requests and using 100 requests per thread. The test was initialized using one file and 4 sets. The probability of sequential access was set to 100% and the time between the requests and the time to process requests to 10 ns. Again, we used a request size of 1024 bytes and, in the WFQ configuration, we set the weights to 1024 (10%), 2048 (20%), 3072 (30%) and 4096 (40%).

In results are shown in Figure 4. The x-axis represents the number of threads and the y-axis the average time AGIOS takes to process each request. We also compared the median and noted that the values resembled to the average time. When the number of threads increases, the number of concurrent requests increases as well, so that puts more pressure into the scheduling algorithm, which takes longer to process requests. That is why the wait time increases with the number of threads.

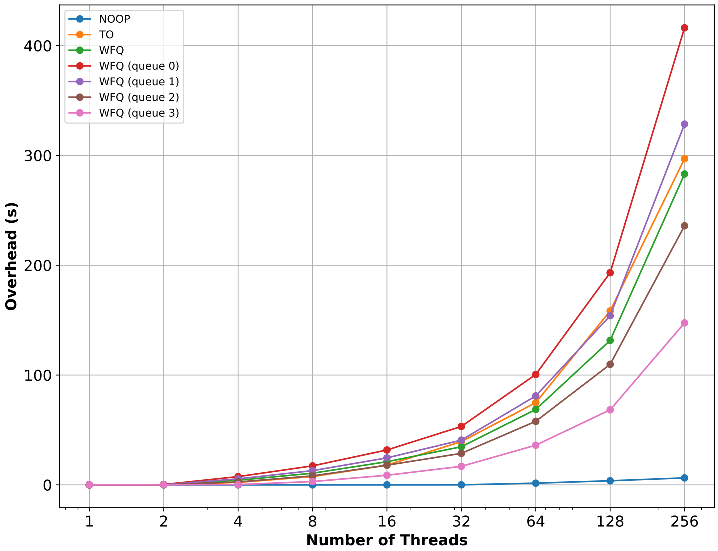


Figure 4: AGIOS run with NOOP, TO and WFQ (4 queues with weights of 1024 (0.1), 2048 (0.2), 3072 (0.3) and 4096 (0.4)). The "WFQ" line is the average of all requests in the test using WFQ.

We show that the average time, the WFQ scheduler takes, is almost the same as the one of the TO. For the queues of high priorities the time is even less. Because of its concept, the TO is a scheduler that does not apply a lot of operations to treat the requests. Hence, we can conclude that the WFQ algorithm ensures an acceptable overhead.

# 6 Conclusion and Future Work

In this work the WFQ algorithm was implemented and integrated in the AGIOS I/O scheduling library. It allows for sharing the bandwidth among requests from different sets in portions that are proportional to these sets' assigned priorities. We evaluated its performance considering distinct scenarios and we show that the algorithm allows to give requests different priorities according to sets they were given. We have also shown its performance to be reasonable.

Future work will focus on the implementation of the I/O Sets method. AGIOS will be integrated in a PFS such as BeeGFS, and also to other parts of the I/O stack such as ad-hoc file system GekkoFS and I/O controller software in the context of the ADMIRE project[1]. The use in a distributed setting brings other challenges, such as ensuring the priority-proportional sharing of the bandwidth even in scenarios where performance is limited by the network.

[1] `https://www.admire-eurohpc.eu/`

# References

[1] F. Z. Boito, E. C. Inacio, J. L. Bez, P. O. A. Navaux, M. A. R. Dantas, and Y. Denneulin. A checkpoint of research on parallel i/o for high-performance computing. *ACM Comput. Surv.*, 51(2), mar 2018.

[2] F. Z. Boito, R. V. Kassick, P. O. Navaux, and Y. Denneulin. Agios: Application-guided i/o scheduling for parallel file systems. In *2013 International Conference on Parallel and Distributed Systems*, pages 43–50, 2013.

[3] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1â12, aug 1989.

[4] G. Pallez, L. Teylo, F. Z. Boito, and N. Vidal. Io-sets: Simple and efficient approaches for i/o bandwidth management. 2022.

[5] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

**RESEARCH CENTRE**
**BORDEAUX – SUD-OUEST**

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399