Grouped Distributed Queues: Distributed Queue, Proportional Share Multiprocessor Scheduling

Bogdan Caprita, Jason Nieh, and Clifford Stein[*]
Department of Computer Science
Columbia University

Technical Report CUCS-004-06
February 2006

Abstract:

We present Grouped Distributed Queues (*GDQ*), the first proportional share scheduler for multiprocessor systems that, by using a distributed queue architecture, scales well with a large number of processors and processes. *GDQ* achieves accurate proportional fairness scheduling with only $O(1)$ scheduling overhead.

GDQ takes a novel approach to distributed queuing: instead of creating per-processor queues that need to be constantly balanced to achieve any measure of proportional sharing fairness, *GDQ* uses a simple grouping strategy to organize processes into groups based on similar processor time allocation rights, and then assigns processors to groups based on aggregate group shares. Group membership of processes is static, and fairness is achieved by dynamically migrating processors among groups. The set of processors working on a group use simple, low-overhead round-robin queues, while processor reallocation among groups is achieved using a new multiprocessor adaptation of the well-known Weighted Fair Queuing (*WFQ*) algorithm. By commoditizing processors and decoupling their allocation from process scheduling, *GDQ* provides, with only constant scheduling cost, fairness within a constant of the ideal generalized processor sharing model for process weights with a fixed upper bound.

We have implemented *GDQ* in Linux and measured its performance. Our experimental results show that *GDQ* has low overhead and scales well with the number of processors.

Keywords: Stochastic Processes/Queuing Theory, Quality of Service, Scheduling, Fair Queuing, Multiprocessor Scheduling

[*]Also in Department of IEOR

1 Introduction

Scheduling the processing resources in a time-sharing system is one of the most critical tasks for any operating system. A process scheduler apportions CPU time to the runnable processes in small periods, or *time quanta*, according to some *scheduling policy*. Since the scheduling code is run every time quantum, the scheduler needs be *efficient* (i.e. run in constant time) regardless of the number of processes in the system. More importantly, on multiprocessor architectures, the scheduler cannot ignore the overhead of synchronization mechanisms and the cache effects of switching tasks between processors, and should be designed to minimize the need for or occurrence of such events ([14]).

An attractive scheduling policy is *proportional sharing*, or *fair-share scheduling*, which allocates a fixed share of CPU time to each process ([9]). Each process is assigned a *weight* that defines the service rights of that process: the CPU time received should be in proportion to the weight. That is, a process A of weight ϕ_A receives a share of $\frac{\phi_A}{\sum_{\text{all processes } C} \phi_C}$. In such a model, a process is guaranteed its share of CPU time regardless of the behavior of other tasks. Proportional share schedulers also provide system administrators with precise control over the allocation of processing time. Because of its benefits, proportional sharing has received much attention, and numerous schemes to implement single resource proportional sharing have been proposed ([1], [3], [6], [7], [8]). However, accurate proportional share schedulers have not been adopted in operating system kernels, mainly because they are difficult to implement accurately ([10]). Instead, simpler heuristic algorithms which allocate CPU time in coarse time intervals are used, but these are not suited for supporting interactivity or for satisfy tight processing requirements. More recently, single processor schedulers have been designed that combine accurate proportional sharing with simple, efficient algorithms ([3]).

Multiprocessor scheduling is considerably less well understood, and, in practice, relies mostly on heuristics ([15]). A multiprocessor scheduler has the same goals as a single processor scheduler, except that the resource is no longer a single CPU, but instead a set of 2 or more processors. Along with the need to distribute the scheduling algorithm on several nodes, a multiprocessor system raises additional difficulties for proportional sharing: the process weights are not guaranteed to form a feasible mix, balancing work across processors requires expensive task migrations, and, in general, book-keeping needs to grow even as the sharing of information becomes more expensive due to synchronization and caching.

Because of this added complexity, proportional share multiprocessor schedulers are scarce, and usually operate with a single, centralized queue ([3], [4]). Due to lock contention, centralized queue schedulers clearly do not scale beyond just a few processors. From an implementation standpoint, there is a qualitative difference between using a single queue and using per-processor queues: the former needs a global lock, which involves accessing main memory each time the lock is grabbed or released, even on a dual processor machine. Furthermore, as the number of processors increases, there will be tremendous contention for the single lock, which hence becomes the performance bottleneck. In addition, if processes are scheduled from a single queue, a single process will be unlikely to run consecutive times on the same processor and therefore will not take advantage of the previous cache state.

In this paper, we present the **Grouped Distributed Queues (*GDQ*)** proportional-share task scheduling algorithm, which achieves fine-grained control over resource isolation in multiprocessor systems. Because of the aforementioned drawbacks to using a centralized queue, we designed *GDQ* to distribute the queuing data structure and localize task queues at each processor. Furthermore, *GDQ* is designed to scale well not only with the number of processors, but also with the number of tasks. Constant overhead and simple data structure updates are key to scheduler efficiency.

Traditionally, distributed queue scheduling implies assigning a queue of tasks to each processor, such that the queue identifies one-to-one with the processor, and the processor works solely on tasks within its queue. To balance load, the queues grow and shrink as processes migrate between queues. This simple model imposes an expensive trade-off between CPU allocation fairness and efficiency. As queues become unbalanced, some processes fall behind and the queues need to be rebalanced. However, moving processes among queues too often nullifies the main benefits of having distributed queues: light lock contention and good cache affinity.

The starting point of *GDQ* was to separate the balancing of processor queues and process scheduling such that the former can be optimized for fairness and the latter for efficiency without globally sacrificing either. *GDQ* proposes an inverted paradigm for pairing up processors and processes: 'queues' are static, and processors migrate from 'queue' to 'queue'. That is, processes are aggregated into groups based on their weight, and remain in their groups for their entire runnable lifetime, whereas processors are assigned to perform work on the groups such that proportional sharing is maintained. At regular intervals, processors are reassigned from one group to another, thus ensuring that groups progress at proportional sharing rates. We present a new algorithm, called *Multiprocessors Fair Queuing* (*MFQ*), that manages the processor allocation among groups. Simple round-robin queues will then be used inside groups to schedule the processes.

The grouping and processor allocation strategies allow *GDQ* to maintain tight fairness among the scheduled processes, while avoiding expensive computation and processor reallocations. Our formal analysis of *GDQ* captures the design goals of *GDQ*:

- constant time overhead, regardless of the number of processes
- fairness within constant bounds of an ideal scheduler

In addition to these theoretical results, we have conducted experiments to demonstrate the power of *GDQ*. Simulation studies show very good fairness bounds that scale well with the number of processes and processors. Furthermore, *GDQ* can be easily and efficiently implemented. A prototype *GDQ* scheduler for Linux compares favorably against standard Linux schedulers ([10]) as well as against a single queue proportional share multiprocessor scheduler ([3]).

Sections 2–4 describe the *GDQ* algorithm, its analysis, and experiments. We defer a detailed comparison to related work until Section 5.

2 *GDQ* Scheduling

At a high-level, the *GDQ* scheduling algorithm consists of three parts, a process grouping strategy, an intra-group allocation algorithm and an inter-group allocation algorithm. In the presentation, we abstract the notion of a *time quantum*, which is the maximum time interval a process is allowed to run before another scheduling decision is made, and refer to the units of time quanta as adimensional time units (tu) rather than an absolute time measure such as seconds.

The next section contains definitions and the basic grouping strategy, while subsequent sections describe the various algorithms.

2.1 Definitions

P and N denote the number of processors, and processes, respectively. The P processors are labeled $\wp_1, \wp_2, \ldots, \wp_P$. The *order* σ_C of a process C having *weight* ϕ_C, is defined as $\lfloor \log \phi_C \rfloor$ (all logs are base 2). The order is easily computed as the first bit set in the binary representation of the process weight. For any C, we keep track of its *work*, w_C, which measures the amount of CPU time that the process has received so far. Work is measured in adimensional time units (tu) which counts how many time quanta a process has consumed. The *normalized virtual time (NVT)* of the process is defined as $nv_C = w_C \frac{2^{\sigma_C}}{\phi_C}$. Because $2^{\sigma_C} \leq \phi_C < 2^{\sigma_C+1}$, the *NVT* scales the work of C down (up to a factor of 2) such that all processes within a group will have similar *NVT*s.

GDQ groups processes together exponentially by weight, such that *group* G^k contains all the processes with weights between 2^k (inclusive) and 2^{k+1} (exclusive) [1]. We call $G^k = \{C : 2^k \leq \phi_C < 2^{k+1}\}$ the *group of order k*, where k is the order of all the processes in G^k. The number of groups is denoted by g, and can be at most $\lfloor \log \phi_{\max} \rfloor + 1$, where $\phi_{\max}$ is the maximum possible weight. For example, with 32 bit weights, $g \leq 32$. We associate the following variables with a group G: the *weight* of the group, Φ_G, is the sum of the weights of all processes in G; the *work* of G, W_G is the sum of the work of all processes in G. Finally, N^k is the number of processes in group G^k. Clearly, $\sum_{G^k} N^k = N$. For the sake of brevity, we will use X_k to mean X_{G^k} for any variable X.

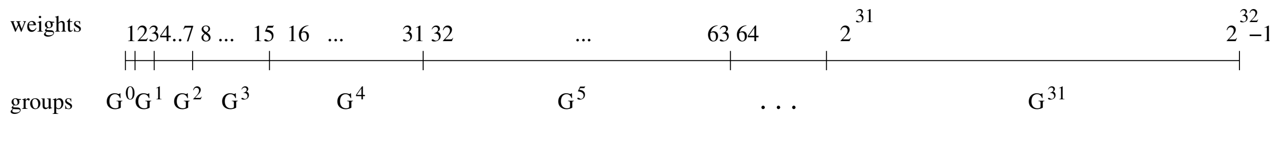

Figure 1: *GDQ* grouping strategy

GDQ also keeps track of the following: Φ_T, the *total weight*, is the sum of all process weights (or group weights); W_T, the *total work*, is the sum of the work of all processes (or groups). When there are at least P

[1] henceforth, except for writing powers of 2, superscript does **not** denote exponentiation.

runnable processes in the system at all times, $W_T = Pt$ where t is the elapsed time.

According to its weight, any group G^k is "entitled" to be serviced by $\frac{\Phi_k}{\Phi_T}P$ processors. *GDQ* attempts to allocate processors to groups fairly. However, a group's processor allocation at any time, denoted by P^k, must be an integer. Define $\underline{P}^k = \left\lfloor \frac{\Phi_k}{\Phi_T}P \right\rfloor$ (floor) and $\overline{P}^k = \left\lceil \frac{\Phi_k}{\Phi_T}P \right\rceil$ (ceiling). Unless $\frac{\Phi_k}{\Phi_T}P \in \mathbb{N}$, $\overline{P}^k = \underline{P}^k + 1$. *GDQ* will then allocate either $\underline{P}^k$ or $\overline{P}^k$ processors to a group depending on the work accumulated by the group so far and the state of the scheduler. The P^k processors allocated to group G^k are labeled $\wp_1^k, \wp_2^k, \ldots, \wp_{P^k}^k$.

Groups are organized into a list of size g. Inside groups, processes are organized into *queues*, which are linked lists. $next(C)$ denotes the process that follows C in the list. A group has $\overline{P}^k$ queues, and each queue is in general associated with a single processor. $\wp(Q)$ denotes the processor that works on the queue Q, and $Q(\wp)$ denotes the queue that processor $\wp$ works on. Each queue Q keeps track of a *current process*, denoted by $C(Q)$, which is receiving service from $\wp(Q)$. A per-queue *NVT*, denoted nv_Q, is used as a round counter to advance the *NVT* of the queue's processes. All queues of a group are organized into a per-group linked list.

The notation introduced is summarized by table 1 in the Appendix.

2.2 Basic Algorithm

Instead of binding individual queues to processors and keeping these queues balanced, we keep the groups fixed and distribute the processors among groups. The set of processors allocated to a group will be kept somewhat stable, thus taking advantage of locality and helping service isolation. In general, depending on its weight, a group can have between 0 and P processors allocated. Work balance is achieved by dynamically changing the processor allocation to groups.

We first present the *GDQ* operation under the assumption that the set of processes and their weights remains unchanged. We call *steady-state* such intervals of time during which the process mix doesn't change.

The *GDQ* algorithm can be briefly described as a two-level hierarchical scheduler:

Inter-group allocation. At any time, each group G^k is allocated either $\underline{P}^k$ or $\overline{P}^k$ processors. At certain times, a processor is removed from a group that is over-allocated, and moved to a group that is under-allocated, thus balancing work across groups, according to the inter-group scheduler (Section 2.2.1).

Intra-group allocation. Let G^k have a processor allocation of P^k. At any time, $\sum_{i=1}^{g} P^k = P$ if we are assuming a work-conserving system which has at least P processes.

The P^k processors that are allocated to group G^k will each be responsible for one of the $\overline{P}^k$ queues of the group. When $P^k = \overline{P}^k$, all the queues are non-empty, and have a processor associated with them. When $P^k = \underline{P}^k$, all but one of the queues have a processor; one queue will be *stalled*. This queue may be empty. All other queues are called *active*. Since all processes within a queue belong to the same group and thus have similar weights, the processor can proceed in a round-robin manner through the queue to select a process to run. To balance the queues of a group, processes may be moved away from the queue that is most behind in terms of its *NVT*. The intra-group scheduler is described in more detail in Section 2.2.2.

2.2.1 Inter-Group Allocation

Since the inter-group scheduler is oblivious with respect to the intra-group scheduler, we will abstract groups as clients, and assign processing time in accordance with their weight (the group weight). In effect, we are presenting a stand-alone multiprocessor scheduler where clients may run in parallel with themselves (i.e., since clients are really groups here, they may be serviced by several processors simultaneously), which we employ as part of *GDQ* to manage processor allocation among groups. The algorithm has the following *smoothness* property: given a client of weight ϕ_i, it will at any time run on either $\left\lfloor \frac{\phi_i}{\Phi_T} P \right\rfloor$ or $\left\lceil \frac{\phi_i}{\Phi_T} P \right\rceil$ processors, which is in some sense the best we can do in terms of matching the client's ideal allocation. The *MFQ* scheduler is described below in its most general form, where the entities being scheduled are called 'clients'.
Multiprocessor Fair Queuing (*MFQ*). For a client C of weight ϕ_C, the *virtual finishing time* (*VFT*) is defined as $F_C = \sum_\tau \frac{w_C(\tau)+1}{\phi_C}$, where the sum is over periods τ during which ϕ_C remains constant. When the client's weight is always constant, F_C is simply $\frac{w_C+1}{\Phi_C}$. [7] and [11] offer a more detailed discussion of the notion of a *VFT*.

We now present *MFQ* below, noting that it clearly preserves the smoothness property:

For each client i such that $\phi_i \geq \frac{\Phi_T}{P}$, we devote $\left\lfloor \frac{\phi_i}{\Phi_T} P \right\rfloor$ processors to this client and, if $\frac{\phi_i}{\Phi_T} P \notin \mathbb{N}$, we create a fictitious client of weight $\hat{\phi}_i = \phi_i - \left\lfloor \frac{\phi_i}{\Phi_T} P \right\rfloor \frac{\Phi_T}{P}$ which replaces client i in the scheduler.

Because of the aforementioned step, we can assume that each client has weight $\hat{\phi}_i < \frac{\Phi_T}{P}$ where $\hat{\phi}_i$, P, and Φ_T are adjusted for any dedicated processors as described above. This means that no client receives more than one processor at any time from the scheduler; dedicated processors are not counted here. We denote the length of the inter-group time quantum by T (typically, much larger than the time quantum that the intra-group scheduler assigns to processes). Every interval of length T is split into P subintervals of time during which the processor assignment to clients stays fixed. For each subinterval, in round-robin order, a processor is removed from the client it is currently assigned to, and is given to the client that has the least virtual finishing time which is not currently assigned a processor. The *VFT* of the client is then incremented by $\frac{1}{\hat{\phi}_i}$. All clients start out with a *VFT* of $\frac{1}{\hat{\phi}_i}$.

The *MFQ* scheduler can then be summarized by the following routine, executed by each processor $\wp_j$ with frequency $1/T$, such that processor $\wp_1$ executes the routine at times $T,\ 2T,\ 3T\ldots$, processor $\wp_2$ at times $T+\frac{1}{P}T,\ 2T+\frac{1}{P}T,\ 3T+\frac{1}{P}T\ldots$, and, in general, processor $\wp_j$ at times $T+\frac{j-1}{P}T,\ 2T+\frac{j-1}{P}T,\ 3T+\frac{j-1}{P}T\ldots$. Since the inter-group clients are really groups, inter-group time quanta T are actually made up of many intra-group time quanta. Therefore, the times when inter-group scheduling decisions are made will be rounded up to the nearest time quantum boundary, affecting neither the fairness (virtual times are computed based on the actual work received by the client), nor data structure contention (the scheduling interval T is taken to be larger than P time quanta).

As mentioned, we have developed this scheduling algorithm to allocate the processors among groups. The pseudo-code below is tailored to our usage of the presented *MFQ* scheduler as an inter-group allocator, where the clients are in fact groups:

```
MFQ($\wp_j^k$)
1  if IS-DEDICATED($\wp_j^k$)
2     then return  ▷ dedicated processors don't get reassigned
3     else $F_{G^k} \leftarrow F_{G^k} + \frac{1}{\Phi_k}$
4          $G \leftarrow NIL$
5          for each group $G^i$
6              do if ($P^k = \overline{P}^k - 1$) AND ($G = NIL$ OR $F_i < F_G$)
7                    then $G \leftarrow G^i$
8          if $G \neq G^k$
9             then REASSIGN($\wp_j^k, G^k, G$)
```

Phrased in terms of groups, REASSIGN$(\wp, G^k, G)$ will move processor $\wp$ from group G^k to group G, by first setting $Q(\wp)$ to be the stalled queue of G^k, and then assigning G's stalled queue to $\wp$. If this queue is empty, a process from the queue of G with at least 2 processes having the smallest NVT is transferred over (such a queue always exists in steady-state operation).

2.2.2 Intra-Group Allocation

Each group G^k has N^k processes with weights between 2^k and $2^{k+1} - 1$, and, according to the inter-group allocation, there are P^k processors assigned to service G^k. If P^k would never change, if N^k was a multiple of P^k, and if all processes had the same weight, then intra-group allocation would be trivial, as we would simply partition the processes equally into P^k round-robin queues and optimal proportional sharing would be maintained (assuming all processors progress at the same rate). However, as we saw in the inter-group algorithm, P^k varies between $\underline{P}^k$ and $\overline{P}^k$, N^k may be any number (greater than $\underline{P}^k$, as we will see), and the weights of processes in G^k may vary by up to a factor of 2. Still, intuitively, partitioning and round-robin traversal should be well-suited to take advantage of the tight weight distribution of processes within the group. The intra-group algorithm follows this approach, but does not explicitly attempt to partition the processes. An optimal partition is beyond efficient computation [2], and while arbitrarily accurate approximation schemes exist, finding a good partition is not worth the high computational cost, since, unless there exists a perfect partition, maintaining fairness demands that we repartition at regular intervals.

[2]Even with job processing times limited to the interval $[2^k..2^{k+1})$, the Minimum Makespan Scheduling Problem remains NP-hard.

Instead, the algorithm creates $\overline{P}^k$ queues for G^k (one of which might be stalled), and strives to keep the *NVT*s of all queues approximately equal. This is accomplished by moving a process from a queue with smallest *NVT* to a queue with larger *NVT*, which will eventually balance the queues implicitly. Each queue has one processor that schedules in round-robin order, where each process runs either one or two consecutive time quanta, depending on its weight and previous allocation history. The queue *NVT* is incremented at the end of each round, while the process *NVT*s increase by $\frac{2^k}{\phi_C} \in (1/2, 1]$. Processes run once or twice to keep their *NVT* ahead of the queue *NVT*. The processes that already ran during a round have an *NVT* larger or equal to the queue *NVT*, and the processes yet to run in the round have an *NVT* smaller than the queue *NVT*. Thus, when the next process' *NVT* is greater than the queue *NVT*, we know the round is over. At the end of each round, the processor checks the queue in the group with the smallest *NVT*. If this is less than the *NVT* of the processor's queue by more than some δ, a process is 'stolen' (transferred over). The processor than works exclusively on that process until its *NVT* is larger than the queue's *NVT*.

The stalled queue does not have a processor assigned to it. In time, its *NVT* stays constant and will become the lowest in the group, so that its processes will be transferred over to the active queues.

The following pseudo-code defines the intra-group algorithm more precisely. The routine is executed by any processor $\wp_j^k$ of group G^k after every time quantum or whenever it needs to select a new process to run.

```
INTRAGROUP($\wp_j^k$)
1   $Q \leftarrow Q(\wp_j^k)$
2   if $nv_{C(Q)} \geq nv_Q$  $\triangleright$ Move on to the next process
3      then $C(Q) \leftarrow next(C(Q))$
4           if $nv_{C(Q)} \geq nv_Q$ $\triangleright$ End of the round
5              then $MinQ \leftarrow$ GET-MIN-QUEUE($G^k$)
6                   if $nv_Q > nv_{MinQ} + \delta$
7                      then $MinC \leftarrow next(C(MinQ))$
8                           MOVE-PROCESS($MinC, Q$)
9                      else  $nv_Q$++
10  $nv_{C(Q)} \leftarrow nv_{C(Q)} + 2^k/\phi_{C(Q)}$
11  return $C(Q)$
```

The number δ, like the interval length T for the inter-group scheduler, is a parameter of the algorithm.

GET-MIN-QUEUE(G) finds the queue in the group G who has the smallest *NVT*. This queue may be either the stalled queue, or may be an active queue. In the latter case, the queue must contain at least 2 processes to be eligible for selection. No such restriction exists for the stalled queue, which is allowed to become empty.

MOVE-PROCESS$(MinC, Q)$ moves process *MinC* to the queue Q and places it right before the current process, $C(Q)$. *MinC* becomes the new current process of Q. Since *MinQ* has at least 2 processes, $MinC \neq C(MinQ)$ and thus it is safe to steal *MinC* from *MinQ*.

2.3 Feasibility

While the weight values assigned to processes in a single-processor environment are essentially unconstrained, this is not the case in a multiprocessor system. Because a process may only receive service from one processor at a time, it is impossible to satisfy a weight assignment that would give a process more than $1/P$ of all processor resources. Such a weight assignment is said to be *infeasible*, and is disallowed. Thus, a feasible weight assignment for the processes in the system is one in which

$$\phi_C \leq \frac{\Phi_T}{P} \text{ for any process } C \tag{1}$$

If a weight assignment is found to be infeasible, it is adjusted to the closest feasible assignment. As a benefit of grouping processes exponentially by weight, we can readily employ the novel weight adjustment algorithm that was introduced in [3]. This algorithm does not need to maintain additional data structures such as sorted lists of weight, and performs fast weight readjustment, with optimal time complexity of only $O(P)$.

We have assumed several times in the description of the *GDQ* algorithm that we have at least one process in each per-processor queue. We now support this assumption by noting that each non-stalled queue will have at least one process provided that $N^k \geq \overline{P}^k$. Since

$$\overline{P}^k < \frac{\Phi_k}{\Phi_T} P + 1 = \frac{N^k \phi^k_{\text{ave}}}{\Phi_T} P + 1 \leq N^k + 1$$

(where ϕ^k_{ave} is the average weight of processes in G^k), and N^k and $\overline{P}^k$ are both integers, it must be that $\overline{P}^k \leq N^k$ (assuming $\frac{\phi^k_{\text{ave}}}{\Phi_T} \leq \frac{1}{P}$, which follows from the feasibility constraint 1).

2.4 Dynamic Considerations

We assumed so far that all the processes are permanent. We now handle the cases when processes are enqueued or dequeued at the scheduler. As a first step, *GDQ* always runs the weight readjustment algorithm to ensure the new weight mix is feasible (Section 2.3). As a result, the two highest order groups may merge (for a departure), or the highest order group may split off a new group of order incremented by one (for an arrival). The group losing or gaining a process also changes its weight. In any case, at most 3 groups are affected.

As a second step, the group's processor allocation, $\overline{P}^k$ and $\underline{P}^k$, are recomputed for all groups G^k, $k = 0 \ldots g-1$. Since the inter-group algorithm, *MFQ*, is virtual time based, no other readjustment is necessary: the rate of increase in virtual time will automatically change with the new group weight. After this step, some groups may find themselves having too few (less than $\underline{P}^k$) or too many (more than $\overline{P}^k$) processors. The inter-group allocation will re-balance this situation as it proceeds. We avoid re-assigning processors at this step, since in the worst-case, $\Omega(P)$ processors would need to change groups.

Finally, in the case of an arrival, the process is added to the smallest (in terms of weight) queue of the group it belongs to. In the case of a departure, the queue the process was on is checked to have at least a process remaining. If this is not the case, a process will be pulled from the queue of the group with the least *NVT*, in accordance to the intra-group scheduler (Section 2.2.2). If there are no queues with at least 2 processes in the group, the processor is reassigned to another group, in accordance to the inter-group scheduler (Section 2.2.1), or is left idle, if N, the number of processes, has fallen below P, the number of processors.

3 *GDQ* Fairness and Complexity

We analyze the fairness and complexity of *GDQ*. We strive to formalize the O(1) error bounds and running time, while allowing dependencies in P or g, constants in practice.

Due to space constraints, most proofs are presented in the appendix.

3.1 Fairness

Proportional share schedulers should guarantee that processes do not deviate too much from their proportional allocation. The metric of choice to analyze the fairness of proportional sharing is the *service error* ([1, 3, 11]). For any process C, the service error e_C is defined as $e_C = w_C - \frac{\phi_C}{\Phi_T} W_T$. The error for a process C captures the difference between the CPU time received by the process and the share of the total CPU time that that process was entitled to according to its weight. A good proportional share algorithm must make sure that the error does not become too negative or too positive, and should ideally keep it around 0.

GDQ is designed to bound the service error by constants that depend only on P, the number of processors, and g, the number of groups. Since there is no dependence on N, the number of processes, the algorithm is fair even in the presence of very large loads.

We will analyze the inter- and intra-group fairness, and then combine them to get the overall fairness of the *GDQ* scheduler. To start, we will present the argument leading to the service error bounds of the inter-group scheduling algorithm *MFQ* (Theorem 3.5) which will be central to the analysis of the fairness of *GDQ*.

3.1.1 Inter-Group (*MFQ*) Fairness

We will use the following model for the operation of the scheduler, justified by the fact that no two processors schedule simultaneously: time is discretized into a sequence of *points*, whose spacing is irrelevant. Processors schedule successively at a point each, and so a processor will schedule every P points. We call the time between two consecutive scheduling points of the same processor an *interval*. As mentioned, an interval consists of P successive points.

Recall, for the purpose of the inter-group scheduler, we abstract groups as being clients of *MFQ*. In this language, a client ‘runs’ on a processor if the processor is allocated to the group. We will first assume that all clients have weight less than $\frac{\Phi_T}{P}$ (no client has any dedicated processors).

The proof of fairness will build on the following lemma, which formalizes the intuition that a client will run consecutively on its processor until it has caught up to its rightful allocation.

Lemma 3.1. *Consider any scheduling point t, and let the client selected be j. If at point t, some client i is such that $\frac{w_i(t)+\delta_i}{\phi_i} < \frac{w_j(t)+1}{\phi_j}$ for some integer $\delta_i \geq 1$, then client i is running at point t, and will be scheduled for another δ_i intervals continuously.*

As shown in [7], for a uniprocessor *VFT* algorithm, $\frac{w_i+1}{\phi_i} \geq \frac{w_j}{\phi_j}$ for any clients i, j. Using the previous lemma, we show that our algorithm preserves this property most of the time:

Lemma 3.2. $\frac{w_i+1}{\phi_i} \geq \frac{w_j}{\phi_j}$ *for any client i that is not running and for any client j.*

Lemma 3.2 suggests that a client never falls behind its ideal allocation by more than 1 tu. This imposes a bound on the negative error for not running clients, and in fact can be extended to all clients.

Lemma 3.3. *For any client i not currently running, $e_i \geq -1$.*

Lemma 3.4. *For any client i, $e_i \geq -1$.*

We conclude with the complete error bounds for *MFQ*:

Theorem 3.5. *For any client i, $-1 \leq e_i \leq N$.*

Proof. The negative error bound is given by Lemma 3.4. For the positive error, we note that at any time, $\sum_{i=1}^{N} E_i = 0$, and hence $E_i > -1 \ \ \forall i$ implies $E_i < N \ \ \forall i$. □

We started with the assumption that all clients have weight less than $\frac{\Phi_T}{P}$. This was for the simplicity of the analysis, and can now be removed. The result of the first step in the algorithm is to separate the total weight Φ_T into a part that receives P^1 dedicated processors, call this weight Φ_T^1, and a part that is subject to the *VFT*-based allocation and runs on the remaining P^2 processors, call it Φ_T^2. It holds that $\frac{P^1}{P} = \frac{\Phi_T^1}{\Phi_T}$, and, since $P^1 + P^2 = P$ and $\Phi_T^1 + \Phi_T^2 = \Phi_T$, we have

$$\frac{P^1}{\Phi_T^1} = \frac{P^2}{\Phi_T^2} = \frac{P}{\Phi_T}.$$

For any client i whose weight was initially less than $\frac{\Phi_T}{P}$, its error at time t, $w_i - \frac{\phi_i}{\Phi_T}Pt$, is the same as $w_i - \frac{\phi_i}{\Phi_T^2}P^2 t$, which is bounded as described in the analysis of the *VFT*-based algorithm.

For a client i whose weight was initially at least $\frac{\Phi_T}{P}$, its weight ϕ_i was split into a part ϕ_i^1 that had P_i^1 processors dedicated to it, such that $\frac{\phi_i^1}{\Phi_T} = \frac{P_i^1}{P}$, and a part ϕ_i^2 that participated in the *VFT*-based algorithm.

Denote the work received by the client from its dedicated processors as w_i^1, and the work received from participating in the *VFT*-based algorithm as w_i^2. The client's error at time t, $w_i - \frac{\phi_i}{\Phi_T}Pt$ is

$$\begin{aligned} e_i &= w_i^1 + w_i^2 - \frac{\phi_i^1 + \phi_i^2}{\Phi_T}Pt &&= P_i^1 t - \frac{\phi_i^1}{\Phi_T}Pt + w_i^2 - \frac{P}{\Phi_T}\phi_i^2 t = \\ &= P_i^1 t - \frac{P_i^1}{P}Pt + w_i^2 - \frac{P^2}{\Phi_T^2}\phi_i^2 t &&= w_i^2 - \frac{\phi_i^2}{\Phi_T^2}P^2 t \end{aligned}$$

which is again the error for the *VFT*-based algorithm.

3.1.2 *GDQ* Fairness

We now proceed to analyze the intra-group algorithm, and see how to use Theorem 3.5 to bound the error of *GDQ*. For the **lower error bound**, we must consider a process that is not running, since clearly a process that is running receives at least as much CPU time as its due share.

Inter-group. From Theorem 3.5, the inter-group error of any group is bounded below by $-T$, where T is the length (in number of time quanta) of the inter-group scheduling interval.

Intra-group. Since the process is not running, it cannot be the sole process of an active queue, or a process just transferred to an active queue from either the stalled queue or another active queue.

An active queue can be either *NVT*-balanced, when it is performing round-robin traversal, or can be *NVT*-imbalanced, if the queue's processor is working to bring the *NVT* of a recently transferred process to the level of the queue *NVT*. In the former case, the *NVT* of all processes in the queue are within 1 of the queue's *NVT*, and the queue's *NVT* is within δ of any other queue *NVT*. Hence, the process' *NVT* is no less than $-(\delta + 2)$ from that of any other process.

If the process is part of a *NVT*-imbalanced queue, then its *NVT* is still within 1 of the queue *NVT*, because the queue had been *NVT*-balanced before the new process was transferred in. This follows from the fact that no process is transferred into an *NVT*-imbalanced queue. The bound $-(\delta + 2)$ holds the same.

A process in a stalled queue can be at most δ behind the queue *NVT*, which is in turn at most δ behind any other queue *NVT*, so the bound is $-(2\delta + 1)$.

The *NVT* bound of $-(2\delta + 1)$ translates into an intra-group error of no less than $-2(2\delta + 1)$.

For the **upper error bound**, consider a running process.

Inter-group. From Theorem 3.5, the inter-group error of any group is bounded above by gT, where T is the length (in number of time quanta) of the inter-group scheduling interval and g is the number of groups.

Intra-group. If the process is part of an *NVT*-balanced active queue, in which case it is ahead of the queue *NVT* by no more than 1, then, since the queue *NVT* is within δ of any other queue, the upper bound on the *NVT* difference is $2\delta + 2$.

If the process is part of an *NVT*-imbalanced active queue, then it must be the process that is being serviced to bring its *NVT* to the level of the queue *NVT*. The process can thus not be more ahead than the *NVT* of the queue, which is within δ of the *NVT* of any other queue. The *NVT* bound of $2\delta+2$ translates into an intra-group error of no more than $2(2\delta+2)$, and the inter- and intra-group bounds can be combined to get the overall scheduling error for *GDQ*:

Theorem 3.6. $-2(2\delta+2)-T \leq e_C \leq 2(2\delta+2)+gT$

Proof. $e_C = w_C - \frac{\phi_C}{\Phi_T}W_T = w_C - \frac{\phi_C}{\Phi_G}W_G + \frac{\phi_C}{\Phi_G}W_G - \frac{\phi_C}{\Phi_T}W_T = w_C - \frac{\phi_C}{\Phi_G}W_G + \frac{\phi_C}{\Phi_G}(W_G - \frac{\phi_G}{\Phi_T}W_T)$. $\square$

To simplify the analysis, we have implicitly assumed that a process which runs continuously on a processor does not decrease its group-relative error. Such an assumption is not justified when $\frac{\phi_i}{\Phi_k} > \frac{1}{P^k}$ for some $C_i \in G^k$. We refer to process i as being *almost infeasible*. This does not contradict the feasibility constraint described in Section 2.3, since $\frac{\phi_i}{\Phi_k} \leq \frac{1}{\underline{P}^k}$ will still hold true. However, in the case of almost infeasible processes, we can use a different approach to bound their error, by noting that before they were running continuously, their error was bounded as in Theorem 3.6, and by running, even though their group-relative error may decrease, their overall error must be non-decreasing. As for processes that are in the same group as almost infeasible processes, their positive error does not grow more than the bound of Theorem 3.6, since, in effect, the almost infeasible processes get their own processors for the duration they are running continuously, and the work of the remaining processors is distributed in the group under the constraints of Theorem 3.6.

3.2 Time Complexity

It is crucial that the kernel scheduler have low overhead, regardless of the number of processes that it needs to schedule. It this section we show that

Theorem 3.7. *GDQ scheduling incurs constant complexity per decision.*

Proof. **Inter-group (*MFQ*) allocation** At each inter-group subinterval, the processor that is scheduled will need to identify the group of least *VFT* that is not running. This takes $O(g)$ time , or $O(\log g)$ time with more complicated data structures (but in practice, we use $O(g)$-time solution, since g is here the number of groups, and we need no locking to traverse an array of groups). The number of subintervals is P, so a time interval T is split into more subintervals as P increases. Since we space the subintervals equally, and only one processor schedules at the border of subintervals, there should be no lock contention issues. Note that no matter how large P is, an individual processor will schedule only every T time units. Per processor, this amounts to a $O(g/T)$ amortized scheduling cost.

Intra-group allocation At the intra-group level, the priority queue for *NVT* is accessed by all processors in a given group, and locking may prove to be too expensive if using a $O(\log P)$ complexity heap. Therefore, we use a linked list, with $O(P)$ scheduling complexity. This overhead is incurred by a processor only when it reaches the end of its runqueue. A quick calculation reveals that, in the case when $N >> P$, the queues are balanced (with less than N/P tasks per queue), and thus the amortized complexity of traversing the list of runqueues is no more than $P/(N/P) = P \cdot P/N$. $\square$

Clearly, δ and T provide a trade-off between accuracy and locking overhead. With small δ, intra-group allocation is kept tight, but processes move across queues more frequently. With small T, processor reallocation is more responsive to discrepancies in service allocated to clients of different groups. This keeps the error bounds down (Theorem 3.6), but is expensive: we want to reallocate processors as rarely as possible, since, besides the locking and time complexity overhead of the operation, moving a processor from one group to another migrates many processes among queues within those two respective groups. We expect that this will also hurt cache affinity.

Given the form of the error bounds in Theorem 3.6, it seems beneficial to choose δ and T on the same order. However, in practice, δ has a more pronounced effect on the error of a processes, whereas the error introduced by T is spread over many processes in the group. δ should be taken to be a small number. We found that $\delta = 4$ works well in practice.

We conclude with a few comments:

- The above bounds hold for static process mixes, where no arrivals or departures are expected. This is mostly to keep the analysis clean and compact, and because the measure of fairness used, the service error, is defined assuming that the process is eligible at all times to receive its due allocation. In terms of running time, as mentioned in Section 2.4, we incur a $O(P)$ cost to readjust weights, and a $O(g)$ cost to recompute the processor allocation of groups. Both g and P can be assumed to be constants.

- The inter-group *MFQ* allocator is a generalization of the simple *WFQ VFT* algorithm ([6]), and is thus a virtual finishing time scheduler. Interestingly, unlike in the uniprocessor case, using virtual start time instead would result in dramatically worse fairness properties (see Note in A.1). We have not attempted to adapt a more complicated, and accurate algorithm such as WF^{2}Q, since the number of groups, g, is a constant, and we are more concerned with the synchronization cost of more complex data structures than with the positive error bound.

4 Measurements and Results

To demonstrate the effectiveness of *GDQ*, we have implemented a prototype *GDQ* CPU scheduler in the Linux operating system and measured its performance. We present some experimental data quantitatively comparing *GDQ* performance against the standard Linux 2.4 and 2.6 kernel schedulers, and against the $O(1)$ GR^3 multiprocessor scheduler ([3]). We have also conducted extensive simulation studies to capture the service error bounds of *GDQ*.

4.1 Simulation Studies

We used a simulator for these studies for two reasons. First, our simulator enabled us to isolate the impact of the scheduling algorithm itself from other activities present in an actual kernel implementation. Second, our simulator enabled us to examine the scheduling behavior across hundreds of thousands of different combinations of processes with different weight values. It would have been much more difficult to obtain this volume of data in a repeatable fashion from just measurements of a kernel scheduler implementation.

The scheduling simulator measures the service time error, described in Section 3, of a scheduler on a synthetic set of processes. The simulator takes as input the scheduling algorithm, the number P of processors, and a process mix, consisting of a list of process weights.

The process mix is provided by a random mix generator, which, given the number of processes N, the total number of weights Φ_T, and an upper limit on the process weights (necessary for feasibility purposes), will generate a random list of process weights. The simulator then schedules the processes using the specified algorithm as a real scheduler would, and tracks the resulting service time error. The simulator runs the schedule, then computes the maximum (most positive) and minimum (most negative) service time error for the given set of processes and weight assignments. This process of random weight allocation and scheduler simulation is repeated for the specified number of process-weight combinations. We then compute the worst-case maximum service time error and worst-case minimum service time error for all processes during all specified number of process-weight combinations to obtain a "worst-case" error range.

To measure proportional fairness accuracy, we ran simulations for each scheduling algorithm on 24 different combinations of N and Φ_T: the number of number of processes ranges exponentially from 512 to 16384 and the total weight ranges exponentially from 32768 to 262144. For each pair (N, Φ_T), we ran 500 process-weight combinations and determined the resulting worst-case errors overall from the 500 runs.

We measured the service error for $P = 1, 2, 4, 8, 16, 32, 64, 128$. We used $\delta = 4$. To understand the effect of the inter-group time quantum T on the service error, we used $T = 20, 40, 80, 160, 320,$ and 640.

We found that the error of *GDQ* does not get worse as the number of processes increases with Φ_T kept constant, and, in fact, for most values of P the error actually improves as N grows, mainly because with more processes and a fixed total weight, the weight skew among processes becomes less accentuated.

Figure 2 shows that for fixed values of T, the negative error improves slightly as the number of processors increases, while the positive error gets slightly worse, but keeps well below the theoretical bound of Theorem 3.6. On one hand, the error ought to get better as P grows, since more queues are serviced simultaneously. On the other hand, increasing P means that the process weight upper bound decreases according to the feasibility constraint, and hence there will be more processes in the largest order group. This accentuates the weight skew among groups, causing the inter-group error (which depends on T) to increase. To put results in perspective, we include the error bounds obtained for the same simulations for the uniprocessor schedulers *SFQ* ([8]) and

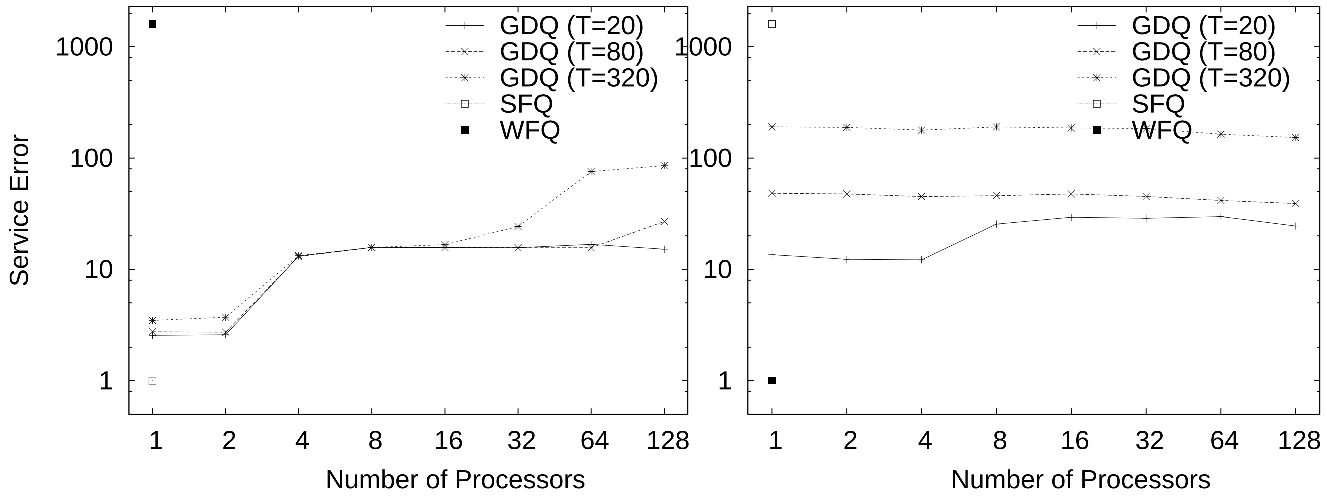

Figure 2: Service error for *GDQ* with $T = 40, 160, 640$ when P ranges from 1 to 128, and for *SFQ* and *WFQ* when $P = 1$. Both axes are logarithmic. *Left:* Positive error. *Right:* Absolute value of negative error.

WFQ ([11]). The overall error bounds for these common uniprocessor schedulers are substantially worse than GDQ for any number of processes.

4.2 Linux Kernel Measurements

We also conducted detailed measurements of real kernel scheduler performance by comparing our prototype *GDQ* Linux implementation against the the $O(1)$ GR^3 multiprocessor scheduler ([3]), as well as the standard single queue Linux 2.4 scheduler, and the distributed queue Linux 2.6 scheduler. In particular, comparing against the standard Linux scheduler and measuring its performance is important because of its growing popularity as a platform for server as well as desktop systems. The experiments we have done quantify the scheduling overhead of these schedulers in a real operating system environment.

We conducted a series of experiments on an 8-processor system to quantify how the scheduling overhead for each scheduler varies as the number of processes increases. Each process executed a simple microbenchmark which performed a few operations in a while loop. A control program was used to fork a specified number of processes, all having the same weight. Once all processes were runnable, we measured the execution time of each scheduling operation that occurred during a fixed time duration of 3 minutes. This was done by inserting a counter and timestamped event identifiers in the Linux scheduling framework. The measurements required two timestamps for each scheduling decision, so measurement errors of 70 cycles are possible due to measurement overhead. We performed these experiments on the standard Linux 2.4 and 2.6 schedulers, and on the the GR^3 and the *GDQ* prototypes, for up to 1000 running processes. The system was provisioned with either 2 or 8 CPUs. *GDQ* used $T = 20$.

As shown in Figure 3, the scheduling overheads of Linux 2.6 and *GDQ* are roughly constant as the number of processes grows. Both schedulers use $O(1)$ algorithms to pick the next process. The highly optimized Linux 2.6 scheduler (which, however, is not a proportional share scheduler) is about 10% faster than *GDQ*. *GR*^{3}, also a $O(1)$ scheduler, is worse than *GDQ* because it uses a single queue. Even for as few as 8 procesors, where there is little lock contention, the scheduler pays the memory latency cost for grabbing global locks. The Linux 2.4 scheduler has overhead linear in the number of processes, orders of magnitude larger than the schedulers plotted in Figure 3.

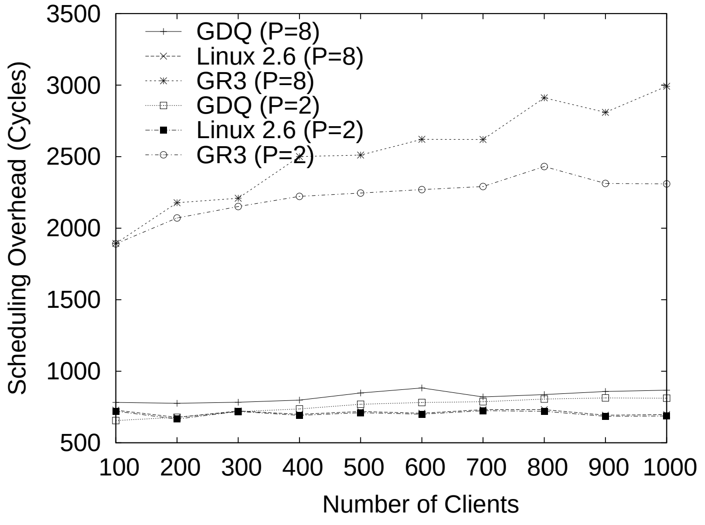

Figure 3: Experimental average scheduling overhead in CPU cycles for *GDQ*, *GR*^{3}, and Linux 2.6 on 2 and 8 processors.

To get a rough idea as to how the schedulers scale with the number of processors, we compare the scheduling overhead of *GDQ* Linux 2.6, and *GR*^{3}, using 2 and 8 processors. The centralized queue *GR*^{3} scheduler incurs much more overhead especially with 8 processors given its increased synchronization costs. Our *GDQ* prototype incurs slightly more overhead than the optimized Linux 2.6 scheduler, but provides the benefit of proportional sharing. Both schedulers demonstrate good scalability between 2 and 8 processors.

5 Related Work

Most commercial operating systems today sport multiprocessor schedulers of varying degrees of complexity. Most of these are built around heuristics that tackle the trade-offs between response time, throughput, and efficiency ([15]). These algorithms do not implement fair-share scheduling even on single processor machines, and rarely do they achieve at least long-term proportional sharing. On multiprocessor systems, Solaris 2.x uses a global dispatch queue from which it schedules processes. Recognizing the potential bottleneck inherent in such approaches, Digital UNIX keeps per-processor queues, and re-balances the queues regularly. The Linux 2.6 kernel is similar ([10]), except Linux uses the SVR4 priority arrays to help each processor schedule in constant time (under the assumption that there is a fixed, narrow range of allowable task weights). An interesting approach to processor allocation, somewhat related to *GDQ*, is taken by the Mach operating system. Mach allows applications to create *processor sets*, which contain a certain number or processors and threads. A processor in a set only works on threads within that set, but processors may move from set to set. The Mach motivation for grouping processors is flexibility in resource management and service guarantee, which *GDQ* achieves indirectly in the context of proportional sharing.

Because of the control it offers to resource allocation and its intuitive fairness model, proportional sharing has been widely studied and applied to both processor and network traffic scheduling. To relate *GDQ* to the very extensive literature on single resource proportional scheduling, we point the reader to [3]. The problem of proportional multiprocessor scheduling has received significantly less attention, mainly due the extra complications introduced by weight infeasibility, task parallelization, and inter-processor coordination. The solutions proposed so far rely on a single global queue for scheduling, and thus scale poorly. [4] presents *SFS*, a multiprocessor extension of the *SFQ* algorithm ([8]), which performs well in practice but has no theoretical fairness bounds. GR^3, introduced in [3], was the first scheduler providing strong fairness bounds with lower scheduling overhead. *SFS* introduced the notion of *feasible* tasks along with a $O(P)$-time weight readjustment algorithm, which requires however that the tasks be sorted by their original weight. By using its grouping strategy, GR^3 performs the same weight readjustment in $O(P)$ time without the need to order processes, thus avoiding the $O(\log N)$ overhead per maintenance operation. Since *GDQ* uses the same task grouping strategy as GR^3, it benefits from the same efficient weight readjustment algorithm.

In the context of link scheduling, [2] considers aggregated links, the analogue of multiprocessors for the network server problem, and presents a global queue algorithm that approximates the idealized fluid GPS model for multi-server systems. Their adaptation of *WFQ* ([11]), called *MSFQ*, and of WF^2Q ([1]), called MSF^2Q, preserve the error bounds of *WFQ* (-1 to N) and of WF^2Q (-1 to +1) respectively whenever the flows are backlogged at all times. Otherwise, with an unlucky interplay of busy periods, the service error can be as small as $-P$, as big as $+P$ for MSF^2Q. *MSFQ*'s positive error bound is presumably $N+P$ is such a case. We note that the $O(P)$ error is unavoidable in any system where flows are not backlogged at all times (correspondingly, tasks are not runnable at all times). Since the model of aggregated network links allows for packets of the same flow to be serviced at the same time, the single resource algorithms cannot be adapted in the same way to multiprocessor scheduling. Even for the case of the *GDQ* inter-group scheduler, where tasks of the same group may run in parallel, we could not use the algorithm of [2], as that would not guarantee a share of $\lfloor \frac{\Phi_G}{\Phi_T} P \rfloor$ or $\lceil \frac{\Phi_G}{\Phi_T} P \rceil$ to group G at all times (the smoothness property).

Multi-resource scheduling has been receiving some attention outside of the proportional sharing paradigm as well. In the periodic task model, [13] looks at providing fairness on multiprocessor systems and, while noting the benefits of distributed queues, contends that a global queue is simpler to design and implement. The authors review some approaches to periodic task scheduling on multiprocessor systems, in particular flavors of the p-fair scheduler, and extend the work on p-fair multiprocessor scheduling in [12]. [5] shows how to adapt a p-fair scheduler in a work-conserving operating system scheduler. [16] targets programmable network routers, and tries to efficiently use the very small instruction cache by keeping packets that use same code on the same processor, and processing them back-to-back. They note the conflicting requirement in scheduling packets to optimize delay or cache affinity. [14] considers cache affinity in relation to multiprocessor scheduling. Cache performance is an important factor of system performance, and a distributed queue approach has the implicit benefit of reusing much more cache state than a global queue design. Given that resources may be poorly used if allocated independently, [17] attempts to combine processor and link scheduling in programmable multiprocessor network routers. An operating system presents similar challenges. While *GDQ* considers cache performance as one of its design motivations, explicitly taking into account memory, disk, or network activity is beyond the scope of this process scheduling algorithm.

6 Conclusions and Future Work

We have designed, implemented, and evaluated Grouped Distributed Queues scheduling in the Linux operating system. We prove that *GDQ* is the first and only $O(1)$ distributed queue multiprocessor scheduling algorithm that guarantees a constant service error bound compared to an idealized processor sharing model, irrespective of the number of runnable processes. Previous approaches to multiprocessor scheduling used single, centralized queues, or relied on heuristics that did not even provide long-term fairness.

To achieve good proportional share fairness with low overhead, *GDQ* employs an exponential grouping strategy and uses a two-level hierarchical scheduler. *GDQ* introduces a new way to consider the pairing of processors and queues and presents a virtual-time-based inter-group scheduling algorithm with good fairness and smoothness properties. For each processor, *GDQ* uses a fair and efficient intra-queue round robin scheme.

We have measured the performance of *GDQ* using both simulations and kernel measurements of a prototype Linux implementation. Our simulation results show that *GDQ* can provide good proportional fairness behavior even as the number of processes exceeds 250,000. Our experimental results using our *GDQ* Linux implementation further demonstrate that *GDQ* provides accurate proportional fairness behavior on real applications with comparable scheduling overhead to the $O(1)$ Linux scheduler.

While *GDQ* is a distributed queue scheduler, there is a fair amount of communication and process exchange among processors and their queues. Re-balancing less often and using less information would have beneficial effects on the synchronization overhead.

A major advantage of distributed queue scheduling lies in benefiting from cache state by keeping processes on the same processor for a long time. Making caching explicit in the scheduler would be worthwhile.

References

[1] Jon C. R. Bennett and Hui Zhang. WF^{2}Q: Worst-Case Fair Weighted Fair Queueing. In *Proceedings of IEEE INFOCOM '96*, pages 120–128, San Francisco, CA, Mar. 1996.

[2] Josep M. Blanquer and Banu Ozden. Fair Queuing for Aggregated Multiple Links. In *Proceedings of ACM SIGCOMM '01*, pages 189–198, San Diego, CA, Aug. 2001.

[3] Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein, and Haoqiang Zheng. Group Ratio Round-Robin: $O(1)$ Proportional Share Scheduling for Uniprocessor and Multiprocessor Systems. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 337–352, Anaheim, CA, April 2005. USENIX.

[4] Abhishek Chandra, Micah Adler, Pawan Goyal, and Prashant J. Shenoy. Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors. In *Proceedings of the 4th Symposium on Operating System Design & Implementation*, pages 45–58, San Diego, CA, Oct. 2000.

[5] Abhishek Chandra, Micah Adler, and Prashant Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportionate fair scheduling in multiprocessor systems. In *RTAS '01: Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*, page 3, Washington, DC, USA, 2001. IEEE Computer Society.

[6] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM '89*, pages 1–12, Austin, TX, Sept. 1989.

[7] S. J. Golestani. A Self-Clocked Fair Queueing Scheme for Broadband Applications. In *Proceedings of IEEE INFOCOM '94*, pages 636–646, april 1994.

[8] P. Goyal, H. Vin, and H. Cheng. Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. *IEEE/ACM Transactions on Networking*, pages 690–704, Oct. 1997.

[9] L. Kleinrock. *Computer Applications*, volume II of *Queueing Systems*. John Wiley & Sons, New York, NY, 1976.

[10] Robert Love. *Linux Kernel Development*. SAMS, Developmer Library Series, first edition, 2004.

[11] 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, June 1993.

[12] A. Srinivasan and J. Anderson. Fair scheduling of dynamic task systems on multiprocessors, 2003.

[13] A. Srinivasan, P. Holman, J. Anderson, S. Baruah, and J. Kaur. *Network Processor Design: Issues and Practices Volume II*, chapter Multiprocessor Scheduling in Processor-based Router Platforms: Issues and Ideas. Morgan Kaufamann Publishers, 2004.

[14] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 24(2):139–151, 1995.

[15] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall, Upper Saddle River, NJ, 1996.

[16] T. Wolf and M. Franklin. Locality-aware predictive scheduling for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 152–159, Tucson, AZ, Nov. 2001.

[17] Yunkai Zhou and Harish Sethu. On Achieving Fairness in the Joint Allocation of Processing and Bandwidth Resources: Principles and Algorithms. Technical Report DU-CS-03-02, Drexel University, Jul. 2003.

A Appendix

A.1 Terminology

Term	Description	Term	Description
C_j	Process j.	ϕ_C	The weight assigned to process C.
ϕ_i	Shorthand notation for ϕ_{C_i}.	σ_C	The order of process C: $\lfloor \log \phi_C \rfloor$.
N	The number of runnable processes.	$\phi_{\max}$	Maximum possible weight.
G^k	Group of order k, $\{C : 2^k \le \phi_C < 2^{k+1}\}$.	g	The number of groups.
Φ_G	The group weight of G: $\sum_{C \in G} \phi_C$.	Φ_k	Shorthand notation for Φ_{G^k}.
N^k	Number of processes in group G^k.	P	Number of processors.
$\underline{P}^k$	$\left\lfloor \frac{\Phi_k}{\Phi_T} P \right\rfloor$.	$\overline{P}^k$	$\left\lceil \frac{\Phi_k}{\Phi_T} P \right\rceil$.
P^k	Number of processors assigned to group G^k.	$\wp_j$	jth processor.
$\wp_j^k$	jth processor of group G^k.	w_C	The work of process C.
nv_C	The *NVT* of process C: $w_C \frac{2^{\sigma_C}}{\phi_C}$.	W_G	The group work of group G.
W_k	Shorthand notation for W_{G^k}.	F_G	The *VFT* of group G.
F_k	Shorthand notation for F_{G^k}.	$\wp(Q)$	Processor assigned to queue Q
$Q(\wp)$	Queue that processor $\wp$ is servicing.	$C(Q)$	Current process in queue Q.
nv_Q	*NVT* of queue Q.	Φ_T	Total weight, $\sum_{j=1}^{N} \phi_j = \sum_{i=1}^{g} \Phi_i$.
W_T	Total work, $\sum_{j=1}^{N} w_j = \sum_{i=1}^{g} W_i$.	e_C	Service error of process C: $w_C - W_T \frac{\phi_C}{\Phi_T}$.
T	Inter-group time quantum.	δ	Maximum intra-group queue *NVT* difference.

Table 1: Summary of *GDQ* Terminology

A.2 Examples

We introduce a concrete example that we will use throughout to illustrate the operation of the *GDQ* scheduler: grouping, inter- and intra-group scheduling.

A.2.1 Grouping Strategy

Consider a mix of 10 processes, $C_1, C_2, \ldots C_{10}$ having the following weights:

$$\phi_1 = 2, \phi_2 = 2, \phi_3 = 3, \phi_4 = 4, \phi_5 = 4, \phi_6 = 4, \phi_7 = 4, \phi_8 = 6, \phi_9 = 6, \phi_{10} = 9.$$

Processes 1, 2 and 3, having weights 2 and 3, belong to the group of order 1, G^1. Processes 4 through 9, with weights between 2^2 and $2^3 - 1$ belong to the group of order 2, G^2, and process 10, with weight between 2^3 and $2^4 - 1$, belongs to G^3, the group of order 3.

GDQ makes sure to place the processes in their respective groups:

$$G^1 = \{C_1, C_2, C_3\},\ G^2 = \{C_4, C_5, C_6, C_7, C_8, C_9\},\ G^3 = \{C_{10}\}.$$

The group weights, computed as the sum of the weights of the processes in the group, are:

$$\Phi_1 = 2+2+3 = 7,\ \Phi_2 = 4+4+4+4+6+6 = 28,\ \Phi_3 = 9.$$

A.2.2 Inter-Group Allocation

Consider the process mix in the example of Section A.2.1, and assume we have a system with $P = 4$ processors. It is the goal of the inter-group scheduler to allocate processors to the three groups, G^1 of weight 7, G^2 of weight 28, and G^3 of weight 9 in proportion to their weight. The total weight is $7+28+9 = 44$. The first step of the inter-group algorithm is to identify the groups who should be allocated dedicated processors. In our case, $\underline{P}^1 = \lfloor \frac{7}{44}4 \rfloor = 0$, $\underline{P}^2 = \lfloor \frac{28}{44}4 \rfloor = 2$, and $\underline{P}^3 = \lfloor \frac{9}{44}4 \rfloor = 0$. Hence, two processors, $\wp^1$ and $\wp^2$, get dedicated to G^2, and its weight is readjusted to $28 - 2\frac{44}{4} = 6$. For the purpose of the inter-group algorithm, we thus have a client of weight $\hat{\Phi}_1 = 7$, one of weight $\hat{\Phi}_2 = 6$, and one of weight $\hat{\Phi}_3 = 9$ which compete for the remaining 2 processors ($\wp^3$ and $\wp^4$) that are not dedicated.

Assume that we use an inter-group time quantum equal to 20 time units ($T = 20$ tu), and assume that every $T/4 = 5$ time units, one of the 4 processors reschedules (runs the inter-group routine).

Consider some time t when $\wp^1$ reschedules, and assume at that time, the *VFT*s of the clients are $F_{G^1} = \frac{2}{7}$, $F_{G^2} = \frac{1}{6}$, and $F_{G^3} = \frac{2}{9}$. This corresponds to the case where G^1 and G^3 already completed a full inter-group time quantum each. Assume that at time t, G^2 has $\wp^3$ and G^3 has $\wp^4$.

Since $\wp^1$ is dedicated, it will do nothing at time t. Suppose 5 tu later, it is $\wp^3$'s turn to reschedule. The *VFT* of G^2 becomes $F_{G^2} = \frac{2}{6}$. The client with the least *VFT* is now G^3, whose *VFT* is $\frac{2}{9}$. However, since G^3 already has a processor assigned ($\wp^4$), we should allocate the processor to the client with the next smallest *VFT*, which is G^1. Note that, if the *VFT* of G^2 were smaller than that of G^1, then $\wp^3$ would continue to service G^2. In the present case though, G^1 receives $\wp^3$.

Suppose the next processor to reschedule is $\wp^2$, at time $t+10$. $\wp^2$ is dedicated, so nothing happens. 5 tu later, $\wp^4$ reschedules. The *VFT* of G^3 is incremented to $F_{G^3} = \frac{3}{9}$. The client with the least *VFT* is now G^2, whose *VFT* is $\frac{2}{6}$ (the tie is broken in favor of the lower order group). 5 tu later, at $t+20$ (thus, at $t+T$), it is again $\wp^1$'s turn, which, being dedicated, does not do anything. Next, at $t+25$, $\wp^3$ reschedules again. The *VFT* of G^1 is incremented to $F_{G^1} = \frac{3}{7}$. The under-allocated client with the minimum *VFT* is now G^3, whose *VFT* is $F_{G^3} = \frac{3}{9}$, and hence gets $\wp^3$.

Figure 4 illustrates the above scenario up to time $t+75$.

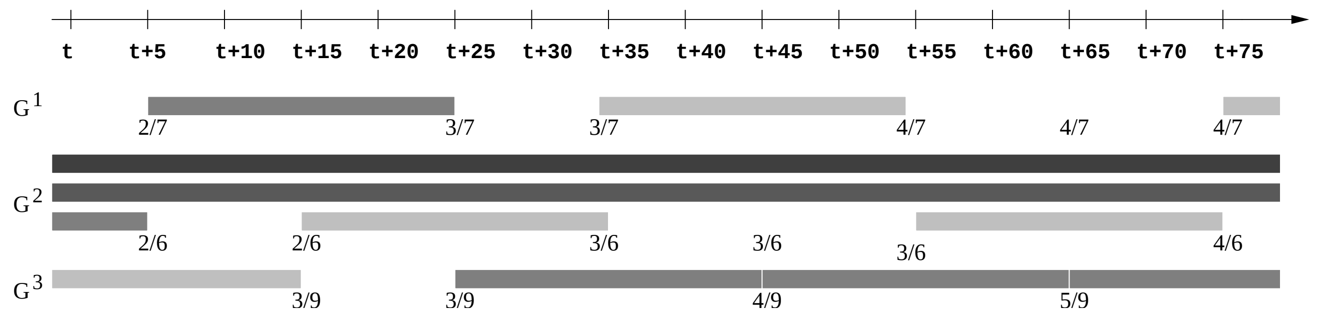

Figure 4: Inter-group scheduling example. The service intervals for each of the 4 processors are represented as solid bars, where $\wp^1$ and $\wp^2$ (the processors dedicated to G^2) are the darkest, $\wp^3$ is lighter, and $\wp^4$ is the lightest. Note that each interval has length equal to T, 20 in this example. Whenever a non-dedicated processor reschedules, we list the *VFT* for the eligible groups. The processor always gets assigned to the group of least *VFT*.

A.2.3 Intra-Group Allocation

Let us revisit the example developed in Sections A.2.1 and A.2.2. As far as the intra-group scheduler is concerned, the processor allocation of G^1 is either 0 or 1, that of G^2 is either 2 or 3, and that of G^3 is either 0 or 1. Whenever G^3 is assigned a processor, it serves the only process of G^3, which is C_{10}. The other time, C_{10} is stalled. Intra-group scheduling for G^1 is almost as simple, except there is a queue containing the 3 processes of G^1. While G^1 has a processor, this works round-robin (taking into account NVTs) on the queue. When G^1's processor allocation is 0, the queue becomes the stalled queue.

To illustrate the intra-group scheduler, we will focus on G^2, and use $\delta = 1$. Recall, G^2 has 2 dedicated processors ($\wp^1$ and $\wp^2$) and, following the scenario in the inter-group example (Section A.2.2), up to $t+5$ it will also have $\wp^3$. Between $t+5$ and $t+15$, it goes down to 2 processors, and at $t+15$, it receives $\wp^4$, bringing its allocation to 3 again.

Denote the $\overline{P}^2 = 3$ queues of G^2 as Q_1, Q_2, and Q_3. $\wp^1$ owns Q_1, $\wp^2$ owns Q_2, and Q_3 goes from $\wp^3$ to stalled to $\wp^4$ during the timespan under consideration. Arbitrarily, suppose that at time t the queues contain processes as follows: $Q_1 = \{C_4, C_5\}$, $Q_2 = \{C_7\}$, $Q_3 = \{C_8, C_9, C_6\}$. Recall the process weights: $\phi_4 = 4, \phi_5 = 4, \phi_6 = 4, \phi_7 = 4, \phi_8 = 6, \phi_9 = 6$.

Assume the NVTs of the processes of G^2 are all 0 at time t. All the queue NVTs are set to 1 at the beginning of the round, $nv_{Q_1} = nv_{Q_2} = nv_{Q_3} = 1$.

At time t, $\wp^3$ runs C_8 from Q_3, and increments its NVT to $4/6 = 2/3$. At the same time, $\wp^1$ runs C_4 from Q_1, and increments its NVT to $4/4 = 1$, while $\wp^2$ runs C_7 from Q_2, and increments its NVT to $4/4 = 1$.

At time $t+1$, $\wp^3$ runs C_8 again, and increments its NVT to $4/3$. At the same time, $\wp^1$ moves on to C_5 in Q_1, and increments its NVT to 1. Q_2 is at the end of a round, so $\wp^2$ increments nv_{Q_2} to 2 and continues to run C_7, incrementing its NVT to 2.

At time $t+2$, $\wp^3$ runs C_9, and increments its NVT to $2/3$. Q_1 is at the end of a round, so $\wp^1$ increments

nv_{Q_1} to 2 and runs C_4, incrementing its NVT to 2. Q_2 is once more at the end of a round, so it increments nv_{Q_2} to 3 and runs C_7 again, incrementing its NVT to 3.

At time $t+3$, $\wp^3$ runs C_9 again, and increments its NVT to $4/3$. $\wp^1$ runs C_5 and increments its NVT to 2. Q_2 is at the end of a round, and this time, $nv_{Q_2} > nv_{Q_3} + \delta$. Therefore, $\wp^2$ steals process C_6 from Q_3 and runs it, incrementing its NVT to 1.

At time $t+4$, $\wp^3$ is at the end of its round, increments nv_{Q_3} to 2, and runs C_8, whose NVT becomes $6/3$. $\wp^1$ is also at the end of a round, increments nv_{Q_1} to 3, runs C_4 and increments its NVT to 3. $\wp^2$ still runs process C_6, incrementing its NVT to 2, since its NVT is smaller than the queue NVT, which is 3.

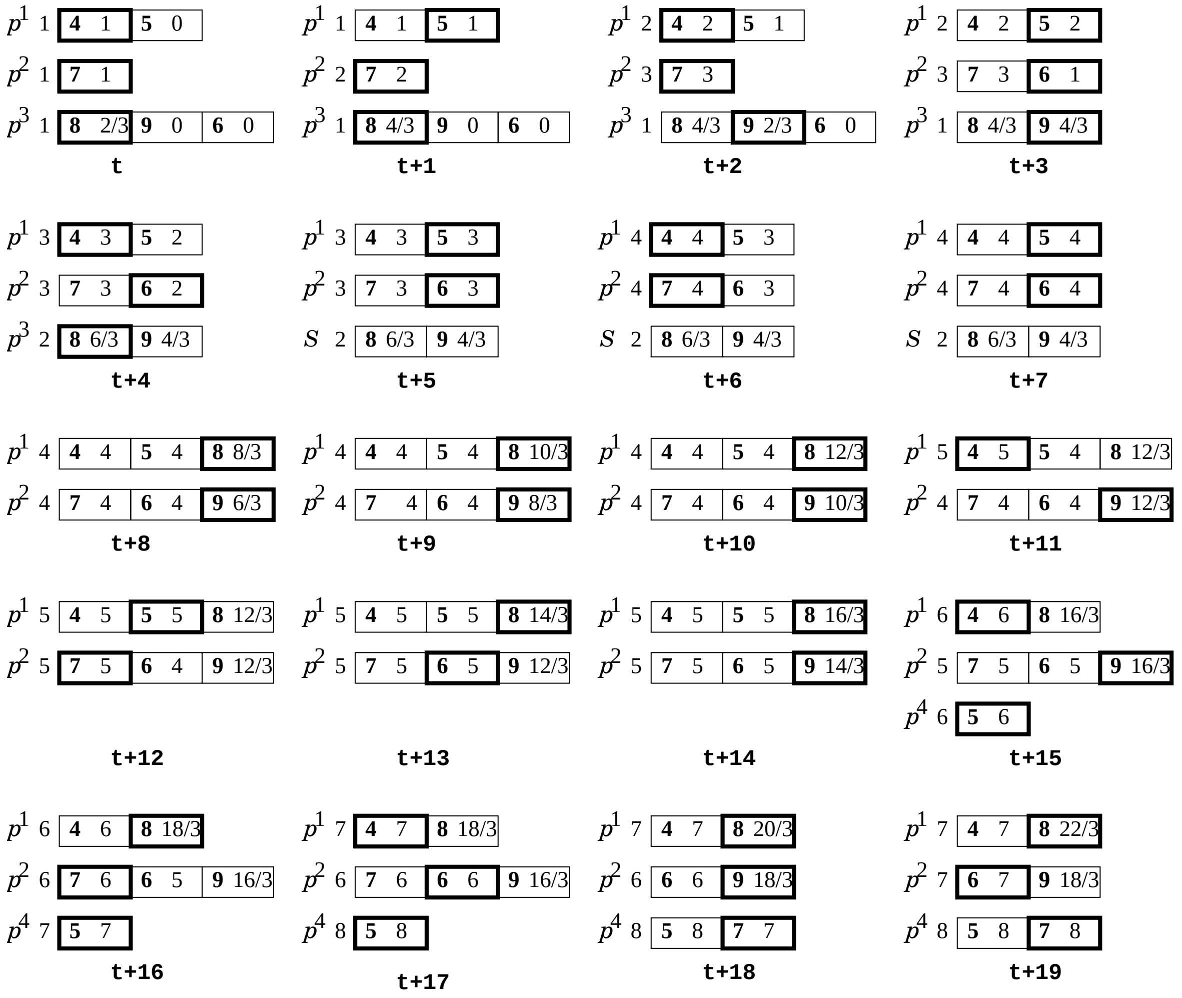

Figure 5: Intra-group scheduling example. The group considered is G^2, having processes C_4, C_5, C_6, and C_7 of weight 4, and C_8 and C_9 of weight 6. For each time unit, the three queues Q_1, Q_2, and Q_3 are displayed (unless Q_3 is empty). For each queue, we list the processor working on that queue ('S' means stalled), the current NVT of the queue, followed by the list of processes. Each process is represented as a box containing the process number (in bold face) and the process NVT.

At time $t+5$, processor $\wp^3$ is reassigned to G^1, so Q_3 becomes stalled. From time $t+5$ to $t+7$, $\wp^1$ and $\wp^2$ round-robin between C_5, C_4, and C_6, C_7 respectively.

At time $t+8$, both queues will be at the end of the round, with their queue NVTs being 4. This is more than $nv_{Q_3}+\delta = 2+1$, hence Q_1 steals C_8 and Q_2 next steals C_9 from the stalled queue, which becomes empty.

Up to time $t+14$, $\wp^1$ round-robins between C_4, C_5, C_8 and $\wp^2$ round-robins between C_6, C_7, C_9.

At time $t+15$, $\wp^4$ gets assigned to G^2, and will grab the next process of Q_1, which is C_5, and run it.

Later, at time $t+18$, Q_3's NVT will be 8 at the end of a round, while Q_2's NVT will be 6. $\wp^4$ will therefore grab the next process of Q_2, C_7 in this case. After this, the composition of the queues will be $Q_1 = \{C_4, C_8\}; Q_2 = \{C_6, C_9\}$, $Q_3 = \{C_5, C_7\}$. We notice that the queues have managed to arrive at an optimal weight balance for the given mix as the algorithm ran. Figure 5 illustrates this example.

A.3 Proofs

In the appendix we include all of the omitted proofs. We repeat the claims also.

Lemma 3.1. *Consider any scheduling point t, and let the client selected be j. If at point t, some client i is such that $\frac{w_i(t)+\delta_i}{\phi_i} < \frac{w_j(t)+1}{\phi_j}$ for some integer $\delta_i \geq 1$, then client i is running at point t, and will be scheduled for another δ_i intervals continuously.*

Proof. We use an inductive argument, noting that for the point $t=0$, the client j of least VFT is selected, so no other client i will satisfy $\frac{w_i+\delta_i}{\phi_i} < \frac{w_j+1}{\phi_j}$ with $\delta_i \geq 0$.

For the purpose of contradiction, let t be the smallest scheduling point such that the lemma does not hold, and let j be the client scheduled at point t. Also, consider the set I of clients i for which $\frac{w_i(t)+\delta_i}{\phi_i} < \frac{w_j(t)+1}{\phi_j}$, but which do not get scheduled for δ_i additional intervals. Let a be the client in I whose continuous run following point t is shortest. It makes sense to talk about such a shortest continuous run, because all clients in I are running at point t; otherwise, by the operation of the algorithm, they would be selected instead of client j.

Let δ'_a be the number of additional consecutive intervals run by client a following point t ($w_a(t') = w_a(t) + \delta'_a$). We have $0 \leq \delta'_a < \delta_a$ (since $a \in I$).

Let $t' > t + \delta'_a P$ be the scheduling point when client a stops running. Let b be the client selected at point t' (client b must have not been running at time t'). The following must then hold at time t':

$$\frac{w_b(t')+1}{\phi_b} \leq \frac{w_a(t')+1}{\phi_a}.$$

Let δ'_b be the number of intervals run by client b since point t ($w_b(t') = w_b(t) + \delta'_b$). Then we have

$$\frac{w_b(t)+\delta'_b+1}{\phi_b} \leq \frac{w_a(t)+\delta'_a+1}{\phi_a} \leq \frac{w_a(t)+\delta_a}{\phi_a} < \frac{w_j(t)+1}{\phi_j}$$

where the last inequality follows from $a \in I$. Hence, for client b, $\frac{w_b+\delta_b}{\phi_b} < \frac{w_j+1}{\phi_j}$ for $\delta_b = \delta'_b + 1$. Since client

b runs at most $\delta'_b < \delta_b$ intervals continuously after time t, it follows that client b satisfies the conditions for membership in set I. But client b stops running before client a does, contradicting the choice of a. $\square$

Lemma 3.2. $\frac{w_i+1}{\phi_i} \geq \frac{w_j}{\phi_j}$ *for any client i that is not running and for any client j.*

Proof. Assume otherwise, and let i and j be two clients and $\bar{t}$ some time moment such that $\frac{w_i(\bar{t})+1}{\phi_i} < \frac{w_j(\bar{t})}{\phi_j}$.

Let $t \leq \bar{t}$ be the last scheduling point when client j was scheduled, and let $t' \leq \bar{t}$ be the scheduling point when client i stopped running. Then

$$w_i(\bar{t}) = w_i(t') \text{ and } w_j(\bar{t}) = w_j(t) + 1.$$

If $t' < t$, $w_i(t) = w_i(t') = w_i(\bar{t})$ and then $\frac{w_i(t)+1}{\phi_i} < \frac{w_j(t)+1}{\phi_j}$, implying that client i would be selected instead of j at time t. Hence, $t \leq t'$.

Let δ'_i be the number of intervals that client i ran since time t ($w_i(t') = w_i(t) + \delta'_i$). Denote $\delta_i = \delta'_i + 1$. At time t,

$$\frac{w_i(t)+\delta_i}{\phi_i} = \frac{w_i(t)+\delta'_i+1}{\phi_i} = \frac{w_i(t')+1}{\phi_i} = \frac{w_i(\bar{t})+1}{\phi_i} < \frac{w_j(\bar{t})}{\phi_j} = \frac{w_j(t)+1}{\phi_j}.$$

Hence, by Lemma 3.1, client i would run at least δ consecutive intervals, contradicting that it stops after $\delta'_i < \delta_i$. $\square$

Lemma 3.3. *For any client i not currently running, $e_i \geq -1$.*

Proof.

$$\begin{aligned}
\frac{w_i+1}{\phi_i} &\geq \frac{w_j}{\phi_j} \ \forall j = 1 \ldots N \iff \\
(w_i+1)\phi_j &\geq w_j\phi_i \ \forall j = 1 \ldots N \implies \\
(w_i+1)\sum_{j=1}^{N}\phi_j &\geq \phi_i \sum_{j=1}^{N} w_j \iff \\
(w_i+1)\Phi_T &\geq \phi_i W_T \iff \\
w_i &\geq -1 + W_T\frac{\phi_i}{\Phi_T} \iff \\
e_i &\geq -1
\end{aligned}$$

$\square$

Lemma 3.4. *For any client i, $e_i \geq -1$.*

Proof. For clients that are not running, the lemma above establishes the result.

If client i is running, it will continuously run for only a bounded amount of time, since eventually its *VFT* will exceed the *VFT* of some other client (under the assumption that it has weight less than $1/P$ of the total weight). If the client has weight exactly $1/P$, then the client will stay on the processor indefinitely.

At the time the client stops running, the above lemma is again applicable. Thus, if we let t_1 and t_2 denote the endpoints of the interval when client i runs continuously ($t_2 \in \mathbb{N} \cup \{\infty\}$), we have $e_i(t_1) \geq -1$ and $e_i(t_2) \geq -1$. For any $t \in (t_1, t_2)$, we can express $e_i(t)$ as $e_1(t_1) + w_i(t) - w_i(t_1) - \frac{\phi_i}{\Phi_T} P(t - t_1)$.

Since client i receives a processor for the entire interval $[t_1, t]$, we have $w_i(t) - w_i(t_1) = t - t_1$ and since $\frac{\phi_i}{\Phi_T} \leq \frac{1}{P}$, we have

$$e_i(t) \geq -1 + t - t_1 - \frac{1}{P} P(t - t_1) \geq -1.$$

□

We conclude with a remark on the choice of virtual finishing time versus virtual start time in the *MFQ* scheduler:

Note A.1. *The MFQ consciously uses virtual finishing times (VFT) instead of virtual start times (VST). While VFT preserves the -1 negative error bound from the uniprocessor case, VST would not preserve its +1 positive error bound if used similarly. Thus, the analogue of Lemma 3.2 for virtual start time is not true. If virtual start times had been used, then we would not have been able to obtain a +1 positive error bound, as the following example illustrates:*

Consider N clients, with $\phi_1 = N - 2, \phi_2 = (N - 2)/2, \phi_i = 1 \; \forall i > 2$. *Then for the first* $(N - 2)/2$ *time steps, the two processors run clients* $3 \ldots N$ *for a time unit each. Then, for another* $(N - 2)/2$ *time steps, one processor runs client 1, and the other runs client 2. The work of client 2 at time* $N - 2$ *is then* $(N - 2)/2$*, whereas it should have received only* $\frac{(N-2)/2}{N-2+(N-2)/2+N-2} 2(N - 2) = \frac{2}{5}(N - 2)$. *The positive error is* $(N - 2)/10 = \Omega(N)$.