Group Ratio Round-Robin: O(1) Proportional Share Scheduling for Uniprocessor and Multiprocessor Systems

Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein[*]*, and Haoqiang Zheng*
Department of Computer Science
Columbia University
Email: {bc2008, wc164, nieh, cliff, hzheng}@cs.columbia.edu

Abstract

We present Group Ratio Round-Robin (GR^3), the first proportional share scheduler that combines accurate proportional fairness scheduling behavior with $O(1)$ scheduling overhead on both uniprocessor and multiprocessor systems. GR^3 uses a simple grouping strategy to organize clients into groups of similar processor allocations which can be more easily scheduled. Using this strategy, GR^3 combines the benefits of low overhead round-robin execution with a novel ratio-based scheduling algorithm. GR^3 introduces a novel frontlog mechanism and weight readjustment algorithm to operate effectively on multiprocessors. GR^3 provides fairness within a constant factor of the ideal generalized processor sharing model for client weights with a fixed upper bound and preserves its fairness properties on multiprocessor systems. We have implemented GR^3 in Linux and measured its performance. Our experimental results show that GR^3 provides much lower scheduling overhead and much better scheduling accuracy than other schedulers commonly used in research and practice.

1 Introduction

Proportional share resource management provides a flexible and useful abstraction for multiplexing processor resources among a set of clients. Proportional share scheduling has a clear colloquial meaning: given a set of clients with associated weights, a proportional share scheduler should allocate resources to each client in proportion to its respective weight. However, developing processor scheduling mechanisms that combine good proportional fairness scheduling behavior with low scheduling overhead has been difficult to achieve in practice. For many proportional share scheduling mechanisms, the time to select a client for execution grows with the number of clients. For server systems which may service large numbers of clients, the scheduling overhead of algorithms whose complexity grows linearly with the number of clients can waste more than 20 percent of system resources [3] for large numbers of clients. Furthermore, little work has been done to provide proportional share scheduling on multiprocessor systems, which are increasingly common especially in small-scale configurations with two or four processors. Over the years, a number of scheduling mechanisms have been proposed, and much progress has been made. However, previous mechanisms have either superconstant overhead or less-than-ideal fairness properties.

We introduce Group Ratio Round-Robin (GR^3), the first proportional share scheduler that provides constant fairness bounds on proportional sharing accuracy with $O(1)$ scheduling overhead for both uniprocessor and small-scale multiprocessor systems. In designing GR^3, we observed that accurate, low-overhead proportional sharing is easy to achieve when scheduling a set of clients with equal processor allocations, but is harder to do when clients require very different allocations. Based on this observation, GR^3 uses a simple client grouping strategy to organize clients into groups of similar processor allocations which can be more easily scheduled. Using this grouping strategy, GR^3 combines the benefits of low overhead round-robin execution with a novel ratio-based scheduling algorithm.

GR^3 uses the same basic uniprocessor scheduling algorithm for multiprocessor scheduling by introducing the notion of a frontlog. On a multiprocessor system, a client may not be able to be scheduled to run on a processor because it is currently running on another processor. To preserve its fairness properties, GR^3 keeps track of a frontlog per client to indicate when the client was already running but could have been scheduled to run on another processor. It then assigns the client a time quantum that is added to its allocation on the processor it is running on. The frontlog ensures that a client receives its proportional share allocation while also taking advantage of any cache affinity by continuing to run the client on the same processor.

GR^3 provides a simple weight readjustment algorithm that takes advantage of its grouping strategy. On a multiprocessor system, proportional sharing is not feasible for some client weight assignments, such as having one client with weight 1 and another with weight 2 on a

[*] also in Department of IEOR

two-processor system. By organizing clients with similar weights into groups, GR^3 adjusts for infeasible weight assignments without the need to order clients, resulting in lower scheduling complexity than previous approaches [7].

We have analyzed GR^3 and show that with only $O(1)$ overhead, GR^3 provides fairness within $O(g^2)$ of the ideal Generalized Processor Sharing (GPS) model [16], where g, the number of groups, grows at worst logarithmically with the largest client weight. Since g is in practice a small constant, GR^3 effectively provides constant fairness bounds with only $O(1)$ overhead. Moreover, we show that GR^3 uniquely preserves its worst-case time complexity and fairness properties for multiprocessor systems.

We have implemented a prototype GR^3 processor scheduler in Linux, and compared it against uniprocessor and multiprocessor schedulers commonly used in practice and research, including the standard Linux scheduler [2], Weighted Fair Queueing (WFQ) [11], Virtual-Time Round-Robin (VTRR) [17], and Smoothed Round-Robin (SRR) [9]. We have conducted both simulation studies and kernel measurements on micro-benchmarks and real applications. Our results show that GR^3 can provide more than an order of magnitude better proportional sharing accuracy than these other schedulers, in some cases with more than an order of magnitude less overhead. These results demonstrate that GR^3 can in practice deliver better proportional share control with lower scheduling overhead than these other approaches. Furthermore, GR^3 is simple to implement and easy to incorporate into existing scheduling frameworks in commodity operating systems.

This paper presents the design, analysis, and evaluation of GR^3. Section 2 describes the uniprocessor scheduling algorithm. Section 3 describes extensions for multiprocessor scheduling, which we refer to as GR^3MP. Section 4 analyzes the fairness and complexity of GR^3. Section 5 presents experimental results. Section 6 discusses related work. Finally, we present some concluding remarks and directions for future work.

2 GR^3 Uniprocessor Scheduling

Uniprocessor scheduling, the process of scheduling a time-multiplexed resource among a set of clients, has two basic steps: 1) order the clients in a queue, 2) run the first client in the queue for its *time quantum*, which is the maximum time interval the client is allowed to run before another scheduling decision is made. We refer to the units of time quanta as time units (tu) rather than an absolute time measure such as seconds. A scheduler can therefore achieve proportional sharing in one of two ways. One way, often called fair queueing [11, 18, 28, 13, 24, 10] is to adjust the frequency that a client is selected to run by adjusting the position of the client in the queue so that it ends up at the front of the queue more or less often. However, adjusting the client's position in the queue typically requires sorting clients based on some metric of fairness, and has a time complexity that grows with the number of clients. The other way is to adjust the size of a client's time quantum so that it runs longer for a given allocation, as is done in weighted round-robin (WRR). This is fast, providing constant time complexity scheduling overhead. However, allowing a client to monopolize the resource for a long period of time results in extended periods of unfairness to other clients which receive no service during those times. The unfairness is worse with skewed weight distributions.

GR^3 is a proportional share scheduler that matches with $O(1)$ time complexity of round-robin scheduling but provides much better proportional fairness guarantees in practice. At a high-level, the GR^3 scheduling algorithm can be briefly described in three parts:

1. **Client grouping strategy**: Clients are separated into groups of clients with similar weight values. The group of order k is assigned all clients with weights between 2^k to $2^{k+1} - 1$, where $k \geq 0$.

2. **Intergroup scheduling**: Groups are ordered in a list from largest to smallest group weight, where the group weight of a group is the sum of the weights of all clients in the group. Groups are selected in a round-robin manner based on the ratio of their group weights. If a group has already been selected more than its proportional share of the time, move on to the next group in the list. Otherwise, skip the remaining groups in the group list and start selecting groups from the beginning of the group list again. Since the groups with larger weights are placed first in the list, this allows them to get more service than the lower-weight groups at the end of the list.

3. **Intragroup scheduling**: From the selected group, a client is selected to run in a round-robin manner that accounts for its weight and previous execution history.

Using this client grouping strategy, GR^3 separates scheduling in a way that reduces the need to schedule entities with skewed weight distributions. The client grouping strategy limits the number of groups that need to be scheduled since the number of groups grows at worst logarithmically with the largest client weight. Even a very large 32-bit client weight would limit the number of groups to no more than 32. The client grouping strategy also ensures that all clients within a group have weight within a factor of two. As a result, the intragroup scheduler never needs to schedule clients with skewed weight distributions. GR^3 groups are simple lists that do not need to be balanced; they do not require any use of more complex balanced tree structures.

2.1 GR^3 Definitions

We now define the state GR^3 associates with each client and group, and describe in detail how GR^3 uses

C_j	Client j. (also called 'task' j)
ϕ_C	The weight assigned to client C.
ϕ_j	Shorthand notation for ϕ_{C_j}.
D_C	The deficit of C.
N	The number of runnable clients.
g	The number of groups.
G_i	i'th group in the list ordered by weight.
$\|G\|$	The number of clients in group G.
$G(C)$	The group to which C belongs.
Φ_G	The group weight of G: $\sum_{C \in G} \phi_C$.
Φ_i	Shorthand notation for Φ_{G_i}.
σ_G	The order of group G.
$\phi_{\min}^G$	Lower bound for client weights in G: 2^{σ_G}.
w_C	The work of client C.
w_j	Shorthand notation for w_{C_j}.
W_G	The group work of group G.
W_i	Shorthand notation for W_{G_i}.
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 client C: $w_C - W_T \frac{\phi_C}{\Phi_T}$
E_G	Group service error of G: $W_G - W_T \frac{\Phi_G}{\Phi_T}$
$e_{C,G}$	Group-relative service error of client C with respect to group G: $w_C - W_G \frac{\phi_C}{\Phi_G}$

Table 1: GR^3 terminology

that state to schedule clients. Table 1 lists terminology we use. For each client, GR^3 maintains the following three values: weight, deficit, and run state. Each client receives a resource allocation that is directly proportional to its *weight*. A client's *deficit* tracks the number of remaining time quanta the client has not received from previous allocations. A client's *run state* indicates whether or not it can be executed. A client is *runnable* if it can be executed.

For each group, GR^3 maintains the following four values: group weight, group order, group work, and current client. The *group weight* is the sum of the corresponding weights of the clients in the group run queue. A group with *group order* k contains the clients with weights between 2^k to $2^{k+1} - 1$. The *group work* is the total execution time clients in the group have received. The *current client* is the most recently scheduled client in the group's run queue.

GR^3 also maintains the following scheduler state: time quantum, group list, total weight, and current group. The *group list* is a sorted list of all groups containing runnable clients ordered from largest to smallest group weight, with ties broken by group order. The *total weight* is the sum of the weights of all runnable clients. The *current group* is the most recently selected group in the group list.

2.2 Basic GR^3 Algorithm

We initially only consider runnable clients in our discussion of the basic GR^3 scheduling algorithm. We discuss dynamic changes in a client's run state in Section 2.3. We first focus on the GR^3 intergroup scheduling algorithm, then discuss the GR^3 intragroup scheduling algorithm.

The GR^3 **intergroup scheduling** algorithm uses the ratio of the group weights of successive groups to determine which group to select. The next group to schedule is selected using only the state of successive groups in the group list. Given a group G_i whose weight is x times larger than the group weight of the next group G_{i+1} in the group list, GR^3 will select group G_i x times for every time that it selects G_{i+1} in the group list to provide proportional share allocation among groups.

To implement the algorithm, GR^3 maintains the total work done by group G_i in a variable W_i. An index i to tracks the current group and is initialized to 1. The scheduling algorithm then executes the following simple routine:

```
INTERGROUP-SCHEDULE()
1  C ← INTRAGROUP-SCHEDULE(G_i)
2  W_i ← W_i + 1
3  if i < g  and  (W_i+1)/(W_i+1+1) > Φ_i/Φ_i+1        (1)
4     then i ← i + 1
5     else i ← 1
6  return C
```

Let us negate (1) under the form:

$$\frac{W_i + 1}{\Phi_i} \leq \frac{W_{i+1} + 1}{\Phi_{i+1}} \tag{2}$$

We will call this relation the *well-ordering condition* of two consecutive groups. GR^3 works to maintain this condition true at all times. The intuition behind (2) is that we would like the ratio of the work of G_i and G_{i+1} to match the ratio of their respective group weights after GR^3 has finished selecting both groups. Recall, $\Phi_i \geq \Phi_{i+1}$. Each time a client from G_{i+1} is run, GR^3 would like to have run Φ_i/Φ_{i+1} worth of clients from G_i. (1) says that GR^3 should not run a client from G_i and increment G_i's group work if it will make it impossible for G_{i+1} to catch up to its proportional share allocation by running one of its clients once.

To illustrate how intergroup scheduling works, Figure 1 shows an example with three clients C_1, C_2, and C_3, which have weights of 5, 2, and 1, respectively. The GR^3 grouping strategy would place each C_i in group G_i, ordering the groups by weight: G_1, G_2, and G_3 have orders 2, 1 and 0 and weights of 5, 2, and 1 respectively. In this example, each group has only one client so there is no intragroup scheduling. GR^3 would start by selecting group G_1, running client C_1, and incrementing W_1. Based on (1), $\frac{W_1+1}{W_2+1} = 2 < \frac{\Phi_1}{\Phi_2} = 2.5$, so GR^3 would select G_1 again and run client C_1. After running C_1, G_1's work would be 2 so that the inequality in (1) would hold and GR^3 would then move on to the next group G_2 and run client C_2. Based on (1), $\frac{W_2+1}{W_3+1} = 2 \leq \frac{\Phi_2}{\Phi_3} = 2$, so GR^3 would reset

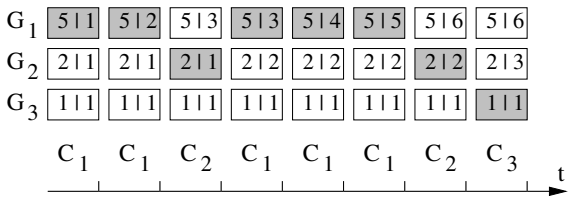

Figure 1: GR^3 intergroup scheduling. At each time step, the shaded box contains the pair $\Phi_G \mid W_G + 1$ for the group G before it is selected.

the current group to the largest weight group G_1 and run client C_1. Based on (1), C_1 would be run for three time quanta before selecting G_2 again to run client C_2. After running C_2 the second time, W_2 would increase such that $\frac{W_2+1}{W_3+1} = 3 > \frac{\Phi_2}{\Phi_3} = 2$, so GR^3 would then move on to the last group G_3 and run client C_3. The resulting schedule would then be: G_1, G_1, G_2, G_1, G_1, G_1, G_2, G_3. Each group therefore receives its proportional allocation in accordance with its respective group weight.

The GR^3 **intragroup scheduling** algorithm selects a client from the selected group. All clients within a group have weights within a factor of two, and all client weights in a group G are normalized with respect to the minimum possible weight, $\phi^G_{\min} = 2^{\sigma_G}$, for any client in the group. GR^3 then effectively traverses through a group's queue in round-robin order, allocating each client its normalized weight worth of time quanta. GR^3 keeps track of subunitary fractional time quanta that cannot be used and accumulates them in a deficit value for each client. Hence, each client is assigned either one or two time quanta, based on the client's normalized weight and its previous allocation.

More specifically, the GR^3 intragroup scheduler considers the scheduling of clients in rounds. A *round* is one pass through a group G's run queue of clients from beginning to end. The group run queue does not need to be sorted in any manner. During each round, the GR^3 intragroup algorithm considers the clients in round-robin order and executes the following simple routine:

INTRAGROUP-SCHEDULE(G)

```
1  C ← G[k]          ▷ k is the current position in the round
2  if D_C < 1
3     then k ← (k + 1) mod |G|
4          C ← G[k]
5          D_C ← D_C + φ_C/φ^G_min
6  D_C ← D_C − 1
7  return C
```

For each runnable client C, the scheduler determines the maximum number of time quanta that the client can be selected to run in this round as $\lfloor \frac{\phi_C}{\phi^G_{\min}} + D_C(r-1) \rfloor$. $D_C(r)$, the deficit of client C after round r, is the time quantum fraction left over after round r: $D_C(r) = \frac{\phi_C}{\phi^G_{\min}} + D_C(r-1) - \lfloor \frac{\phi_C}{\phi^G_{\min}} + D_C(r-1) \rfloor$, with $D_C(0) = \frac{\phi_C}{\phi^G_{\min}}$. Thus, in each round, C is allotted one time quantum plus any additional leftover from the previous round, and $D_C(r)$ keeps track of the amount of service that C missed because of rounding down its allocation to whole time quanta. We observe that $0 \leq D_C(r) < 1$ after any round r so that any client C will be allotted one or two time quanta. Note that if a client is allotted two time quanta, it first executes for one time quantum and then executes for the second time quantum the next time the intergroup scheduler selects its respective group again (in general, following a timespan when clients belonging to other groups get to run).

To illustrate how GR^3 works with intragroup scheduling, Figure 2 shows an example with six clients C_1 through C_6 with weights 12, 3, 3, 2, 2, and 2, respectively. The six clients will be put in two groups G_1 and G_2 with respective group order 1 and 3 as follows: $G_1 = \{C_2, C_3, C_4, C_5, C_6\}$ and $G_2 = \{C_1\}$. The weight of the groups are $\Phi_1 = \Phi_2 = 12$. GR^3 intergroup scheduling will consider the groups in this order: G_1, G_2, G_1, G_2, G_1, G_2, G_1, G_2, G_1, G_2, G_1, G_2. G_2 will schedule client C_1 every time G_2 is considered for service since it has only one client. Since $\phi^{G_1}_{\min} = 2$, the normalized weights of clients C_2, C_3, C_4, C_5, and C_6 are 1.5, 1.5, 1, 1, and 1, respectively. In the beginning of round 1 in G_1, each client starts with 0 deficit. As a result, the intragroup scheduler will run each client in G_1 for one time quantum during round 1. After the first round, the deficit for C_2, C_3, C_4, C_5, and C_6 are 0.5, 0.5, 0, 0, and 0. In the beginning of round 2, each client gets another $\phi_i/\phi^{G_1}_{\min}$ allocation, plus any deficit from the first round. As a result, the intragroup scheduler will select clients C_2, C_3, C_4, C_5, and C_6 to run in order for 2, 2, 1, 1, and 1 time quanta, respectively, during round 2. The resulting schedule would then be: C_2, C_1, C_3, C_1, C_4, C_1, C_5, C_1, C_6, C_1, C_2, C_1, C_2, C_1, C_3, C_1, C_3, C_1, C_4, C_1, C_5, C_1, C_6, C_1.

2.3 GR^3 Dynamic Considerations

We now discuss how GR^3 allows clients to be dynamically created, terminated, or change run state. Runnable clients can be selected for execution by the scheduler, while clients that are not runnable cannot. With no loss of generality, we assume that a client is created before it can become runnable, and a client becomes not runnable before it is terminated. As a result, client creation and termination have no effect on the GR^3 run queues.

When a client C with weight ϕ_C becomes runnable, it is inserted into group $G = G(C)$ such that ϕ_C is between 2^{σ_G} and $2^{\sigma_G+1} - 1$. If the group was previously empty, a new group is created, the client becomes the current client of the group, and g, the number of groups, is incremented. If the group was not previously empty, GR^3 inserts the client into the respective group's run queue right before the current client; it will be serviced after all of the other clients

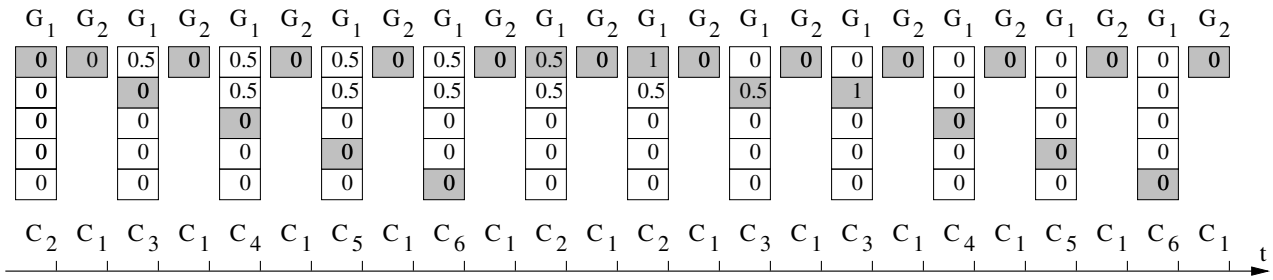

Figure 2: GR^3 intragroup scheduling. At each time step, the shaded box contains the deficit of the client before it is run.

in the group have first been considered for scheduling. The initial deficit D_C will be initialized to 0.

When a newly runnable client C is inserted into its respective group G, the group needs to be moved to its new position on the ordered group list based on its new group weight. Let this new position be k. The corresponding group work and group weight of G need to be updated and the client's deficit needs to be initialized. The group weight is simply incremented by the client's weight. We also want to scale the group work of G such that the work ratio of consecutive groups will continue to be proportional to their weight ratio:

$$W_G = \begin{cases} \left\lfloor (W_{k+1}+1)\frac{\Phi_G}{\Phi_{k+1}} \right\rfloor - 1 & \text{if } k < g \\ \left\lceil (W_{g-1}+1)\frac{\Phi_G}{\Phi_{g-1}} \right\rceil - 1 & \text{if } k = g \end{cases}$$

We will motivate these equations when analyzing the fairness of the algorithm in Section 4, but intuitively, we want to preserve the invariants that result from (2).

When a client C with weight ϕ_C becomes not runnable, we need to remove it from the group's run queue. This requires updating the group's weight, which potentially includes moving the group in the ordered group list, as well as adjusting the measure of work received according to the new processor share of the group. This can be achieved in several ways. GR^3 is optimized to efficiently deal with the common situation when a blocked client may rapidly switch back to the runnable state again. This approach is based on "lazy" removal, which minimizes overhead associated with adding and removing a client, while at the same time preserving the service rights and service order of the runnable clients. Since a client blocks when it is running, we know that it will take another full intragroup round before the client will be considered again. The only action when a client blocks is to set a flag on the client, marking it for removal. If the client becomes runnable by the next time it is selected, we reset the flag and run the client as usual. Otherwise, we remove the client from $G(C)$. In the latter situation, as in the case of client arrivals, the group may need to be moved to a new position on the ordered group list based on its new group weight. The corresponding group weight is updated by subtracting the client's weight from the group weight. The corresponding group work is scaled by the same rules as for client insertion, depending on the new position of the group and its next neighbor. After performing these removal operations, GR^3 resumes scheduling from the largest weight group in the system.

Whenever a client C blocks during round r, we set $D_C(r) = \min(D_C(r-1) + \phi_C/\phi_{\min}^{G(C)} - \lceil w \rceil, 1)$, where w is the service that the client received during round r until it blocked. This preserves the client's credit in case it returns by the next round, while also limiting the deficit to 1 so that a client cannot gain credit by blocking. However, the group consumes 1 tu (its work is incremented) no matter how long the client runs. Therefore, the client forfeits its extra credit whenever it is unable to consume its allocation.

If the client fails to return by the next round, we may remove it. Having kept the weight of the group to the old value for an extra round has no adverse effects on fairness, despite the slight increase in service seen by the group during the last round. By scaling the work of the group and rounding up, we determine its future allocation and thus make sure the group will not have received undue service. We also immediately resume the scheduler from the first (largest) group in the readjusted group list, so that any minor discrepancies caused by rounding may be smoothed out by a first pass through the group list.

3 GR^3 Multiprocessor Extensions (GR^3MP)

We now present extensions to GR^3 for scheduling a P-way multiprocessor system from a single, centralized queue. This simple scheme, which we refer to as GR^3MP, preserves the good fairness and time complexity properties of GR^3 in small-scale multiprocessor systems, which are increasingly common today, even in the form of multi-core processors. We first describe the basic GR^3MP scheduling algorithm, then discuss dynamic considerations. Table 2 lists terminology we use. To deal with the problem of infeasible client weights, we then show how GR^3MP uses its grouping strategy in a novel weight readjustment algorithm.

3.1 Basic GR^3MP Algorithm

GR^3MP uses the same GR^3 data structure, namely an ordered list of groups, each containing clients whose weights are within a factor of two from each other. When a

P	Number of processors.
$\wp^k$	Processor k.
$C(\wp)$	Client running on processor $\wp$.
F_C	Frontlog for client C.

Table 2: GR^3MP terminology

processor needs to be scheduled, GR^3MP selects the client that would run next under GR^3, essentially scheduling multiple processors from its central run queue as GR^3 schedules a single processor. However, there is one obstacle to simply applying a uniprocessor algorithm on a multiprocessor system. Each client can only run on one processor at any given time. As a result, GR^3MP cannot select a client to run that is already running on another processor even if GR^3 would schedule that client in the uniprocessor case. For example, if GR^3 would schedule the same client consecutively, GR^3MP cannot schedule that client consecutively on another processor if it is still running.

To handle this situation while maintaining fairness, GR^3MP introduces the notion of a **frontlog**. The frontlog F_C for some client C running on a processor $\wp^k$ ($C = C(\wp^k)$) is defined as the number of time quanta for C accumulated as C gets selected by GR^3 and cannot run because it is already running on $\wp^k$. The frontlog F_C is then queued up on $\wp^k$.

Given a client that would be scheduled by GR^3 but is already running on another processor, GR^3MP uses the frontlog to assign the client a time quantum now but defer the client's use of it until later. Whenever a processor finishes running a client for a time quantum, GR^3MP checks whether the client has a non-zero frontlog, and, if so, continues running the client for another time quantum and decrements its frontlog by one, without consulting the central queue. The frontlog mechanism not only ensures that a client receives its proportional share allocation, it also takes advantage of any cache affinity by continuing to run the client on the same processor.

When a processor finishes running a client for a time quantum and its frontlog is zero, we call the processor *idle*. GR^3MP schedules a client to run on the idle processor by performing a GR^3 scheduling decision on the central queue. If the selected client is already running on some other processor, we increase its frontlog and repeat the GR^3 scheduling, each time incrementing the frontlog of the selected client, until we find a client that is not currently running. We assign this client to the idle processor for one time quantum. This description assumes that there are least $P+1$ clients in the system. Otherwise, scheduling is easy: an idle processor will either run the client it just ran, or idles until more clients arrive. In effect, each client will simply be assigned its own processor. Whenever a processor needs to perform a scheduling decision, it thus executes the following routine:

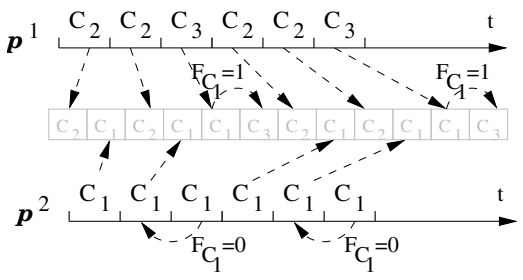

Figure 3: GR^3 multiprocessor scheduling. The two processors schedule either from the central queue, or use the frontlog mechanism when the task is already running.

```
MP-SCHEDULE(℘^k)
 1  C ← C(℘^k)                          ▷ Client just run
 2  if C = NIL
 3     then if N < P
 4             then return NIL           ▷ Idle
 5     else  if F_C > 0
 6             then F_C ← F_C − 1
 7                  return C
 8  C ← INTERGROUP-SCHEDULE()
 9  while ∃℘ s.t. C = C(℘)
10        do F_C ← F_C + 1
11           C ← INTERGROUP-SCHEDULE()
12  return C
```

To illustrate GR^3MP scheduling, Figure 3 shows an example on a dual-processor system with three clients C_1, C_2, and C_3 of weights 3, 2, and 1, respectively. C_1 and C_2 will then be part of the order 1 group (assume C_2 is before C_1 in the round-robin queue of this group), whereas C_3 is part of the order 0 group. The GR^3 schedule is C_2, C_1, C_2, C_1, C_1, C_3. $\wp^1$ will then select C_2 to run, and $\wp^2$ selects C_1. When $\wp^1$ finishes, according to GR^3, it will select C_2 once more, whereas $\wp^2$ selects C_1 again. When $\wp^1$ again selects the next GR^3 client, which is C_1, it finds that it is already running on $\wp^2$ and thus we set $F_{C_1} = 1$ and select the next client, which is C_3, to run on $\wp^1$. When $\wp^2$ finishes running C_1 for its second time quantum, it finds $F_{C_1} = 1$, sets $F_{C_1} = 0$ and continues running C_1 without any scheduling decision on the GR^3 queue.

3.2 GR^3MP Dynamic Considerations

GR^3MP basically does the same thing as the GR^3 algorithm under dynamic considerations. However, the frontlogs used in GR^3MP need to be accounted for appropriately. If some processors have long frontlogs for their currently running clients, newly arriving clients may not be run by those processors until their frontlogs are processed, resulting in bad responsiveness for the new clients. Although in between any two client arrivals or departures, some processors must have no frontlog, the set of such

processors can be as small as a single processor. In this case, newly arrived clients will end up competing with other clients already in the run queue only for those few processors, until the frontlog on the other processors is exhausted.

GR^3MP provides fair and responsive allocations by creating frontlogs for newly arriving clients. Each new client is assigned a frontlog equal to a fraction of the total current frontlog in the system based on its proportional share. Each processor now maintains a queue of frontlog clients and a new client with a frontlog is immediately assigned to one of the processor frontlog queues. Rather than running its currently running client until it completes its frontlog, each processor now round robins among clients in its frontlog queue. Given that frontlogs are small in practice, round-robin scheduling is used for frontlog clients for its simplicity and fairness. GR^3MP balances the frontlog load on the processors by placing new frontlog clients on the processor with the smallest frontlog summed across all its frontlog clients.

More precisely, whenever a client C arrives, and it belongs in group $G(C)$, GR^3MP performs the same group operations as in the single processor GR^3 algorithm. GR^3MP finds the processor $\wp^k$ with the smallest frontlog, then creates a frontlog for client C on $\wp^k$ of length $F_C = F_T \frac{\phi_C}{\Phi_T}$, where F_T is the total frontlog on all the processors. Let $C' = C(\wp^k)$. Then, assuming no further clients arrive, $\wp^k$ will round-robin between C and C' and run C for F_C and C' for $F_{C'}$ time quanta.

When a client becomes not runnable, GR^3MP uses the same lazy removal mechanism used in GR^3. If it is removed from the run queue and has a frontlog, GR^3MP simply discards it since each client is assigned a frontlog based on the current state of the system when it becomes runnable again.

3.3 GR^3MP Weight Readjustment

Since no client can run on more than one processor at a time, no client can consume more than a $1/P$ fraction of the processing in a multiprocessor system. A client C with weight ϕ_C greater than Φ_T/P is considered *infeasible* since it cannot receive its proportional share allocation ϕ_C/Φ_T without using more than one processor simultaneously. GR^3MP should then give the client its maximum possible service, and simply assign such a client its own processor to run on. However, since the scheduler uses client weights to determine which client to run, an infeasible client's weight must be adjusted so that it is feasible to ensure that the scheduling algorithm runs correctly to preserve fairness (assuming there are at least P clients). GR^3MP potentially needs to perform weight readjustment whenever a client is inserted or removed from the run queue to make sure that all weights are feasible.

To understand the problem of weight readjustment, consider the sequence of all clients, ordered by weight: $S_{1,N} = C_1, C_2, \ldots, C_N$ with $\phi_1 \geq \phi_2 \geq \ldots \geq \phi_N$. We call the subsequence $S_{k,N} = C_k, C_{k+1}, \ldots, .C_N$ *Q-feasible*, if $\phi_k \leq \frac{1}{Q}\sum_{j=k}^{N}\phi_j$.

Lemma 1. *The client mix in the system is* feasible *if and only if $S_{1,N}$ is P-feasible.*

Proof. If $\phi_1 > \frac{\Phi_T}{P}$, C_1 is infeasible, so the mix is infeasible. Conversely, if $\phi_1 \leq \frac{\Phi_T}{P}$, then for any client C_l, $\phi_l \leq \phi_1 \leq \frac{\Phi_T}{P}$, implying all clients are feasible. The mix is then feasible $\iff \phi_1 \leq \frac{\Phi_T}{P} = \frac{1}{P}\sum_{j=1}^{N}\phi_j$, or, equivalently, $S_{1,N}$ is P-feasible. □

Lemma 2. *$S_{k,N}$ is Q-feasible $\Rightarrow$ $S_{k+1,N}$ is $(Q-1)$-feasible.*

Proof. $\phi_k \leq \frac{1}{Q}\sum_{j=k}^{N}\phi_j \iff Q\phi_k \leq \phi_k + \sum_{j=k+1}^{N}\phi_j \iff \phi_k \leq \frac{1}{Q-1}\sum_{j=k+1}^{N}\phi_j$. Since $\phi_{k+1} \leq \phi_k$, the lemma follows. □

The feasibility problem is then to identify the least k (denoted the *feasibility threshold*, f) such that $S_{k,N}$ is $(P-k+1)$-feasible. If $f = 1$, then the client mix is feasible. Otherwise, the *infeasible set* $S_{1,f-1} = C_1, \ldots, C_{f-1}$ contains the infeasible clients, whose weight needs to be scaled down to $1/P$ of the resulting total weight. The cardinality $f-1$ of the infeasible set is less than P. However, the sorted sequence $S_{1,N}$ is expensive to maintain, such that traversing it and identifying the feasibility threshold is not an efficient solution.

GR^3MP leverages its grouping strategy to perform fast weight readjustment. GR^3MP starts with the unmodified client weights, finds the set I of infeasible clients, and adjust their weights to be feasible. To construct I, the algorithm traverses the list of groups in decreasing order of their group order σ_G, until it finds a group not all of whose clients are infeasible. We denote by $|I|$ the cardinality of I and by Φ_I the sum of weights of the clients in I, $\sum_{C\in I}\phi_C$. The GR^3MP weight readjustment algorithm is as follows:

```
WEIGHT-READJUSTMENT()
 1  RESTORE-ORIGINAL-WEIGHTS
 2  I ← ∅
 3  G ← greatest order group
 4  while |G| < P − |I|  and  2^σG > (Φ_T − Φ_I − Φ_G)/(P − |I| − |G|)
 5        do I ← I ∪ G
 6           G ← next(G)            ▷ by group order
 7  if |G| < 2(P − |I|)
 8     then I ← I ∪ INFEASIBLE(G, P − |I|, Φ_T − Φ_I)
 9  Φ_T^f ← Φ_T − Φ_I
10  Φ_T ← P/(P − |I|) Φ_T^f
11  for each C ∈ I
12        do φ_C ← Φ_T/P
```

The correctness of the algorithm is based on Lemma 2. Let some group G span the subsequence $S_{i,j}$ of the sequence of ordered clients $S_{1,N}$. Then $2^{\sigma_G+1} - 1 \geq \phi_i \geq \ldots \geq \phi_j \geq 2^{\sigma_G}$ and it is easy to show:

- $2^{\sigma_G} > \frac{\Phi_T - \Phi_I - \Phi_G}{P-|I|-|G|} \Rightarrow j < f$ (all clients in $S_{1,j}$ are infeasible).
- $2^{\sigma_G} \leq \frac{\Phi_T - \Phi_I - \Phi_G}{P-|I|-|G|} \Rightarrow j+1 \geq f$ (all clients in $S_{j+1,N}$ are feasible).

Once we reach line 7, we know $S_{j+1,N}$ is $(P-j)$-feasible, and $i \leq f \leq j+1$. If $|G| \geq 2(P-|I|)$, GR^3MP can stop searching for infeasible clients since all clients $C \in G$ are feasible, and $f = i$ (equivalently, $S_{i,N}$ is $(P-|I|)$-feasible): $\phi_C < 2^{\sigma_G+1} \leq 2\frac{1}{|G|}\Phi_G \leq \frac{1}{P-|I|}\Phi_G \leq \frac{1}{P-|I|}(\Phi_T - \Phi_I)$. Otherwise, if $|G| < 2(P-|I|)$, then $i < f \leq j+1$ and GR^3MP needs to search through G to determine which clients are infeasible (equivalently, find f). Since the number of clients in G is small, we can sort all clients in G by weight. Then, starting from the largest weight client in G, find the first feasible client. A simple algorithm is then the following:

INFEASIBLE(G, Q, Φ)

```
1  I ← ∅
2  for each C ∈ G in sorted order
3       do if φ_C > 1/(Q−|I|)(Φ − Φ_I)
4            then I ← I ∪ {C}
5            else return I
6  return I
```

GR^3MP can alternatively use a more complicated but lower time complexity divide-and-conquer algorithm to find the infeasible clients in G. In this case, GR^3MP partitions G around its median $\bar{C}$ into G_S, the set of G clients that have weight less than $\phi_{\bar{C}}$ and G_B, the set of G clients that have weight larger than $\phi_{\bar{C}}$. By Lemma 2, if $\bar{C}$ is feasible, $G_S \cup \{\bar{C}\}$ is feasible, and we recurse on G_B. Otherwise, all clients in $G_B \cup \{\bar{C}\}$ are infeasible, and we recurse on G_S to find all infeasible clients. The algorithm finishes when the set we need to recurse on is empty:

INFEASIBLE(G, Q, Φ)

```
1  if G = ∅
2     then return ∅
3  C̄ ← MEDIAN(G)
4  (G_S, G_B) ← PARTITION(G, φ_C̄)
5  if φ_C̄ > (Φ−Φ_{G_B})/(Q−|G_B|)
6     then return G_B ∪ {C̄} ∪
        INFEASIBLE(G_S, Q − |G_B| − 1, Φ − Φ_{G_B} − φ_C̄)
7     else return INFEASIBLE(G_B, Q, Φ)
```

Once all infeasible clients have been identified, WEIGHT-READJUSTMENT() determines the sum of the weights of all feasible clients, $\Phi_T^f = \Phi_T - \Phi_I$. We can now compute the new total weight in the system as $\Phi_T = \frac{P}{P-|I|}\Phi_T^f$, namely the solution to the equation $\Phi_T^f + |I|\frac{x}{P} = x$. Once we have the adjusted Φ_T, we change all the weights for the infeasible clients in I to $\frac{\Phi_T}{P}$. Lemma 6 in Section 4.2 shows the readjustment algorithm runs in time $O(P)$ and is thus asymptotically optimal, since there can be $\Theta(P)$ infeasible clients.

4 GR^3 Fairness and Complexity

We analyze the fairness and complexity of GR^3 and GR^3MP. To analyze fairness, we use a more formal notion of proportional fairness defined as *service error*, a measure widely used [1, 7, 9, 17, 18, 19, 25, 27] in the analysis of scheduling algorithms. To simplify the analysis, we will assume that clients are always runnable and derive fairness bounds for such a case. Subsequently, we address the impact of arrivals and departures.

We use a strict measure of service error (equivalent in this context to the *Normalized Worst-case Fair Index* [1]) relative to Generalized Processor Sharing (GPS) [16], an idealized model that achieves *perfect fairness*: $w_C = W_T\frac{\phi_C}{\Phi_T}$, an ideal state in which each client C always receives service exactly proportional to its weight. Although all real-world schedulers must time-multiplex resources in time units of finite size and thus cannot maintain perfect fairness, some algorithms stay closer to perfect fairness than others and therefore have less service error. We quantify how close an algorithm gets to perfect fairness using the *client service time error*, which is the difference between the service received by client C and its share of the total work done by the processor: $e_C = w_C - W_T\frac{\phi_C}{\Phi_T}$. A positive service time error indicates that a client has received more than its ideal share over a time interval; a negative error indicates that it has received less. To be precise, the error e_C measures how much time a client C has received beyond its ideal allocation. A proportional share scheduler should minimize the absolute value of the allocation error of all clients with minimal scheduling overhead.

We provide bounds on the service error of GR^3 and GR^3MP. To do this, we define two other measures of service error. The *group service time error* is a similar measure for groups that quantifies the fairness of allocating the processor among groups: $E_G = W_G - W_T\frac{\Phi_G}{\Phi_T}$. The *group-relative service time error* represents the service time error of client C if there were only a single group $G = G(C)$ in the scheduler and is a measure of the service error of a client with respect to the work done on behalf of its group: $e_{C,G} = w_C - W_G\frac{\phi_C}{\Phi_G}$. We first show bounds on the group service error of the intergroup scheduling algorithm. We then show bounds on the group-relative service error of the intragroup scheduling algorithm. We combine these results to obtain the overall client service error bounds. We also discuss the scheduling overhead of GR^3 and GR^3MP in

terms of their time complexity. We show that both algorithms can make scheduling decisions in $O(1)$ time with $O(1)$ service error given a constant number of groups. Due to space constraints, most of the proofs are omitted. Further proof details are available in [5].

4.1 Analysis of GR^3

Intergroup Fairness For the case when the weight ratios of consecutive groups in the group list are integers, we get the following:

Lemma 3. *If* $\frac{\Phi_j}{\Phi_{j+1}} \in \mathbb{N},\ 1 \leq j < g$, *then* $-1 < E_{G_k} \leq (g-k)\frac{\Phi_k}{\Phi_T}$ *for any group* G_k.

Proof sketch: If the group currently scheduled is G_k, then the work to weight ratio of all groups G_j, $j < k$, is the same. For $j > k$, $\frac{W_{j+1}}{\Phi_{j+1}} \leq \frac{W_j}{\Phi_j} \leq \frac{W_{j+1}+1}{\Phi_{j+1}} - \frac{1}{\Phi_j}$ as a consequence of the well-ordering condition (2). After some rearrangements, we can sum over all j and bound W_k, and thus E_{G_k} above and below. The additive $\frac{1}{\Phi_j}$ will cause the $g-1$ upper bound.

In the general case, we get similar, but slightly weaker bounds.

Lemma 4. *For any group* G_k, $-\frac{(g-k)(g-k-1)}{2}\frac{\Phi_k}{\Phi_T} - 1 < E_{G_k} < g-1$.

The proof for this case (omitted) follows reasoning similar to that of the previous lemma, but with several additional complications.

It is clear that the lower bound is minimized when setting $k = 1$. Thus, we have

Corollary 1. $-\frac{(g-1)(g-2)}{2}\frac{\Phi_G}{\Phi_T} - 1 < E_G < g-1$ *for any group* G.

Intragroup Fairness Within a group, all weights are within a factor of two and the group-relative error is bound by a small constant. The only slightly subtle point is to deal with fractional rounds.

Lemma 5. $-3 < e_{C,G} < 4$ *for any client* $C \in G$.

Overall Fairness of GR^3 Based on the identity $e_C = e_{C,G} + \frac{\phi_C}{\Phi_G}E_G$ which holds for any group G and any client $C \in G$, we can combine the inter- and intragroup analyses to bound the overall fairness of GR^3.

Theorem 1. $-\frac{(g-1)(g-2)}{2}\frac{\phi_C}{\Phi_T} - 4 < e_C < g+3$ *for any client* C.

The negative error of GR^3 is thus bounded by $O(g^2)$ and the positive error by $O(g)$. Recall, g, the number of groups, does not depend on the number of clients in the system.

Dynamic Fairness of GR^3 We can consider a client arrival or removal as an operation where a group is first removed from the group list and added in a different place with a different weight. We argue that fairness is preserved by these operations: when group G_k is removed, then G_{k-1}, G_k, and G_{k+1} were well-ordered as defined in (2), so after the removal, G_{k-1} and G_{k+1}, now neighbors, will be well-ordered by transitivity. When a group, call it $G_{i+(1/2)}$, is inserted between G_i and G_{i+1}, it can be proven that the work readjustment formula in Section 2.3 ensures $G_{i+(1/2)}$ and G_{i+1} are well-ordered. In the case of G_i and $G_{i+(1/2)}$, we can show that we can achieve well-ordering by running $G_{i+(1/2)}$ at most one extra time. Thus, modulo this readjustment, the intragroup algorithm's fairness bounds are preserved. An important property of our algorithm that follows is that the pairwise ratios of work of clients *not* part of the readjusted group will be unaffected. Since the intragroup algorithm has constant fairness bounds, the disruption for the work received by clients inside the adjusted group is only $O(1)$.

Time Complexity GR^3 manages to bound its service error by $O(g^2)$ while maintaining a strict $O(1)$ scheduling overhead. The intergroup scheduler either selects the next group in the list, or reverts to the first one, which takes constant time. The intragroup scheduler is even simpler, as it just picks the next client to run from the unordered round robin list of the group. Adding and removing a client is worst-case $O(g)$ when a group needs to be relocated in the ordered list of groups. This could of course be done in $O(\log g)$ time (using binary search, for example), but the small value of g in practice does not justify a more complicated algorithm.

The *space complexity* of GR^3 is $O(g) + O(N) = O(N)$. The only additional data structure beyond the unordered lists of clients is an ordered list of length g to organize the groups.

4.2 Analysis of GR^3MP

Overall Fairness of GR^3MP Given feasible client weights after weight readjustment, the service error for GR^3MP is bounded below by the GR^3 error, and above by a bound which improves with more processors.

Theorem 2. $-\frac{(g-1)(g-2)}{2}\frac{\phi_C}{\Phi_T} - 4 < e_C < 2g + 10 + \frac{(g-1)(g-2)}{2P}$ *for any client* C.

Time Complexity of GR^3MP The frontlogs create an additional complication when analyzing the time complexity of GR^3MP. When an idle processor looks for its next client, it runs the simple $O(1)$ GR^3 algorithm to find a client C. If C is not running on any other processor, we are done, but otherwise we place it on the frontlog and then we must rerun the GR^3 algorithm until we find a client

that is not running on any other processor. Since for each such client, we increase its allocation on the processor it runs, the amortized time complexity remains $O(1)$. The upper bound on the time that any single scheduling decision takes is given by the maximum length of any scheduling sequence of GR^3 consisting of only some fixed subset of $P-1$ clients.

Theorem 3. *The time complexity per scheduling decision in GR^3MP is bounded above by $\frac{(g-k)(g-k+1)}{2} + (k+1)(g-k+1)P + 1$ where $1 \leq k \leq g$.*

Thus, the length of any schedule consisting of at most $P-1$ clients is $O(g^2P)$. Even when a processor has front-logs for several clients queued up on it, it will schedule in $O(1)$ time, since it performs round-robin among the front-logged clients. Client arrivals and departures take $O(g)$ time because of the need to readjust group weights in the saved list of groups. Moreover, if we also need to use the weight readjustment algorithm, we incur an additional $O(P)$ overhead on client arrivals and departures.

Lemma 6. *The complexity of the weight readjustment algorithm is $O(P)$.*

Proof. Restoring the original weights will worst case touch a number of groups equal to the number of previously infeasible clients, which is $O(P)$. Identifying the infeasible clients involves iterating over at most P groups in decreasing sequence based on group order, as described in Section 3.3. For the last group considered, we only attempt to partition it into feasible and infeasible clients of its size is less than $2P$. Since partitioning of a set can be done in linear time, and we recurse on a subset half the size, this operation is $O(P)$ as well. □

For small P, the $O(P\log(P))$ sorting approach to determine infeasible clients in the last group considered is simpler and in practice performs better than the $O(P)$ recursive partitioning. Finally, altering the active group structure to reflect the new weights is a $O(P+g)$ operation, as two groups may need to be re-inserted in the ordered list of groups.

5 Measurements and Results

We have implemented GR^3 uniprocessor and multiprocessor schedulers in the Linux operating system and measured their performance. We present some experimental data quantitatively comparing GR^3 performance against other popular scheduling approaches from both industrial practice and research. We have conducted both extensive simulation studies and detailed measurements of real kernel scheduler performance on real applications.

Section 5.1 presents simulation results comparing the proportional sharing accuracy of GR^3 and GR^3MP against WRR, WFQ [18], SFQ [13], VTRR [17], and SRR [9]. The simulator enabled us to isolate the impact of the scheduling algorithms themselves and examine the scheduling behavior of these different algorithms across hundreds of thousands of different combinations of clients with different weight values.

Section 5.2 presents detailed measurements of real kernel scheduler performance by comparing our prototype GR^3 Linux implementation against the standard Linux scheduler, a WFQ scheduler, and a VTRR scheduler. The experiments we have done quantify the scheduling overhead and proportional share allocation accuracy of these schedulers in a real operating system environment under a number of different workloads.

All our kernel scheduler measurements were performed on an IBM Netfinity 4500 system with one or two 933 MHz Intel Pentium III CPUs, 512 MB RAM, and 9 GB hard drive. The system was installed with the Debian GNU/Linux distribution version 3.0 and all schedulers were implemented using Linux kernel version 2.4.19. The measurements were done by using a minimally intrusive tracing facility that writes timestamped event identifiers into a memory log and takes advantage of the high-resolution clock cycle counter available with the Intel CPU, providing measurement resolution at the granularity of a few nanoseconds. Getting a timestamp simply involved reading the hardware cycle counter register. We measured the timestamp overhead to be roughly 35 ns per event.

The kernel scheduler measurements were performed on a fully functional system. All experiments were performed with all system functions running and the system connected to the network. At the same time, an effort was made to eliminate variations in the test environment to make the experiments repeatable.

5.1 Simulation Studies

We built a scheduling simulator that measures the service time error, described in Section 4, of a scheduler on a set of clients. The simulator takes four inputs, the scheduling algorithm, the number of clients N, the total sum of weights Φ_T, and the number of client-weight combinations. The simulator randomly assigns weights to clients and scales the weights to ensure that they add up to Φ_T. It then schedules the clients using the specified algorithm as a real scheduler would, assuming no client blocks, and tracks the resulting service time error. The simulator runs the scheduler until the resulting schedule repeats, then computes the maximum (most positive) and minimum (most negative) service time error across the nonrepeating portion of the schedule for the given set of clients and weight assignments. This process is repeated for the specified number of client-weight combinations. We then compute the maximum service time error and minimum service time error for the specified number of client-weight combinations to obtain a "worst-case" error range.

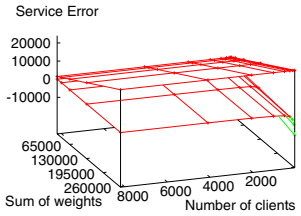

Figure 4: WRR error

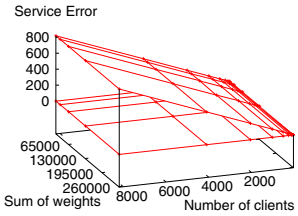

Figure 5: WFQ error

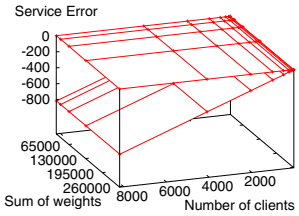

Figure 6: SFQ error

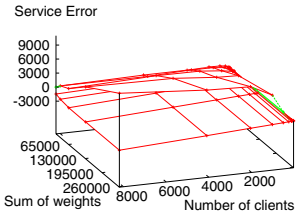

Figure 7: VTRR error

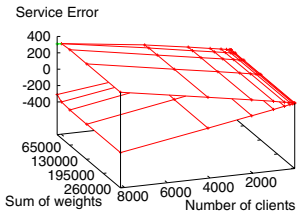

Figure 8: SRR error

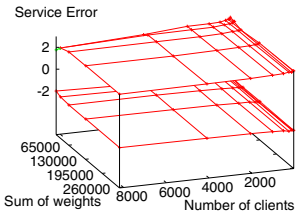

Figure 9: GR^3 error

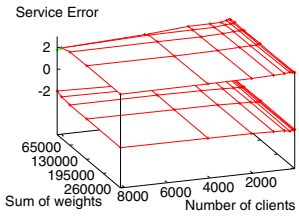

Figure 10: GR^3MP error

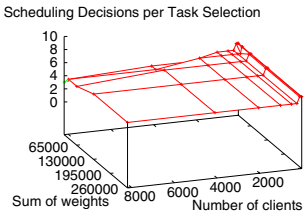

Figure 11: GR^3MP overhead

To measure proportional fairness accuracy, we ran simulations for each scheduling algorithm on 45 different combinations of N and Φ_T (32 up to 8192 clients and 16384 up to 262144 total weight, respectively). Since the proportional sharing accuracy of a scheduler is often most clearly illustrated with skewed weight distributions, one of the clients was given a weight equal to 10 percent of Φ_T. All of the other clients were then randomly assigned weights to sum to the remaining 90 percent of Φ_T. For each pair (N, Φ_T), we ran 2500 client-weight combinations and determined the resulting worst-case error range.

The worst-case service time error ranges for WRR, WFQ, SFQ, VTRR, SRR, and GR^3 with these skewed weight distributions are in Figures 4 to 9. Due to space constraints, WF^{2}Q error is not shown since the results simply verify its known mathematical error bounds of -1 and 1 tu. Each figure consists of a graph of the error range for the respective scheduling algorithm. Each graph shows two surfaces representing the maximum and minimum service time error as a function of N and Φ_T for the same range of values of N and Φ_T. Figure 4 shows WRR's service time error is between -12067 tu and 23593 tu. Figure 5 shows WFQ's service time error is between -1 tu and 819 tu, which is much less than WRR. Figure 6 shows SFQ's service time error is between -819 tu and 1 tu, which is almost a mirror image of WFQ. Figure 7 shows VTRR's service error is between -2129 tu and 10079 tu. Figure 8 shows SRR's service error is between -369 tu and 369 tu.

In comparison, Figure 9 shows the service time error for GR^3 only ranges from -2.5 to 3.0 tu. GR^3 has a smaller error range than all of the other schedulers measured except WF^{2}Q. GR^3 has both a smaller negative and smaller positive service time error than WRR, VTRR, and SRR. While GR^3 has a much smaller positive service error than WFQ, WFQ does have a smaller negative service time error since it is bounded below at -1. Similarly, GR^3 has a much smaller negative service error than SFQ, though SFQ's positive error is less since it is bounded above at 1. Considering the total service error range of each scheduler, GR^3 provides well over two orders of magnitude better proportional sharing accuracy than WRR, WFQ, SFQ, VTRR, and SRR. Unlike the other schedulers, these results show that GR^3 combines the benefits of low service time errors with its ability to schedule in $O(1)$ time.

Note that as the weight skew becomes more accentuated, the service error can grow dramatically. Thus, increasing the skew from 10 to 50 percent results in more than a fivefold increase in the error magnitude for SRR, WFQ, and SFQ, and also significantly worse errors for WRR and VTRR. In contrast, the error of GR^3 is still bounded by small constants: -2.3 and 4.6.

We also measured the service error of GR^3MP using this simulator configured for an 8 processor system, where the weight distribution was the same as for the uniprocessor simulations above. Note that the client given 0.1 of the total weight was feasible, since $0.1 < \frac{1}{8} = 0.125$. Figure 10 shows GR^3MP's service error is between -2.5 tu and 2.8 tu, slightly better than for the uniprocessor case, a benefit of being able to run multiple clients in parallel. Figure 11 shows the maximum number of scheduling decisions that an idle processor needs to perform until it finds a client that is not running. This did not exceed seven, indicating that the number of decisions needed in practice is well below the worst-case bounds shown in Theorem 3.

5.2 Linux Kernel Measurements

To evaluate the scheduling overhead of GR^3, we compare it against the standard Linux 2.4 scheduler, a WFQ scheduler, and a VTRR scheduler. Since WF^{2}Q has the-

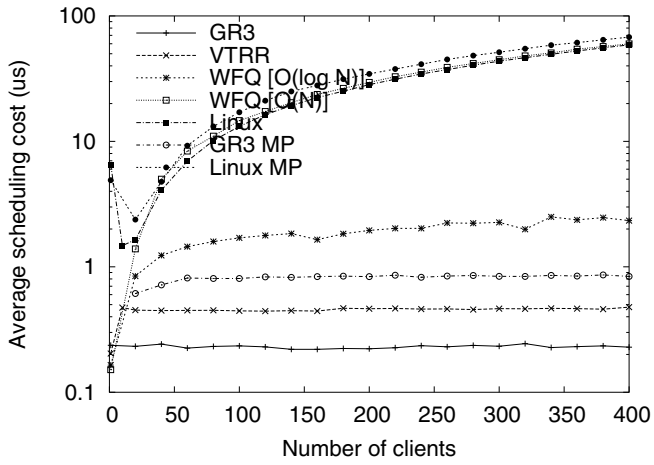

Figure 12: Average scheduling overhead

oretically the same time complexity as WFQ (but with larger constants, because of the complexity of its steps), we present WFQ as a lower bound for the overhead of WF^{2}Q. We present results from several experiments that quantify how scheduling overhead varies as the number of clients increases. For the first experiment, we measure scheduling overhead for running a set of clients, each of which executed a simple micro-benchmark which performed a few operations in a while loop. A control program was used to fork a specified number of clients. Once all clients were runnable, we measured the execution time of each scheduling operation that occurred during a fixed time duration of 30 seconds. The measurements required two timestamps for each scheduling decision, so measurement error of 70 ns are possible due to measurement overhead. We performed these experiments on the standard Linux scheduler, WFQ, VTRR, and GR^3 for 1 to 400 clients.

Figure 12 shows the average execution time required by each scheduler to select a client to execute. Results for GR^3, VTRR, WFQ, and Linux were obtained on uniprocessor system, and results for GR^3MP and LinuxMP were obtained running on a dual-processor system. Dual-processor results for WFQ and VTRR are not shown since MP-ready implementations of them were not available.

For this experiment, the particular implementation details of the WFQ scheduler affect the overhead, so we include results from two different implementations of WFQ. In the first, labeled "WFQ [$O(N)$]", the run queue is implemented as a simple linked list which must be searched on every scheduling decision. The second, labeled "WFQ [$O(\log N)$]", uses a heap-based priority queue with $O(\log N)$ insertion time. To maintain the heap-based priority queue, we used a fixed-length array. If the number of clients ever exceeds the length of the array, a costly array reallocation must be performed. Our initial array size was large enough to contain more than 400 clients, so this additional cost is not reflected in our measurements.

As shown in Figure 12, the increase in scheduling overhead as the number of clients increases varies a great deal between different schedulers. GR^3 has the smallest scheduling overhead. It requires roughly 300 ns to select a client to execute and the scheduling overhead is essentially constant for all numbers of clients. While VTRR scheduling overhead is also constant, GR^3 has less overhead because its computations are simpler to perform than the virtual time calculations required by VTRR. In contrast, the overhead for Linux and for $O(N)$ WFQ scheduling grows linearly with the number of clients. Both of these schedulers impose more than 200 times more overhead than GR^3 when scheduling a mix of 400 clients. $O(\log N)$ WFQ has much smaller overhead than Linux or $O(N)$ WFQ, but it still imposes significantly more overhead than GR^3, with 8 times more overhead than GR^3 when scheduling a mix of 400 clients. Figure 12 also shows that GR^3MP provides the same $O(1)$ scheduling overhead on a multiprocessor, although the absolute time to schedule is somewhat higher due to additional costs associated with scheduling in multiprocessor systems. The results show that GR^3MP provides substantially lower overhead than the standard Linux scheduler, which suffers from complexity that grows linearly with the number of clients. Because of the importance of constant scheduling overhead in server systems, Linux has switched to Ingo Molnar's $O(1)$ scheduler in the Linux 2.6 kernel. As a comparison, we also repeated this microbenchmark experiment with that scheduler and found that GR^3 still runs over 30 percent faster.

As another experiment, we measured the scheduling overhead of the various schedulers for $hackbench$ [21], a Linux benchmark used for measuring scheduler performance with large numbers of processes entering and leaving the run queue at all times. It creates groups of readers and writers, each group having 20 reader tasks and 20 writer tasks, and each writer writes 100 small messages to each of the other 20 readers. This is a total of 2000 messages sent per writer, per group, or 40000 messages per group. We ran a modified version of hackbench to give each reader and each writer a random weight between 1 and 40. We performed these tests on the same set of schedulers for 1 group up to 100 groups. Using 100 groups results in up to 8000 processes running. Because hackbench frequently inserts and removes clients from the run queue, the cost of client insertion and removal is a more significant factor for this benchmark. The results show that the simple dynamic group adjustments described in Section 2.3 have low overhead, since $O(g)$ can be considered constant in practice.

Figure 13 shows the average scheduling overhead for each scheduler. The average overhead is the sum of the times spent on all scheduling events, selecting clients to run and inserting and removing clients from the run queue, divided by the number of times the scheduler selected a client to run. The overhead in Figure 13 is higher than the av-

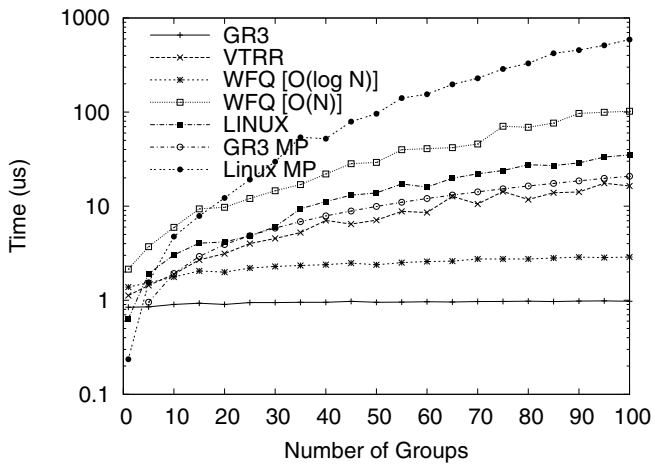

Figure 13: Hackbench weighted scheduling overhead

erage cost per schedule in Figure 12 for all the schedulers measured since Figure 13 includes a significant component of time due to client insertion and removal from the run queue. GR^3 still has by far the smallest scheduling overhead among all the schedulers measured. The overhead for GR^3 remains constant while the overhead for $O(\log N)$ WFQ, $O(N)$ WFQ, VTRR, and Linux grows with the number of clients. Client insertion, removal, and selection to run in GR^3 are independent of the number of clients. The cost for GR^3 is 3 times higher than before, with client selection to run, insertion, and removal each taking approximately 300 to 400 ns. For VTRR, although selecting a client to run is also independent of the number of clients, insertion overhead grows with the number of clients, resulting in much higher VTRR overhead for this benchmark.

To demonstrate GR^3's efficient proportional sharing of resources on real applications, we briefly describe three simple experiments running web server workloads using the same set of schedulers: GR^3 and GR^3MP, Linux 2.4 uniprocessor and multiprocessor schedulers, WFQ, and VTRR. The web server workload emulates a number of virtual web servers running on a single system. Each virtual server runs the guitar music search engine used at guitarnotes.com, a popular musician resource web site with over 800,000 monthly users. The search engine is a perl script executed from an Apache mod-perl module that searches for guitar music by title or author and returns a list of results. The web server workload configured each server to pre-fork 100 processes, each running consecutive searches simultaneously.

We ran multiple virtual servers with each one having different weights for its processes. In the first experiment, we used six virtual servers, with one server having all its processes assigned weight 10 while all other servers had processes assigned weight 1. In the second experiment, we used five virtual servers and processes assigned to each server had respective weights of 1, 2, 3, 4, and 5. In the third experiment, we ran five virtual servers which assigned a random weight between 1 and 10 to each process. For the Linux scheduler, weights were assigned by selecting `nice` values appropriately. Figures 14 to 19 present the results from the first experiment with one server with weight 10 processes and all other servers with weight 1 processes. The total load on the system for this experiment consisted of 600 processes running simultaneously. For illustration purposes, only one process from each server is shown in the figures. Conclusions drawn from the other experiments are the same; those results are omitted due to space constraints.

GR^3 and GR^3MP provided the best overall proportional fairness for these experiments while Linux provided the worst overall proportional fairness. Figures 14 to 17 show the amount of processor time allocated to each client over time for the Linux scheduler, WFQ, VTRR, and GR^3. All of the schedulers except GR^3 and GR^3MP have a pronounced "staircase" effect for the search engine process with weight 10, indicating that CPU resources are provided in irregular bursts over a short time interval. For the applications which need to provide interactive responsiveness to web users, this can result in extra delays in system response time. It can be inferred from the smoother curves of Figure 17 that GR^3 and GR^3MP provide fair resource allocation at a finer granularity than the other schedulers.

6 Related Work

Round robin is one of the oldest, simplest and most widely used proportional share scheduling algorithms. Weighted round-robin (WRR) supports non-uniform client weights by running all clients with the same frequency but adjusting the size of their time quanta in proportion to their respective weights. Deficit round-robin (DRR) [22] was developed to support non-uniform service allocations in packet scheduling. These algorithms have low $O(1)$ complexity but poor short-term fairness, with service errors that can be on the order of the largest client weight in the system. GR^3 uses a novel variant of DRR for intragroup scheduling with $O(1)$ complexity, but also provides $O(1)$ service error by using its grouping mechanism to limit the effective range of client weights considered by the intragroup scheduler.

Fair-share schedulers [12, 14, 15] provide proportional sharing among users in a way compatible with a UNIX-style time-sharing framework based on multi-level feedback with a set of priority queues. These schedulers typically had low $O(1)$ complexity, but were often ad-hoc and could not provide any proportional fairness guarantees. Empirical measurements show that these approaches only provide reasonable proportional fairness over relatively large time intervals [12].

Lottery scheduling [26] gives each client a number of tickets proportional to its weight, then randomly selects a ticket. Lottery scheduling takes $O(\log N)$ time and relies on the law of large numbers for providing proportional fair-

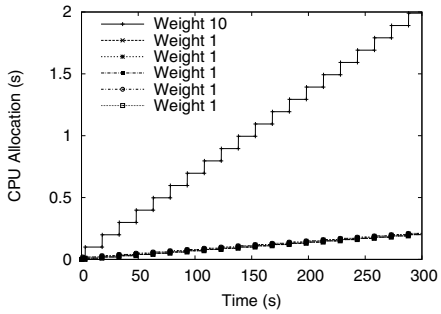

Figure 14: Linux uniprocessor

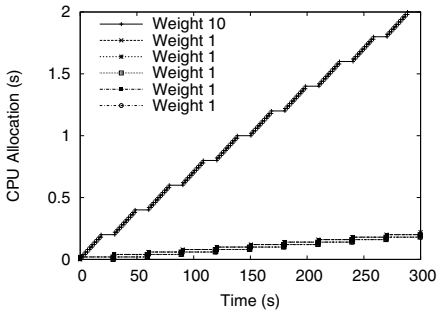

Figure 15: WFQ uniprocessor

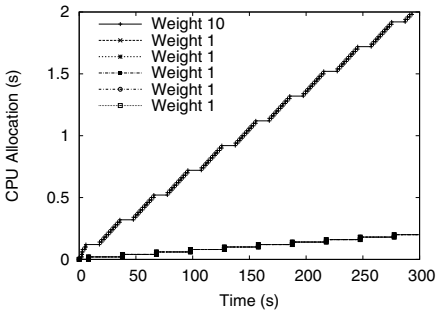

Figure 16: VTRR uniprocessor

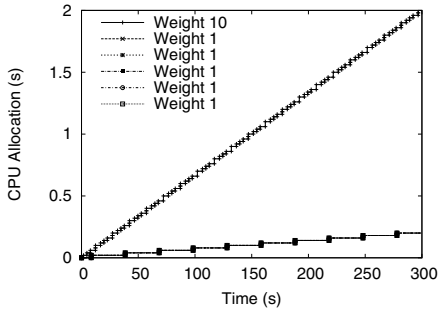

Figure 17: GR^3 uniprocessor

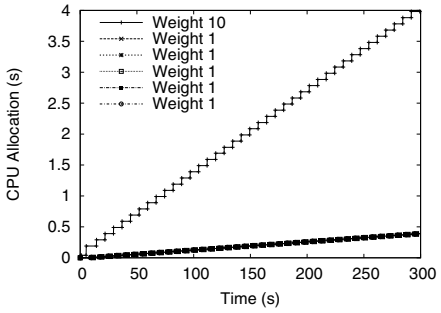

Figure 18: Linux multiprocessor

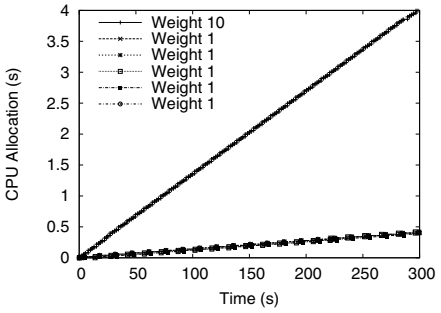

Figure 19: GR^3MP multiprocessor

ness. Thus, its allocation errors can be very large, typically much worse than WRR for clients with smaller weights.

Weighted Fair Queueing (WFQ) [11, 18], was first developed for network packet scheduling, and later applied to uniprocessor scheduling [26]. It assigns each client a virtual time and schedules the client with the earliest virtual time. Other fair queueing variants such as Virtual-clock [28], SFQ [13], SPFQ [24], and Time-shift FQ [10] have also been proposed. These approaches all have $O(\log N)$ time complexity, where N is the number of clients, because the clients must be ordered by virtual time. It has been shown that WFQ guarantees that the service time error for any client never falls below -1 [18]. However, WFQ can allow a client to get far ahead of its ideal allocation and accumulate a large positive service time error of $O(N)$, especially with skewed weight distributions.

Several fair queueing approaches have been proposed for reducing this $O(N)$ service time error. A hierarchical scheduling approach [26] reduces service time error to $O(\log N)$. Worst-Case Weighted Fair Queueing (WF^{2}Q) [1] introduced eligible virtual times and can guarantee both a lower and upper bound on error of -1 and $+1$, respectively for network packet scheduling. It has also been applied to uniprocessor scheduling as Eligible Virtual Deadline First (EEVDF) [25]. These algorithms provide stronger proportional fairness guarantees than other approaches, but are more difficult to implement and still require at least $O(\log N)$ time.

Motivated by the need for faster schedulers with good fairness guarantees, one of the authors developed Virtual-Time Round-Robin (VTRR) [17]. VTRR first introduced the simple idea of going round-robin through clients but skipping some of them at different frequencies without having to reorder clients on each schedule. This is done by combining round-robin scheduling with a virtual time mechanism. GR^3's intergroup scheduler builds on VTRR but uses weight ratios instead of virtual times to provide better fairness. Smoothed Round Robin (SRR) [9] uses a different mechanism for skipping clients using a Weight Matrix and Weight Spread Sequence (WSS) to run clients by simulating a binary counter. VTRR and SRR provide proportional sharing with $O(1)$ time complexity for selecting a client to run, though inserting and removing clients from the run queue incur higher overhead: $O(\log N)$ for VTRR and $O(k)$ for SRR , where $k = \log \phi_{\max}$ and $\phi_{\max}$ is the maximum client weight allowed. However, unlike GR^3, both algorithms can suffer from large service time errors especially for skewed weight distributions. For example, we can show that the service error of SRR is worst-case $O(kN)$.

Grouping clients to reduce scheduling complexity has been used by [20], [8] and [23]. These fair queueing approaches group clients into buckets based on client virtual timestamps. With the exception of [23], which uses exponential grouping, the fairness of these virtual time bin sorting schemes depends on the granularity of the buckets and is adversely affected by skewed client weight distributions. On the other hand, GR^3 groups based on client weights, which are relatively static, and uses groups as schedulable entities in a two-level scheduling hierarchy.

The grouping strategy used in GR^3 was first introduced by two of the authors for uniprocessor scheduling [6] and generalized by three of the authors to network packet scheduling [4]. A similar grouping strategy was independently developed in Stratified Round Robin (StRR) [19] for

network packet scheduling. StRR distributes all clients with weights between 2^{-k} and $2^{-(k-1)}$ into class F_k (F here not to be confused with our frontlog). StRR splits time into scheduling slots and then makes sure to assign all the clients in class F_k one slot every scheduling interval, using a credit and deficit scheme within a class. This is also similar to GR^3, with the key difference that a client can run for up to two consecutive time units, while in GR^3, a client is allowed to run only once every time its group is selected regardless of its deficit.

StRR has weaker fairness guarantees and higher scheduling complexity than GR^3. StRR assigns each client weight as a fraction of the total processing capacity of the system. This results in weaker fairness guarantees when the sum of these fractions is not close to the limit of 1. For example, if we have $N = 2^k + 1$ clients, one of weight 0.5 and the rest of weight $2^{-(k+2)}$ (total weight = 0.75), StRR will run the clients in such a way that after 2^{k+1} slots, the error of the large client is $\frac{-N}{3}$, such that this client will then run uniterruptedly for N tu to regain its due service. Client weights could be scaled to reduce this error, but with additional $O(N)$ complexity. StRR requires $O(g)$ worst-case time to determine the next class that should be selected, where g is the number of groups. Hardware support can hide this complexity assuming a small, predefined maximum number of groups [19], but running an StRR processor scheduler in software still requires $O(g)$ complexity.

GR^3 also differs from StRR and other deficit round-robin variants in its distribution of deficit. In DRR, SRR, and StRR, the variation in the deficit of all the clients affects the fairness in the system. To illustrate this, consider $N + 1$ clients, all having the same weight except the first one, whose weight is N times larger. If the deficit of all the clients except the first one is close to 1, the error of the first client will be about $\frac{N}{2} = O(N)$. Therefore, the deficit mechanism as employed in round-robin schemes doesn't allow for better than $O(N)$ error. In contrast, GR^3 ensures that a group consumes all the work assigned to it, so that the deficit is a tool used only in distributing work within a certain group, and not within the system. Thus, groups effectively isolate the impact of unfortunate distributions of deficit in the scheduler. This allows for the error bounds in GR^3 to depend only on the number of groups instead of the much larger number of clients.

A rigorous analysis on network packet scheduling [27] suggests that $O(N)$ delay bounds are unavoidable with packet scheduling algorithms of less than $O(\log N)$ time complexity. GR^3's $O(g^2)$ error bound and $O(1)$ time complexity are consistent with this analysis, since delay and service error are not equivalent concepts. Thus, if adapted to packet scheduling, GR^3 would worst-case incur $O(N)$ delay while preserving an $O(g^2)$ service error.

Previous work in proportional share scheduling has focused on scheduling a single resource and little work has been done in proportional share multiprocessor scheduling. WRR and fair-share multiprocessor schedulers have been developed, but have the fairness problems inherent in those approaches. The only multiprocessor fair queueing algorithm that has been proposed is Surplus Fair Scheduling (SFS) [7]. SFS also adapts a uniprocessor algorithm, SFQ, to multiple processors using a centralized run queue. No theoretical fairness bounds are provided. If a selected client is already running on another processor, it is removed from the run queue. This operation may introduce unfairness if used in low overhead, round-robin variant algorithms. In contrast, GR^3MP provides strong fairness bounds with lower scheduling overhead.

SFS introduced the notion of *feasible* clients along with a $O(P)$-time weight readjustment algorithm, which requires however that the clients be sorted by their original weight. By using its grouping strategy, GR^3MP performs the same weight readjustment in $O(P)$ time without the need to order clients, thus avoiding SFS's $O(\log N)$ overhead per maintenance operation. The optimality of SFS's and our weight readjustment algorithms rests in preservation of ordering of clients by weight and of weight proportions among feasible clients, and not in minimal overall weight change, as [7] claims.

7 Conclusions and Future Work

We have designed, implemented, and evaluated Group Ratio Round-Robin scheduling in the Linux operating system. We prove that GR^3 is the first and only $O(1)$ uniprocessor and multiprocessor scheduling algorithm that guarantees a service error bound of less than $O(N)$ compared to an idealized processor sharing model, where N is the number of runnable clients. In spite of its low complexity, GR^3 offers better fairness than the $O(N)$ service error bounds of most fair queuing algorithms that need $O(\log N)$ time for their operation. GR^3 achieves these benefits due to its grouping strategy, ratio-based intergroup scheduling, and highly efficient intragroup round robin scheme with good fairness bounds. GR^3 introduces a novel frontlog mechanism and and weight readjustment algorithm to schedule small-scale multiprocessor systems while preserving its good bounds on fairness and time complexity.

Our experiences with GR^3 show that it is simple to implement and easy to integrate into existing commodity operating systems. We have measured the performance of GR^3 using both simulations and kernel measurements of real system performance using a prototype Linux implementation. Our simulation results show that GR^3 can provide more than two orders of magnitude better proportional fairness behavior than other popular proportional share scheduling algorithms, including WRR, WFQ, SFQ, VTRR, and SRR. Our experimental results using our GR^3 Linux implementation further demonstrate that GR^3 provides accurate proportional fairness behavior on real ap-

plications with much lower scheduling overhead than other Linux schedulers, especially for larger workloads.

While small-scale multiprocessors are the most widely available multiprocessor configurations today, the use of large-scale multiprocessor systems is growing given the benefits of server consolidation. Developing accurate, low-overhead proportional share schedulers that scale effectively to manage these large-scale multiprocessor systems remains an important area of future work.

Acknowledgements

Chris Vaill developed the scheduling simulator and helped implement the kernel instrumentation used for our experiments. This work was supported in part by NSF grants EIA-0071954 and DMI-9970063, an NSF CAREER Award, and an IBM SUR Award.

References

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

[2] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, Sebastopol, CA, second edition, 2002.

[3] R. Bryant and B. Hartner. Java Technology, Threads, and Scheduling in Linux. IBM developerWorks Library Paper. IBM Linux Technology Center, Jan. 2000.

[4] B. Caprita, W. C. Chan, and J. Nieh. Group Round-Robin: Improving the Fairness and Complexity of Packet Scheduling. Technical Report CUCS-018-03, Columbia University, June 2003.

[5] W. C. Chan. Group Ratio Round-Robin: An O(1) Proportional Share Scheduler. Master's thesis, Columbia University, June 2004.

[6] W. C. Chan and J. Nieh. Group Ratio Round-Robin: An O(1) Proportional Share Scheduler. Technical Report CUCS-012-03, Columbia University, Apr. 2003.

[7] A. Chandra, M. Adler, P. Goyal, and P. 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.

[8] S. Y. Cheung and C. S. Pencea. BSFQ: Bin-Sort Fair Queueing. In *Proceedings of IEEE INFOCOM '02*, pages 1640–1649, New York, NY, June 2002.

[9] G. Chuanxiong. SRR: An O(1) Time Complexity Packet Scheduler for Flows in Multi-Service Packet Networks. In *Proceedings of ACM SIGCOMM '01*, pages 211–222, San Diego, CA, Aug. 2001.

[10] J. Cobb, M. Gouda, and A. El-Nahas. Time-Shift Scheduling – Fair Scheduling of Flows in High-Speed Networks. *IEEE/ACM Transactions on Networking*, pages 274–285, June 1998.

[11] 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.

[12] R. Essick. An Event-Based Fair Share Scheduler. In *Proceedings of the Winter 1990 USENIX Conference*, pages 147–162, Berkeley, CA, Jan. 1990. USENIX.

[13] 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.

[14] G. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1857, Oct. 1984.

[15] J. Kay and P. Lauder. A Fair Share Scheduler. *Commun. ACM*, 31(1):44–55, 1988.

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

[17] J. Nieh, C. Vaill, and H. Zhong. Virtual-Time Round-Robin: An O(1) Proportional Share Scheduler. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 245–259, Berkeley, CA, June 2001. USENIX.

[18] 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.

[19] S. Ramabhadran and J. Pasquale. Stratified Round Robin: a Low Complexity Packet Scheduler with Bandwidth Fairness and Bounded Delay. In *Proceedings of ACM SIGCOMM '03*, pages 239–250, Karlsruhe, Germany, Aug. 2003.

[20] J. Rexford, A. G. Greenberg, and F. Bonomi. Hardware-Efficient Fair Queueing Architectures for High-Speed Networks. In *Proceedings of IEEE INFOCOM '96*, pages 638–646, Mar. 1996.

[21] R. Russell. Hackbench: A New Multiqueue Scheduler Benchmark. `http://www.lkml.org/archive/2001/12/11/19/index.html`, Dec. 2001. Message to Linux Kernel Mailing List.

[22] M. Shreedhar and G. Varghese. Efficient Fair Queueing using Deficit Round Robin. In *Proceedings of ACM SIGCOMM '95*, pages 231–242, Sept. 1995.

[23] D. Stephens, J. Bennett, and H. Zhang. Implementing Scheduling Algorithms in High Speed Networks. *IEEE JSAC Special Issue on High Performance Switches/Routers*, Sept. 1999.

[24] D. Stiliadis and A. Varma. Efficient Fair Queueing Algorithms for Packet-Switched Network. *IEEE/ACM Transactions on Networking*, pages 175–185, Apr. 1998.

[25] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and G. Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 288 – 289, Dec. 1996.

[26] C. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Dept. of EECS, MIT, Sept. 1995.

[27] J. Xu and R. J. Lipton. On Fundamental Tradeoffs between Delay Bounds and Computational Complexity in Packet Scheduling Algorithms. In *Proceedings of ACM SIGCOMM '02*, pages 279–292, Aug. 2002.

[28] L. Zhang. Virtualclock: a New Traffic Control Algorithm for Packet-Switched Networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.