GROUP RATIO ROUND ROBIN: AN $O(1)$ PROPORTIONAL SHARE SCHEDULER

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
OF COLUMBIA UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Wong Chun Chan
June 2004

Acknowledgements

I would like to thank Bogdan Caprita and Chris Vaill for their contributions to this paper. Bogdan contributed to the analysis of GR^3 and Chris provided the scheduler simulator, benchmark programs, and kernel implementation of WFQ and VTRR.

This work was supported in part by an NSF CAREER Award, NSF grant EIA-0071954, and an IBM SUR Award.

Abstract

Proportional share resource management provides a flexible and useful abstraction for multiplexing time-shared resources. We present Group Ratio Round-Robin (GR^3), a proportional share scheduler that combines accurate proportional fairness scheduling behavior with $O(1)$ scheduling overhead. GR^3 uses a novel 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. We analyze the behavior of GR^3 and show that it can provide fairness within a constant factor of the ideal generalized processor sharing model for client weights with a fixed upper bound. GR^3 can be easily implemented using simple data structures. We have implemented GR^3 in Linux and measured its performance against other schedulers commonly used in research and practice, including the standard Linux scheduler, Weighted Fair Queueing, Virtual-Time Round-Robin, and Smoothed Round-Robin. Our experimental results demonstrate that GR^3 can provide much lower scheduling overhead and much better scheduling accuracy in practice than these other approaches.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Proportional share resource management provides a flexible and useful abstraction for multiplexing time-shared processor resources among a set of clients with associated weights. 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.

We introduce Group Ratio Round-Robin (GR^3), a proportional share scheduler that can provide constant fairness bounds on proportional sharing accuracy with $O(1)$ scheduling overhead. 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 processor allocations. Based on this observation, GR^3 uses a novel 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. Specifically, we show that with only $O(1)$ overhead, GR^3 provides fairness within $O(g^2)$ of the ideal Generalized Processing Sharing (GPS) model [13], where g, the number of groups,

is in practice a small constant that grows at worst logarithmically with the largest client weight.

GR^3 is simple to implement and can be easily incorporated into existing scheduling frameworks in commercial operating systems. We have implemented a prototype GR^3 processor scheduler in Linux, and compared our GR^3 Linux prototype against schedulers commonly used in practice and research, including the standard Linux scheduler [2], Weighted Fair Queueing [6], Virtual-Time Round-Robin [15], and Smoothed Round-Robin [4]. We have conducted extensive 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. Furthermore, our results show that GR^3 achieves this accuracy with lower scheduling overhead that is more than an order of magnitude less than the standard Linux scheduler and typical Weighted Fair Queueing implementations. These results demonstrate that GR^3 can in practice deliver better proportional share control with lower scheduling overhead than these other approaches.

This paper presents the design, analysis, and evaluation of GR^3. Chapter 2 presents the GR^3 scheduling algorithm. Chapter 3 analyzes the fairness and complexity of GR^3. Chapter 4 presents performance results from both simulation studies and real kernel measurements that compare GR^3 against other well-known scheduling algorithms. Chapter 5 discusses related work. Finally, we present some concluding remarks and directions for future work.

Chapter 2

GR^3 Scheduling

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. Without loss of generality, we can model the process of scheduling a time-multiplexed resource among a set of clients in two steps: 1) the scheduler orders the clients in a queue, 2) the scheduler runs 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. Note that the time quantum is typically expressed in time units of constant size determined by the hardware. As a result, we refer to the units of time quanta as time units (tu) in this paper rather than an absolute time measure such as seconds.

Based on the above scheduler model, a scheduler can achieve proportional sharing in one of two ways. One way 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. Fair queueing algorithms [6, 16, 24, 9, 21, 5] are common examples of this approach. However, adjusting the position of the client in the queue typically requires sorting clients based on some metric of fairness. This can be expensive since it requires an operation with time complexity that grows with the number of clients. The other way is to adjust the size of the time quantum of a client so that it runs longer for a given allocation. Weighted round-robin is the most common example of this approach. This approach 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.

GR^3 is a proportional share scheduler that schedules with $O(1)$ time complexity like round-robin scheduling but with much better proportional fairness guarantees in practice. In designing GR^3, we observed that accurate, low-overhead proportional sharing is easy to achieve when all clients have equal weights, but is harder to do when clients have skewed weight distributions. Based on this observation, GR^3 uses a novel client grouping strategy to organize clients into groups of similar weights which can be more easily scheduled. GR^3 then combines two scheduling algorithms: (1) an intergroup scheduling algorithm to select a group from which to select a client to execute, and (2) an intragroup scheduling algorithm to select a client from within the selected group to execute. 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. Each group of order k is assigned 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**: Once a group has been selected, a client within the group 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 such 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. As a result, the intergroup scheduler never needs to schedule a large number of groups. The client grouping strategy also limits the weight distributions that the intragroup scheduler needs to consider since the range of client weights within a group is less than a factor of two. As a result, the intragroup scheduler never needs to schedule clients with skewed weight distributions since the clients within a group must have relatively similar weights. Note that 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

To provide a more in depth description of GR^3, we first define more precisely the state GR^3 associates with each client and group, and then describe in detail how GR^3 uses that state to schedule clients. Table 2.1 presents a list of terminology we use. In GR^3, a client has three values associated with its execution state: weight, deficit, and run state. A client's *weight* defines its resource rights. 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* is an indication of whether or not the client can be executed. A client is *runnable* if it can be executed. For example for a CPU scheduler, a client would not be runnable if it is blocked waiting for I/O and cannot execute.

A group in GR^3 has a similar set of values associated with it: group weight, group order, group work, and current client. The *group weight* is defined as the sum of the corresponding weights of the clients in the group run queue. The *group order* uniquely identifies a group in the group list and determines the clients that are in the group. A group with *group order* k contains clients with weights between 2^k to $2^{k+1}-1$. The *group work* is a measure of the total execution time clients in the group

C_i	Client i.
ϕ_i	The weight assigned to C_i.
D_{C_i}	The deficit of C_i.
N	The number of runnable clients in the system.
Φ_T	The sum of the weights of all runnable clients: $\sum_{C_j} \phi_j$.
g	The number of groups.
N_G	The number of clients in group G.
G_i	Group with index i in the ordered list of groups.
$G(i)$	The group to which C_i belongs.
Φ_G	The group weight: $\sum_{C_i \in G} \phi_i$.
Φ_i	Shorthand notation for Φ_{G_i} .
σ_G	The order of group G.
W_C	The work of client C.
W_G	The group work of group G.
W_k	shorthand notation for W_{G_k}.
W_T	The sum of the group work of all groups.

Table 2.1: GR^3 Terminology

have received. The *current client* is the most recently scheduled client in the group's run queue.

In addition to the per client and per group state described, GR^3 maintains the following scheduler state: time quantum, group list, total weight, total work, and current group. The *time quantum* is the duration of a standard time slice assigned to a client to execute. The *group list* is a sorted list of all groups containing runnable clients ordered from largest to smallest group weight. If two groups have equal group weights, the group order is used to break the tie in ordering the groups on the group list. The *total weight* is the sum of the weights of all runnable clients. The *total work* is the sum of the work of all groups. The *current group* is the most recently selected group in the group list.

2.2 Basic GR^3 Algorithm

We will initially only consider runnable clients in our discussion of the basic GR^3 scheduling algorithm. We will discuss dynamic changes in a client's run state in

Chapter 2.3. We first focus on the development of the GR^3 intergroup scheduling algorithm and then discuss the development of the GR^3 intragroup scheduling algorithm.

The GR^3 **intergroup scheduling** algorithm essentially uses the ratio of the group weights of successive groups to determine which group to select. The basic idea is 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. In other words, since groups are ordered from largest group weight to smallest group weight, GR^3 selects a group G_{i+1} with a smaller group weight for one time only after its previous group G_i with a larger group weight is selected a number of times that is equal to the ratio of the group weights of G_i and G_{i+1}.

While this algorithm can be formulated in terms of ratios of group weights, it is simpler to formulate it in terms of the work of each group. The work of a group increases by one each time that a client in the group is run for one time quantum. The GR^3 intergroup scheduler initially sets the current group to be the first group in the group list, which is the group with the largest group weight. Once a group is selected as the current group, the intragroup scheduler schedules the current client from the current group's run queue to run for one time quantum. Once the current client has completed its time quantum, the work of the current group is incremented by one. GR^3 will compare the work of the current group G_i and the next group G_{i+1} in the group list using the following inequality:

$$\frac{W_i + 1}{W_{i+1} + 1} > \frac{\Phi_i}{\Phi_{i+1}} \tag{2.1}$$

If the inequality holds, GR^3 will assign the next group on the group list to be the current group and select a client from that group. Otherwise GR^3 will reset the current group to the largest weight group at the beginning of the group list.

The intuition behind (2.1) 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. For each time a client from G_{i+1} is run, GR^3 would like to

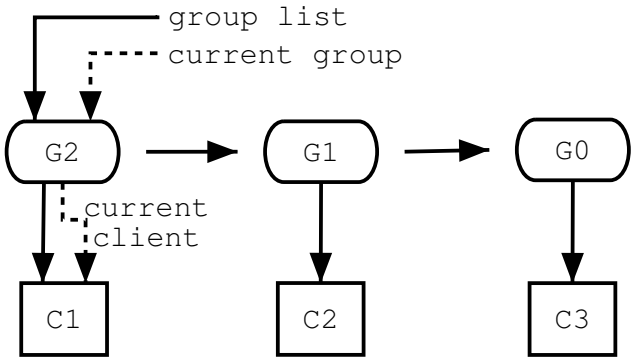

Figure 2.1: GR^3 intergroup scheduler: Clients C_1, C_2 and C_3 with weights 5, 2, and 1 are in groups G_2, G_1 and G_0. Initially, the current group and current client are G_2 and C_1.

have run $\frac{\Phi_i}{\Phi_{i+1}}$ worth of clients from G_i. Another way to think of this is that when GR^3 selects a client from group G_{i+1} to run, it would like the work of G_{i+1} to have caught up to its proportional share allocation once that client has run for a time quantum. GR^3 does not want to run clients from G_i so often that running a client from G_{i+1} still leaves the work of G_{i+1} behind its proportional share allocation. (2.1) 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 this works, consider an example in which we have 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 C_1 in group G_2 with group order 2, C_2 in group G_1 with group order 1, and C_3 in group G_0 with group order 0, resulting in three groups with group weights of 5, 2, and 1, respectively. In this example, each group has only one client so there is no intragroup scheduling and each group's unique client is selected to run when the group is selected through intergroup scheduling. Assume that the work of each group is initial zero. GR^3 first orders the groups from largest to smallest group weight. GR^3 would start by selecting group G_2 and running client C_1. Figure 2.1 shows the clients and their respective groups on the group list. Its group work would then be incremented to 1. Based on (2.1), $\frac{W_{G(1)}+1}{W_{G(2)}+1} = 2 < \frac{\Phi_{G(1)}}{\Phi_{G(2)}} = 2.5$, so GR^3 would

select G_2 again and run client C_1. After running C_1, G_2's work would be 2 so that the inequality in (2.1) would hold and GR^3 would then move on to the next group G_1 and run client C_2. Based on (2.1), $\frac{W_{G(2)}+1}{W_{G(3)}+1} = 2 \leq \frac{\Phi_{G(2)}}{\Phi_{G(3)}} = 2$, so GR^3 would reset the current group to the largest weight group G_2 and run client C_1. Based on (2.1), C_1 would be run for three time quanta before selecting G_1 again to run client C_2. After running C_2 the second time, the work of G_1 would increase such that $\frac{W_{G(2)}+1}{W_{G(3)}+1} = 3 > \frac{\Phi_{G(1)}}{\Phi_{G(2)}} = 2$, so GR^3 would then move on to the last group G_0 and run client C_3. The resulting repeating order in time units in which GR^3 selects the groups would then be: G_2, G_2, G_1, G_2, G_2, G_2, G_1, G_0. Each group therefore receives its proportional allocation in accordance with its respective group weight.

The GR^3 **intragroup scheduling** algorithm works with the intergroup scheduling algorithm to select a client from a group to run once a group has been selected. Because of the GR^3 grouping strategy, clients within a group have weights that differ by less than a factor of two. GR^3 uses this fact and normalizes all client weights in a group G with respect to the minimum possible weight, $\phi_{min} = 2^{\sigma_G}$, for any client in the group. GR^3 then effectively runs each client within a group in round-robin order for a number of time quanta equal to the client's normalized weight, rounded down to the nearest integer value. GR^3 keeps track of fractional time quanta that are not used and accumulates them in a deficit value for each client, then allocates an extra time quantum to a client when its deficit reaches one.

More specifically, the GR^3 intragroup scheduler considers the scheduling of clients in rounds. A *round* is one pass through a group'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. For each runnable client C_i, the scheduler determines the maximum number of time quanta that the client can be selected to run in this round as $\lfloor \frac{\phi_i}{\phi_{min}} + D_{C_i}(r-1) \rfloor$. $D_{C_i}(r)$, the deficit of client C_i after round r, is defined recursively as $D_{C_i}(r) = \frac{\phi_i}{\phi_{min}} + D_{C_i}(r-1) - \lfloor \frac{\phi_i}{\phi_{min}} + D_{C_i}(r-1) \rfloor$, with $D_{C_i}(0) = 0$. Thus, in each round, C_i is allotted one time quantum plus any additional leftover from the previous round, and $D_{C_i}(r)$ keeps track of the amount of service that C_i missed because of rounding down its allocation to whole time quanta. We observe that $0 \leq D_{C_i}(r) < 1$ after any

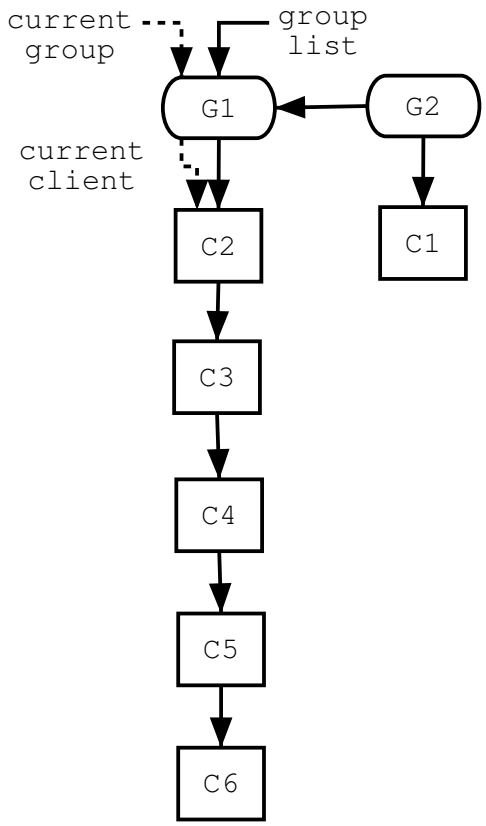

Figure 2.2: GR^3 intragroup scheduler: Clients C_1, C_2, C_3, C_4, C_5 and C_6 with weights 12, 3, 3, 2, 2, and 2 are in groups G_1 and G_2. Initially, the current group and current client are G_1 and C_1.

round r so that any client C_i 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 execute 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).

Consider the following example to illustrate further how GR^3 scheduling works. Consider a set of six clients that need to be scheduled, one client C_1 with weight 12, two clients C_2 and C_3 each with weight 3, and the other three clients C_4, C_5, and C_6 each with weight 2. Assume that all clients are runnable for simplicity. The six clients will be put into two groups G_1 and G_2 with respective group order 1 and 2

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 $\Phi_1 = 12$ and $\Phi_2 = 12$. Figure 2.2 shows the clients and their respective groups on the group list. 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_2 will schedule client C_1 every time G_2 is considered for service since it has only one client. We will show the order of service for the first two rounds of G_1. Rounds 3 and 4 of G_1 has the same order of service as rounds 1 and 2. Since $\phi_{\min(G_1)} = 2$, 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 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 $\frac{\phi_i}{\phi_{min}}$ allocation. 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 sequence of clients that the scheduler runs for each unit of time is 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.

We considered an alternative method for the intragroup scheduler. In the alternative approach, the intragroup scheduler determines at the beginning of each round which clients within a group have not exceeded their proportional share of service with respect to the group, and runs those clients during the round. To do this, GR^3 sets the last group work W_G^{last} equal to the group work W_G at the beginning of a round. Given a client C_i with weight ϕ_i and work W_C in a group G with group weight Φ_G, the intragroup scheduler runs a client if the following inequality holds when $W_G^{last} > 0$:

$$\frac{\phi_i}{\Phi_G} \leq \frac{W_C}{W_G^{last}} \tag{2.2}$$

The result of this intragroup scheduling algorithm will be that clients within a group will be scheduled to run in a round-robin order but that some clients will be skipped some of the time.

Consider the situation where a group has n clients and $n-1$ of the clients have weights close to the minimum group weight and only 1 client with weight that's

slightly less than twice of the minimum group weight. Using the alternative intragroup scheduler, all of the clients except one, will be skipped almost every other time the clients are considered for running. Thus, the overhead to select a client within the group becomes O(n). This is not the case with the current intragroup scheduler we used. Under the current intragroup scheduler, this scenario would still take O(1) time to schedule a client within the group.

2.3 GR^3 Dynamic Considerations

In the previous section, we presented the basic GR^3 scheduling algorithm, but we did not discuss how GR^3 deals with dynamic considerations that are a necessary part of any on-line scheduling algorithm. We now discuss how GR^3 allows clients to be dynamically created, terminated, or change run state.

We distinguish between clients that are runnable and not runnable. As mentioned earlier, clients that are runnable can be selected for execution by the scheduler, while clients that are not runnable cannot. Only runnable clients are placed in the run queue. 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_i with weight ϕ_i becomes runnable, it is inserted into group $G(i)$ such that ϕ_i is between $2^{\sigma_{G(i)}}$ and $2^{\sigma_{G(i)}+1} - 1$. If the group was previously empty, the client becomes the current client of the group. If the group was not previously empty, GR^3 inserts the client into the respective group's run queue right before the current client. This requires the newly runnable client to wait its turn to be serviced until all of the other clients in the group have first been considered for scheduling since the other clients were already in the run queue waiting to execute.

When a newly runnable client C_i is inserted into its respective group $G(i)$, the group needs to be moved to its new position on the ordered group list based on its new group weight. The corresponding group work and group weight 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 want to scale the group work of $G(i)$ in a similar manner.

Denote $W^{old}_{G(i)}$ as the group work of $G(i)$ and W^{old}_T as the total work before inserting client C_i, respectively. We then scale the group work $W_{G(i)}$ as follows:

$$W_{G(i)} = \lfloor W^{old}_{G(i)} \frac{\Phi_{G(i)}}{\Phi^{old}_{G(i)}} \rfloor \tag{2.3}$$

if the group $G(i)$ was non-empty or:

$$W_{G(i)} = \lfloor W^{old}_T \frac{\phi_i}{\Phi^{old}_T} \rfloor \tag{2.4}$$

otherwise. In both cases, we need to update the total work counter:

$$W_T = W^{old}_T + W_{G(i)} - W^{old}_{G(i)} \tag{2.5}$$

Also, since we have decreased the average work in the group through these operations, we need to set the deficit of C_i so that the future increase in service given to the group because of this decrease should be absorbed by the new client. The idea is to have the impact of a new client insertion be as local as possible, preserving the relationship among the work of the other clients and groups. We therefore assign an initial deficit as follows:

$$D_{C_i} = \frac{\Phi_{G(i)}}{\Phi^{old}_{G(i)}} W^{old}_{G(i)} - \lfloor \frac{\Phi_{G(i)}}{\Phi^{old}_{G(i)}} W^{old}_{G(i)} \rfloor \tag{2.6}$$

if the group $G(i)$ was non-empty or:

$$D_{C_i} = \frac{\phi_i}{\Phi_T} W^{old}_T - \lfloor \frac{\phi_i}{\Phi_T} W^{old}_T \rfloor \tag{2.7}$$

otherwise. Since this deficit is less than 1, the new client is mildly compensated for having to wait an entire round until it gets to run, while not obtaining more credit than other, already runnable, clients.

Consider again the example shown in Figure 2.1, which was used to illustrate how the intergroup scheduler works. Suppose we want to insert a client C_4 with a weight 2 after the following order of groups was run: G_2, G_2, and G_1. C_4 is inserted into

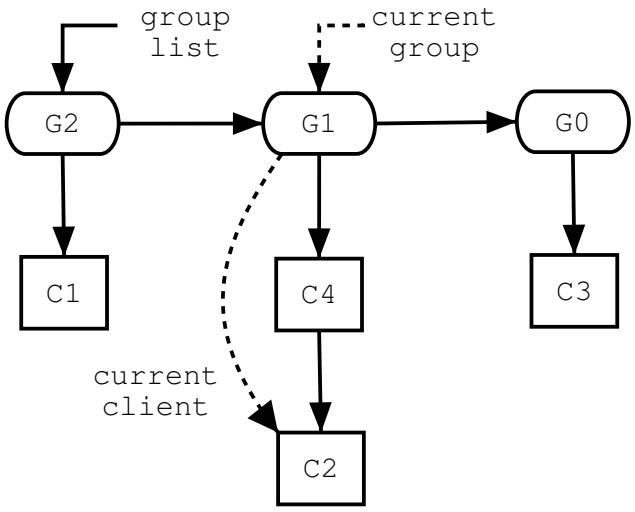

Figure 2.3: GR^3 dynamic client insertion: Client C_4 is inserted into group G_1, the same group as C_2.

group G_1, the same group as C_2. The new group weight and group work for G_1 are 4 and 2. The total weight and total work counter for all clients become 10 and 4. C_4 is assigned an initial deficit of 0. Since $\Phi_{G(2)}$ is still less than $\Phi_{G(1)}$, group G_1 will not move its position on the ordered group list. Figure 2.3 is the clients and groups in the ordered group list after inserting C_4. Since $\frac{W_{G(2)}+1}{W_{G(3)}+1} = 2.5 \leq \frac{\Phi_{G(2)}}{\Phi_{G(3)}} = 4$, GR^3 would select G_2 again and run client C_1.

When a client C_i with weight ϕ_i becomes not runnable, we need to remove it from the group's runqueue. 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 involves overhead and potentially introduces local unfairness by distorting the group structure upon which the scheduler bases its decisions. Intuitively, we would like to preserve the normalized work of the group such that the service rights of the remaining clients will be least affected. There are several ways to accomplish this: preserving the client's credit in the deficit and adding it back when the client becomes runnable again; selecting the optimal position in the runqueue that the client will occupy once it returns; keeping extra state variables or scaling existing ones. These mechanisms, in addition

to their unnecessary complexity, need provisions to guard against a client receiving undue service by exploiting the scheduling algorithm through repeated switching to and from the runnable state. Furthermore, fairness is a fairly flexible concept in dynamic considerations: uncontrollable factors such as the order of the clients in the group's queue, the deficits of other clients, or the particular state of the intragroup scheduler affect the accuracy of any definition of fairness based on received and due service. We have therefore optimized our algorithm to efficiently deal with the common situation when a blocked client may rapidly switch back to the runnable state again. For example, a client might be waiting for a resource that will be released the first time the owner client is scheduled. It is therefore desirable to avoid the overhead associated with adding and removing a client, while at the same time preserving the service rights and service order of the clients. The approach we decided on is based on 'lazy' removal, and has a simple implementation that suits our fairness needs. Since a client blocks when it is running, we know that it will take another full 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 again, we reset the flag and run the client as usual. Otherwise, we remove the client from $G(i)$. In the latter situation, as in the case of client arrivals, the group may need to be moved to its new position on the ordered group list based on its new group weight. The corresponding group weight is updated by simply subtracting the client's weight from the group weight. The corresponding group work is scaled in a similar manner as for client insertion:

$$W_{G(i)} = \lceil W_{G(i)}^{old} \frac{\Phi_{G(i)}}{\Phi_{G(i)}^{old}} \rceil \tag{2.8}$$

Finally, the total work counter needs to be updated:

$$W_T = W_T^{old} + W_{G(i)} - W_{G(i)}^{old} \tag{2.9}$$

After having performed these removal operations, we restart the scheduler from the largest weight group in the system.

The fairness we achieve by this technique is as follows: whenever a client blocks during round r, we set $D_{C_i}(r) = min(D_{C_i}(r-1) + \frac{\phi_i}{\phi_{min}} - \lceil W(i,r) \rceil, 1)$, where $W(i,r)$ 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. The intuition is that a client shouldn't be penalized for a quick blocking interval. 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 restart the scheduler from the first 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.

Chapter 3

GR^3 Fairness and Complexity

A proportional share scheduler is more accurate if it allocates resources in a manner that is more proportionally fair. We can formalize this notion of proportional fairness by defining the *service error*, a measure widely used ([8, 10, 18, 22]) in the analysis of scheduling algorithms. To simplify the discussion, we assume that clients are always runnable in the following analysis. Our definition draws heavily from Generalized Processor Sharing (GPS) [13].

GPS is an idealized discipline in the sense that it achieves *perfect fairness*: $W_C = W_T \frac{\phi_C}{\Phi_T}$, an ideal state in which each client has received service exactly proportional to its share. Although all real-world scheduling algorithms 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. To evaluate the fairness performance of a proportional sharing mechanism, we must quantify how close an algorithm gets to perfect fairness. For this purpose, we use three measures of service error. The client service time error is the difference between the service received by client C and its share of the total work done by the processor: $E_C = W_C - \phi_C \frac{W_T}{\Phi_T}$. The group service time error is a similar measure for groups that quantifies the fairness of allocating the processor among groups: $E_G = W_G - \Phi_G \frac{W_T}{\Phi_T}$. The group relative service time error represents the service time error of client C if there were only a single group G 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: $E_{C,G} = W_C - \phi_C \frac{W_G}{\Phi_G}$. A positive service

time error indicates that a client or group 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 or group C has received beyond its ideal allocation. The goal of a proportional share scheduler should be to minimize the absolute value of the allocation error between clients with minimal scheduling overhead.

We analyze the fairness of GR^3 by providing bounds on its service error. 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. Finally, we combine these results to obtain the client service error bounds for the overall scheduler. We also discuss the scheduling overhead of GR^3 in terms of its time complexity.

3.1 Intergroup Fairness

Let the GR^3 scheduler organize the clients into g groups, according to the clients' weights, such that group k has a group weight Φ_k, where $\Phi_1 \geq \Phi_2 \geq ... \geq \Phi_k \geq \Phi_{k+1} \geq ... \geq \Phi_g$.

Let us negate (2.1) under the form:

$$\frac{W_j + 1}{\Phi_j} \leq \frac{W_{j+1} + 1}{\Phi_{j+1}} \tag{3.1}$$

After having selected G_j, GR^3 will select G_{j+1} if and only if (3.1) is violated.

To show the intuition behind the fairness mechanism of GR^3, we begin by assuming the weight ratios of consecutive groups in the group list are integers. For this case, we state and prove the following:

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

Proof: Let us consider the decision faced by the intergroup scheduler after having selected some group G_j. When W_j becomes $(W_{j+1}+1)\frac{\Phi_j}{\Phi_{j+1}} - 1$, (3.1) is still true (we have an equality), and so it will take another selection of G_j before GR^3 will move on

to G_{j+1}. Therefore, after G_{j+1} is selected, the ratio of the work of the two consecutive groups equals the ratio of their weights:

$$\frac{W_j}{\Phi_j} = \frac{W_{j+1}}{\Phi_{j+1}}. \tag{3.2}$$

In particular, let the last selected group be G_k; then we know (3.2) holds for all $1 \leq j < k$. If $j > k$, then we know (3.2) held after G_{j+1} was selected. Until the next time that G_{j+1} gets selected again and (3.2) holds once more, W_j can only increase, with W_{j+1} fixed. Thus,

$$\frac{W_{j+1}}{\Phi_{j+1}} \leq \frac{W_j}{\Phi_j} \leq \frac{W_{j+1}+1}{\Phi_{j+1}} - \frac{1}{\Phi_j}. \tag{3.3}$$

The right inequality in (3.3) is just (3.1), slightly rearranged.

For the particular case when $j = k$, we can write based on (3.3) just before having selected G_k (when W_k was less by 1):

$$\frac{W_{k+1}}{\Phi_{k+1}} + \frac{1}{\Phi_k} \leq \frac{W_k}{\Phi_k} \leq \frac{W_{k+1}+1}{\Phi_{k+1}}. \tag{3.4}$$

By summing (3.3) over $k < j < i$ and adding (3.4), we get $\frac{W_i}{\Phi_i} + \frac{1}{\Phi_k} \leq \frac{W_k}{\Phi_k} \leq \frac{W_i+1}{\Phi_i} \forall i$, $k < i \leq g$. Also, from (10), we have $\frac{W_k}{\Phi_k} = \frac{W_i}{\Phi_i} \forall i$, $1 \leq i < k$.

Multiplying by Φ_i and summing over all i, we get

$$W_T + \frac{1}{\Phi_k} \sum_{i=k+1}^{g} \Phi_i \leq W_k \frac{\Phi_T}{\Phi_k} \leq W_T + g - k. \tag{3.5}$$

Therefore, right after G_k is selected, its error $E_{G_k} = W_k - W_T \frac{\Phi_k}{\Phi_T}$ lies between $\frac{1}{\Phi_T} \sum_{i=k+1}^{g} \Phi_i \in (0,1)$ and $(g-k)\frac{\Phi_k}{\Phi_T}$. Since the minimum error will occur right before some selection of G_k, when W_k is less by 1 than in the above analysis, we can bound the negative error by -1. □

We immediately observe that the error is maximized for $k = 1$; thus:

Corollary 1 *If* $\frac{\Phi_j}{\Phi_{j+1}} \in \boldsymbol{N}$, $1 \leq j < g$, *then* $-1 < E_G < (g-1)$ *for any group* G.

Let us now turn to the general case, when group weight ratios are rational numbers.

We present the general group error bounds under the following form:

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

Proof: The proof follows the discussion in the integer weight ratio case, starting with some important remarks related to selecting some group G_j. First, we make the observation that (3.1) was invalid just before G_{j+1} was selected, but held the previous time when G_j was selected. Thus,

$$\frac{W_j}{\Phi_j} \leq \frac{W_{j+1}}{\Phi_{j+1}} < \frac{W_j + 1}{\Phi_j} \tag{3.6}$$

holds immediately after G_{j+1} is selected. Furthermore, selecting G_{j+1} has the consequence of making (3.1) valid again, since

$$\frac{W_j}{\Phi_j} + \frac{1}{\Phi_j} \leq \frac{W_{j+1}}{\Phi_{j+1}} + \frac{1}{\Phi_j} \leq \frac{W_{j+1}}{\Phi_{j+1}} + \frac{1}{\Phi_{j+1}}. \tag{3.7}$$

Also, the right inequality in (3.6) is true in general, since after G_{j+1} was selected and (3.6) held, W_j could have only increased, while W_{j+1} stayed fixed.

Now assume that the last group to have been selected is G_k. Based on the above, we will derive relationships between W_j and W_{j+1} depending on whether $j < k$, $j = k$, or $j > k$.

1. $j < k$ When G_k is selected, we know that G_{j+1} was selected right after G_j for all $j < k$, and so (3.6) holds for all $j < k$.

2. $j > k$ We use (3.7) and the right inequality of (3.6) to obtain

$$\frac{W_{j+1}}{\Phi_{j+1}} < \frac{W_j + 1}{\Phi_j} \leq \frac{W_{j+1} + 1}{\Phi_{j+1}} \tag{3.8}$$

3. $j = k$ The situation right before G_k got selected is an instance (with $j = k$) of the previous case, with the observation that the new W_k is the old $W_k + 1$. Thus,

$$\frac{W_{k+1}}{\Phi_{k+1}} < \frac{W_k}{\Phi_k} \leq \frac{W_{k+1} + 1}{\Phi_{k+1}} \tag{3.9}$$

Let us now turn our attention to finding the bounds for the group errors.

By summing up (3.6) over j between i and $k-1$, we get

$$\frac{W_i}{\Phi_i} \leq \frac{W_k}{\Phi_k} < \frac{W_i}{\Phi_i} + \sum_{j=i}^{k-1} \frac{1}{\Phi_j},\ 1 \leq i < k.$$

Similarly, by summing up (3.8) over j between $k+1$ and i and adding (3.9), we get

$$\frac{W_i}{\Phi_i} - \sum_{j=k+1}^{i-1} \frac{1}{\Phi_j} < \frac{W_k}{\Phi_k} \leq \frac{W_i}{\Phi_i} + \frac{1}{\Phi_i},\ k < i \leq g.$$

We can multiply the above inequalities by Φ_i to obtain:

$$W_i \leq W_k \frac{\Phi_i}{\Phi_k} < W_i + \sum_{j=i}^{k-1} \frac{\Phi_i}{\Phi_j},\ 1 \leq i < k. \tag{3.10}$$

and, respectively,

$$W_i - \sum_{j=k+1}^{i-1} \frac{\Phi_i}{\Phi_j} < W_k \frac{\Phi_i}{\Phi_k} \leq W_i + 1,\ k < i \leq g. \tag{3.11}$$

Adding (3.10) over $1 \leq i < k$ with (3.11) over $k < i \leq g$, and with the identity $W_k = W_k \frac{\Phi_k}{\Phi_k}$, we get:

$$W_T - \sum_{i=k+1}^{g} \sum_{j=k+1}^{i-1} \frac{\Phi_i}{\Phi_j} < W_k \frac{\Phi_T}{\Phi_k} \leq W_T + \sum_{i=1}^{k-1} \sum_{j=i}^{k-1} \frac{\Phi_i}{\Phi_j} + g - k. \tag{3.12}$$

We notice that $\sum_{i=1}^{k-1} \sum_{j=i}^{k-1} \frac{\Phi_i}{\Phi_j} = \sum_{i=1}^{k-1} (\frac{1}{\Phi_i} \sum_{j=1}^{i} \Phi_j) < \sum_{i=1}^{k-1} (\frac{\Phi_T}{\Phi_i}) < (k-1)\frac{\Phi_T}{\Phi_k}$.
Also, $\sum_{i=k+1}^{g} \sum_{j=k+1}^{i-1} \frac{\Phi_i}{\Phi_j} \leq \sum_{i=k+1}^{g} \sum_{j=k+1}^{i-1} 1 = \sum_{i=k+1}^{g} (i-k-1) = \frac{(g-k)(g-k-1)}{2}$.

(3.12) then yields
$W_T - \frac{(g-k)(g-k-1)}{2} < W_k \frac{\Phi_T}{\Phi_k} < W_T + (k-1)\frac{\Phi_T}{\Phi_k} + g - k$, or
$W_T \frac{\Phi_k}{\Phi_T} - \frac{(g-k)(g-k-1)}{2} \frac{\Phi_k}{\Phi_T} < W_k < W_T \frac{\Phi_k}{\Phi_T} + (k-1) + (g-k)\frac{\Phi_k}{\Phi_T}$, and, since $\frac{\Phi_k}{\Phi_T} \leq 1$,

$$W_T \frac{\Phi_k}{\Phi_T} - \frac{(g-k)(g-k-1)}{2} < W_k < W_T \frac{\Phi_k}{\Phi_T} + (g-1).$$

We rewrite the last relation using the definition for the error: $E_{G_k} = W_k - W_T\frac{\Phi_k}{\Phi_T}$ to get

$$-\frac{(g-k)(g-k-1)}{2} < E_{G_k} < (g-1).$$

The above holds right after G_k is selected. To bound E_{G_k} in general, we note that the minimum of E_{G_k} can only occur right before W_k is incremented (group G_k is selected), while the maximum is reached right after the selection of G_k. Hence, subtracting 1 on the negative side concludes the proof. □

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

Corollary 2 $-\frac{g(g-3)}{2} < E_G < g-1$ *for any group* G.

3.2 Intragroup Fairness

We will show that the due to the tight weight distribution within groups, the group relative error of any client in bound by a constant.

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

Proof: The intragroup Round Robin runs the clients within a group G of order σ_G in the order of the group queue. After r rounds, $W_A = r\frac{\Phi_A}{2^{\sigma_G}} - D_A$ holds for any client A in G. Then, $W_G = \sum_{A\in G} W_A = r\frac{\Phi_G}{2^{\sigma_G}} - \sum_{A\in G} D_A$, and so $E_{C,G} = W_C - W_G\frac{\phi_C}{\Phi_G} = \frac{\phi_C}{\Phi_G}\sum_{A\in G} D_A - D_C$ for any client $C \in G$ after a round.

Depending on the position of C in the group's queue, the error in general will be different from the error in between rounds.

Worst-case, if C happens to be at the head of G's queue, right after it runs, $E_{C,G} = \frac{\phi_C}{\Phi_G}\sum_{A\in G} D_A - D_C + \frac{\phi_C}{2^{\sigma_G}} < N_G\frac{\phi_C}{\Phi_G} + \frac{\phi_C}{2^{\sigma_G}} < N_G\frac{2^{\sigma_G+1}}{N_G 2^{\sigma_G}} + \frac{2^{\sigma_G+1}}{2^{\sigma_G}} = 4$.

Similarly, C might be at the tail of the queue, in which case, right before it runs, $E_{C,G} = \frac{\phi_C}{\Phi_G}\sum_{A\in G} D_A - D_C - \frac{\phi_C}{2^{\sigma_G}} > -1 - \frac{\phi_C}{2^{\sigma_G}} > -1 - \frac{2^{\sigma_G+1}}{2^{\sigma_G}} = -3$. □

3.3 Overall Fairness of GR^3

Given the error bounds for the inter- and intragroup scheduling algorithms, we can now analyze the overall GR^3 fairness.

Lemma 4 *For any client $C \in G$, $E_C = E_{C,G} + \frac{\phi_C}{\Phi_G} E_G$.*

Proof: $E_C = W_C - W_T \frac{\phi_C}{\Phi_T} = W_C - W_G \frac{\phi_C}{\Phi_G} + W_G \frac{\phi_C}{\Phi_G} - W_T \frac{\phi_C}{\Phi_T} = E_{C,G} + \frac{\phi_C}{\Phi_G}(W_G - W_T \frac{\Phi_G}{\Phi_T})$. □

Therefore, We can now give the following theorem to bound the service error relative to GPS of any client in the scheduler to a $O(g)$ positive service error bound and a $O(g^2)$ negative service error bound:

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

Proof: Follows from Corollary 2, Lemma 3, and Lemma 4 where we have $\frac{\phi_C}{\Phi_G} \leq 1$. □

In the desirable case when the group weight ratios are integers, we have the tighter bounds of

Theorem 2 *If $\frac{\Phi_j}{\Phi_{j+1}} \in \boldsymbol{N},\ 1 \leq j < g$ then $-4 < E_C < g + 3$ for any client C.*

Proof: Follows from Corollary 1, Lemma 3, and Lemma 4 where we have $\frac{\phi_C}{\Phi_G} \leq 1$. □

3.4 Complexity of GR^3

GR^3 manages to bound its service error by $O(g^2)$ while maintaining a strict $O(1)$ scheduling overhead. This *time complexity* property of GR^3 is evident from the description of the algorithm. 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 $O(log\ g)$ (using binary search, for example), but the small value of g in practice does not justify tampering with the simplicity of the 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.

Chapter 4

Measurements and Results

To demonstrate the effectiveness of GR^3, we have implemented a prototype GR^3 CPU scheduler in the Linux operating system and measured its 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.

We conducted simulation studies to compare the proportional sharing accuracy of GR^3 against WF^2Q, WFQ, SFQ, WRR, SRR, and VTRR. We used a simulator for these studies for two reasons. First, our simulator enabled us to isolate the impact of the scheduling algorithms themselves and purposefully do not include the effects of other activities present in an actual kernel implementation. Second, our simulator enabled us to examine the scheduling behavior of these different algorithms across hundreds of thousands of different combinations of clients 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. Our simulation results are presented in Chapter 4.1.

We also conducted detailed measurements of real kernel scheduler performance by comparing our prototype GR^3 Linux implementation against the standard Linux 2.4 scheduler, a WFQ scheduler, and a VTRR 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 and proportional share allocation accuracy of these schedulers in a real operating system environment under a number of different workloads. Our measurements of kernel scheduler performance are presented in Chapter 4.2.1.

All of our kernel scheduler measurements were performed on an IBM Netfinity 4500 system with a 933 MHz Intel Pentium III CPU, 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 logs events at significant points in the application and the operating system code. This is done via a light-weight mechanism that writes timestamped event identifiers into a memory log. The mechanism takes advantage of the high-resolution clock cycle counter available with the Intel CPU to provide measurement resolution at the granularity of a few nanoseconds. Getting a timestamp simply involved reading the hardware cycle counter register, which could be read from user-level or kernel-level code. We measured the cost of the mechanism on the system to be roughly 35 ns per event.

The kernel scheduler measurements were performed on a fully functional system to represent a realistic system environment. 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.

4.1 Simulation Studies

We built a scheduling simulator that measures the service time error, described in Chapter 3, of a scheduler on a set of clients. The simulator takes four inputs, the scheduling algorithm, the number of clients N, the total number of weights S, and the number of client-weight combinations. The simulator randomly assigns weights to clients and scales the weight values to ensure that they add up to S. It then schedules the clients using the specified algorithm as a real scheduler would, 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. The simulator assumes that all clients are runnable at all times. This process of random weight allocation and scheduler simulation is repeated for the specified number of client-weight combinations. We then compute an average maximum service time error and average minimum service time error for the specified number of client-weight combinations to obtain an "average-case" error range.

To measure proportional fairness accuracy, we conducted two simulation tests. In the first test, we ran simulations for each scheduling algorithm considered on 32 different combinations of N and S for 2 to 256 number of clients and weights totaling 256 to 2048. For each set of (N, S), we ran 2500 client-weight combinations and determined the resulting average error ranges. The average service time error ranges for WF^2Q, WFQ, SFQ, WRR, SRR, VTRR, and GR^3 are shown in Figures 4.1 to 4.7. Each figure consist 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 S for the same range of values of N and S. Table 4.1 summarizes the minimum and maximum service error ranges for the first test.

WF^2Q, shown in Figure 4.1, has an error range between -1 and 1 tu. WF^2Q is mathematically bounded [1] between -1 and $+1$ tu. Figures 4.2 and 4.3 show the error ranges for WFQ and SFQ. WFQ's error range is between -1 tu and 2 tu, and SFQ's error range is between -2 tu and 1 tu. The negative service time error for WFQ and the positive service time error for SFQ are mathematically bounded by -1 and 1 tu, respectively. Figure 4.4 shows the service time error ranges for WRR. Within the range of values of N and S shown, WRR's error range is between -796 tu and 796 tu. With a time unit of 10 ms per tick as in Linux, a client under WRR can on average get ahead or behind its correct proportional share CPU time allocation by almost 8 seconds, which is a substantial amount of service time error. Figures 4.5 and 4.6 show the service error ranges for SRR and VTRR, respectively. SRR's error

Weight Distribution	Algorithm	Minimum	Maximum
Random	WF^2Q	-1	1
Random	WFQ	-1	2
Random	SFQ	-2	1
Random	WRR	-796	796
Random	SRR	-1.85	1.85
Random	$VTRR$	-4.75	17.86
Random	GR^3	-2.2	2.3
10% Skew	WF^2Q	-1	1
10% Skew	WFQ	-1	25.5
10% Skew	SFQ	-24.6	1
10% Skew	WRR	-848	696
10% Skew	SRR	-11.3	11.1
10% Skew	$VTRR$	-62.7	167.6
10% Skew	GR^3	-2.2	2.4
50% Skew	WF^2Q	-1	1
50% Skew	WFQ	-1	127
50% Skew	SFQ	-127	1
50% Skew	WRR	-1024	1024
50% Skew	SRR	-64	64
50% Skew	$VTRR$	-142.34	94.78
50% Skew	GR^3	-2.2	2.6

Table 4.1: Average service error range with number of clients=[2:256] and total weights=[256:2048]

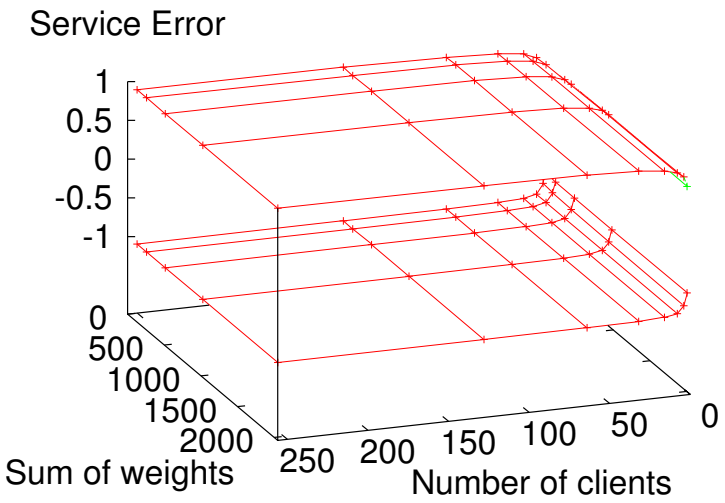

Figure 4.1: WF^2Q service error with number of clients=[2:256], total weights=[256:2048], and random weight distribution

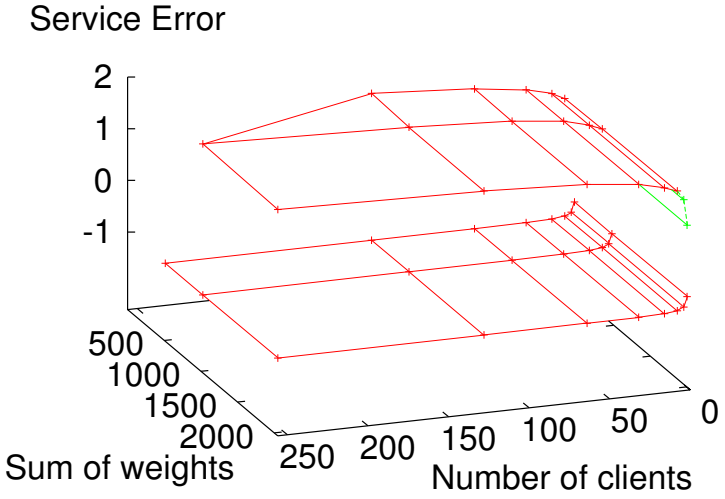

Figure 4.2: WFQ service error with number of clients=[2:256], total weights=[256:2048], and random weight distribution

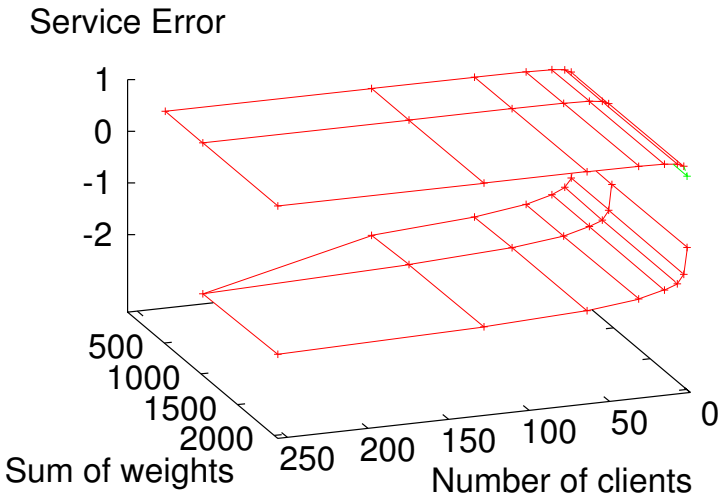

Figure 4.3: SFQ service error with number of clients=[2:256], total weights=[256:2048], and random weight distribution

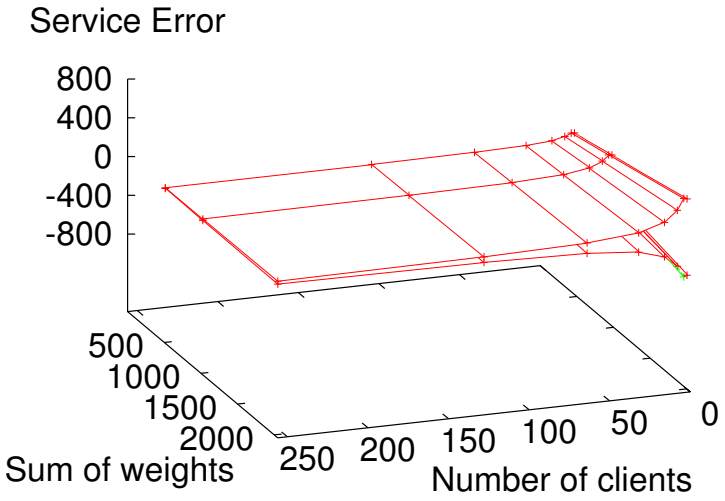

Figure 4.4: WRR service error with number of clients=[2:256], total weights=[256:2048], and random weight distribution

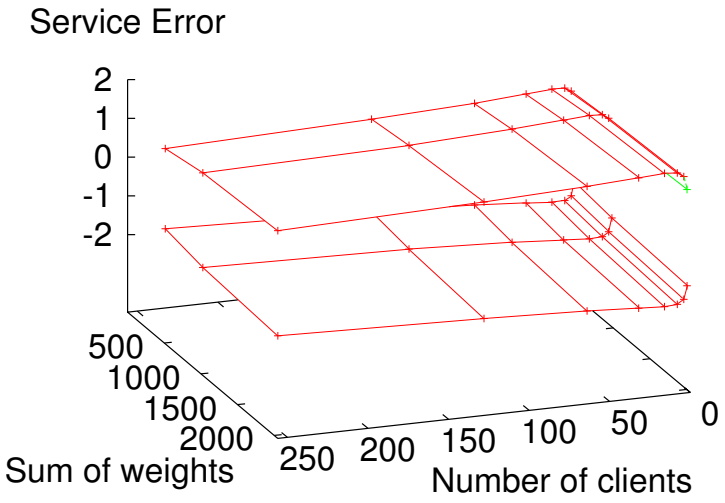

Figure 4.5: SRR service error with number of clients=[2:256], total weights=[256:2048], and random weight distribution

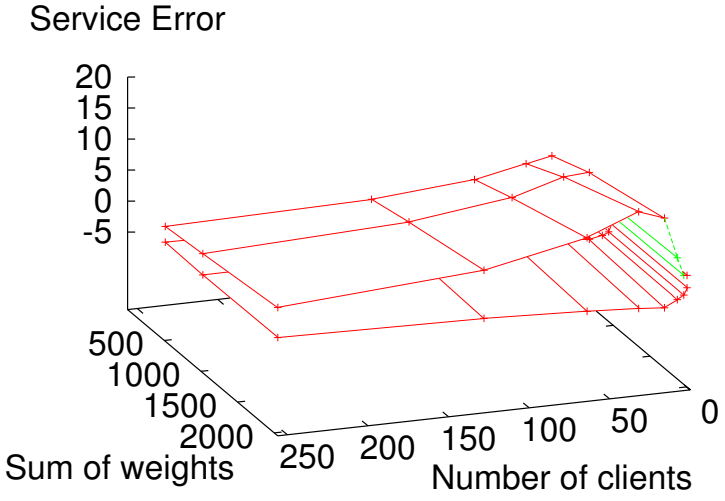

Figure 4.6: VTRR service error with number of clients=[2:256], total weights=[256:2048], and random weight distribution

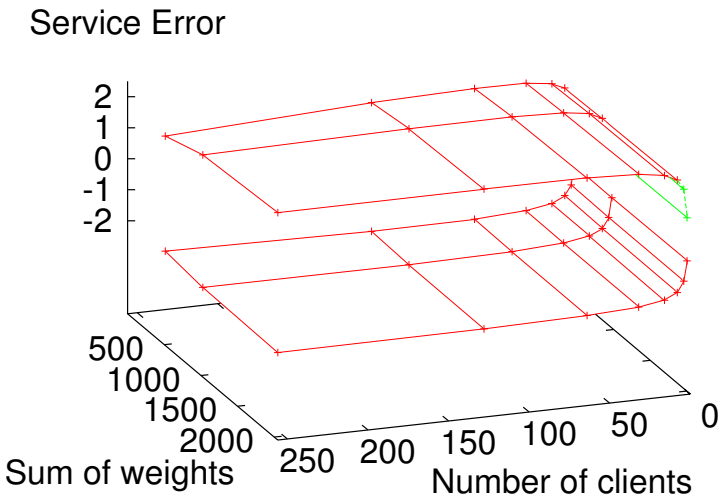

Figure 4.7: GR^3 service error with number of clients=[2:256], total weights=[256:2048], and random weight distribution

range is between -1.85 to 1.85 tu. VTRR's error range is between -4.75 to 17.86 tu. Although WRR and VTRR's error ranges are somewhat worse than WFQ, SFQ, and SRR, WRR and VTRR can schedule in $O(1)$ time. In comparison, Figure 4.7 shows the service time error ranges for GR^3. GR^3's service time error only ranges from -2.2 to 2.3 tu. GR^3 has a smaller error range than WRR and VTRR.

Since the proportional sharing accuracy of a scheduler is often most clearly illustrated with skewed weight distributions, we repeated the simulation studies over the same range of N and S but with one of the clients given a weight equal to 10 and 50 percent of S. All of the other clients were then randomly assigned weights to sum to the remaining 90 and 50 percent of S. We again considered 32 different combinations of N and S. For each set of (N, S), we ran 2500 client-weight combinations and determined the resulting average error ranges. The average service time error ranges for WF^2Q, WFQ, SFQ, WRR, SRR, VTRR, and GR^3 with these skewed weight distributions are shown in Figures 4.8 to 4.21. Each figure shows two surfaces representing the maximum and minimum service time error as a function of N and

S for the respective scheduling algorithm.

Figures 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, and 4.14 show the service time error ranges for WF^2Q, WFQ, SFQ, WRR, SRR, VTRR, and GR^3 with a 10% skewed weight distributions, respectively. WF^2Q still has an error range of -1 to 1 tu. As expected, WRR's error range of -848 to 696 tu was the worst among the seven scheduling algorithms. The error ranges for WFQ, SFQ, SRR, and VTRR are -1 to 25.5 tu, -24.6 to 1 tu, -11.3 to 11.1 tu, and -62.7 to 167.6. Figure 4.14 shows that GR^3's error range is only -2.2 to 2.4 tu. Except for WF^2Q and GR^3, the service time error ranges for all the schedulers increased significantly.

Figure 4.18 shows that WRR's service time error range is -1024 to 1024 tu for 50% skewed weight distribution. WFQ and SFQ's error ranges are -1 to 127 tu and -127 to 1 tu, as shown in Figures 4.16 and 4.17. The error ranges of SRR (Figure 4.19) and VTRR (Figure 4.20 increased to -64 to 64 tu and -142.34 to 94.78 tu. The service error ranges of WFQ, SFQ, SRR, and VTRR are much larger for the 50% skewed weight distributions. Their error ranges increased by more than an order of magnitude from the random weight distribution to 50% skewed weight distribution. In contrast, Figure 4.14 shows that the service time errors for GR^3 remain relatively low even for the 50% skewed weight distributions (-2.2 to 2.6 tu) .

We conducted a second set of simulations on a different range of N and S. We ran simulations for each scheduling algorithm on 45 different combinations of N and S for 32 to 8192 clients and 16384 to 262144 total weights. We ran the simulations where all weights are randomly distributed, one client was given a weight equal to 10 percent of S and one client was given a weight equal to 50 percent of S. For each pair (N, S), we ran 2500 client-weight combinations and determined the resulting average error ranges.

The average service time error ranges for WF^2Q, WFQ, SFQ, WRR, SRR, VTRR, and GR^3 with random weight distributions, 10% skewed weight distributions, and 50% skewed weight distributions are shown in Figures 4.22 to 4.42. The error range results are summarized in Table 4.2. WF^2Q's error range remains at -1 to 1 tu (Figure 4.36). Figure 4.39 shows WRR's service time error for 50% skewed weight distribution is between -7139 tu and 65536 tu. Figure 4.37 shows WFQ's service

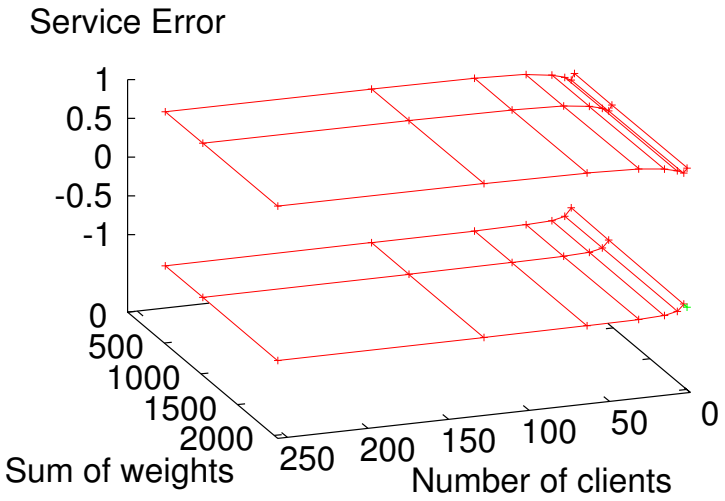

Figure 4.8: WF^2Q service error with number of clients=[2:256], total weights=[256:2048], and 10% skewed weight distribution

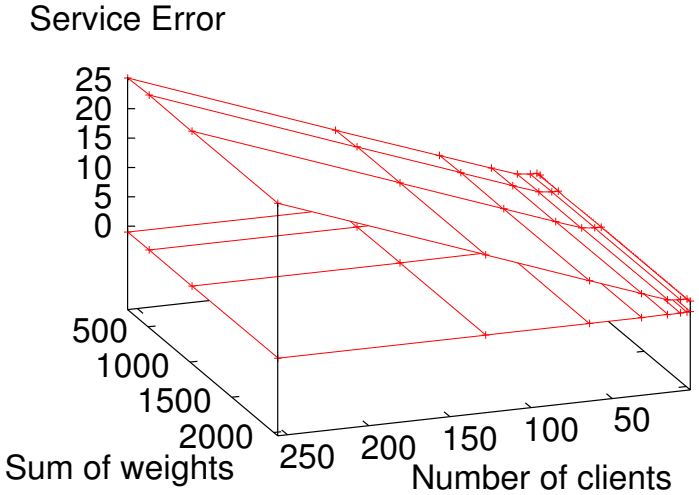

Figure 4.9: WFQ service error with number of clients=[2:256], total weights=[256:2048], and 10% skewed weight distribution

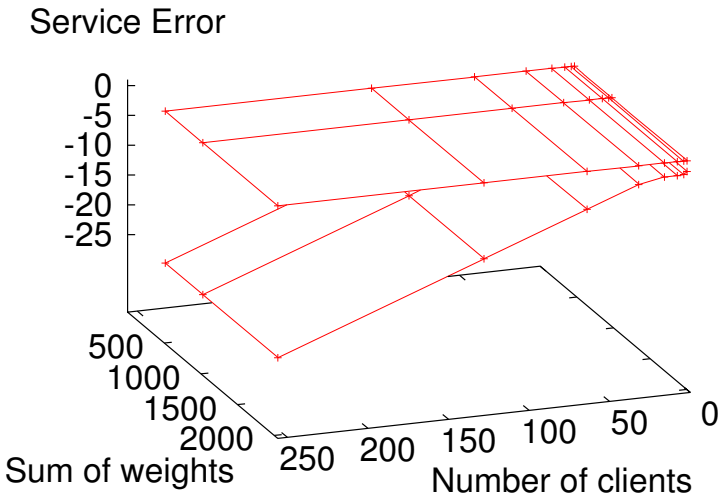

Figure 4.10: SFQ service error with number of clients=[2:256], total weights=[256:2048], and 10% skewed weight distribution

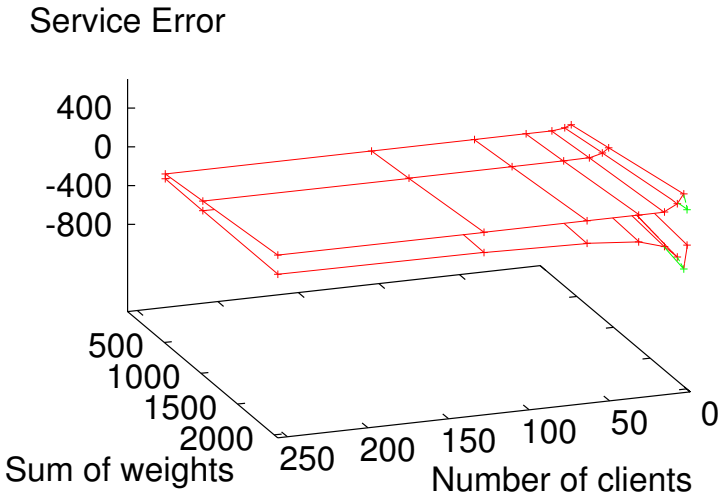

Figure 4.11: WRR service error with number of clients=[2:256], total weights=[256:2048], and 10% skewed weight distribution

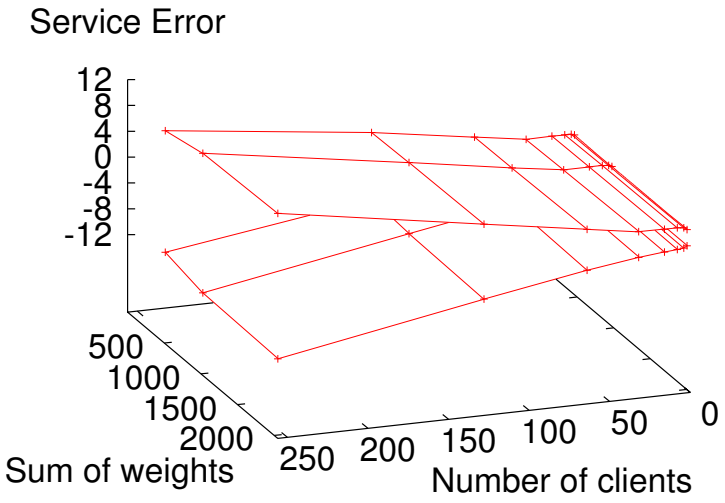

Figure 4.12: SRR service error with number of clients=[2:256], total weights=[256:2048], and 10% skewed weight distribution

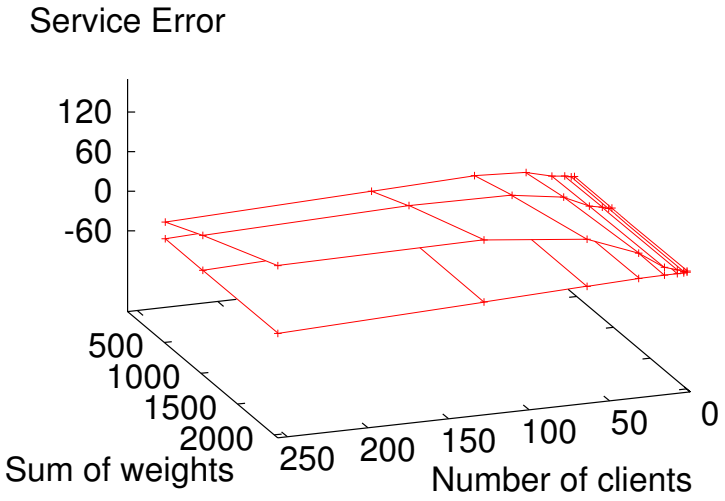

Figure 4.13: VTRR service error with number of clients=[2:256], total weights=[256:2048], and 10% skewed weight distribution

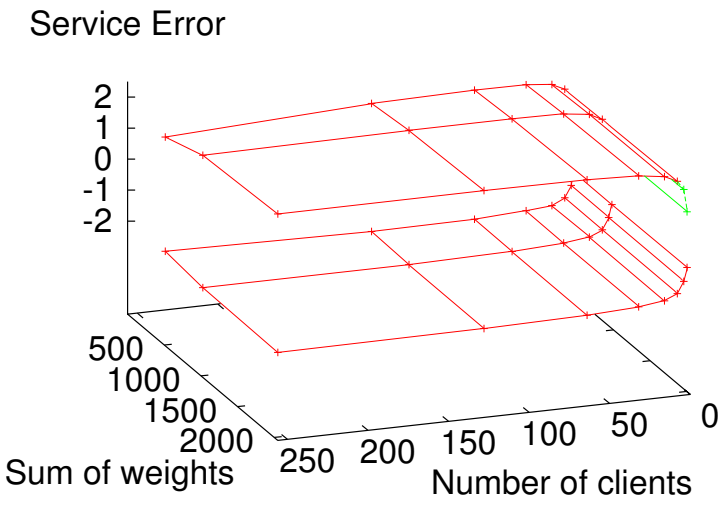

Figure 4.14: GR^3 service error with number of clients=[2:256], total weights=[256:2048], and 10% skewed weight distribution

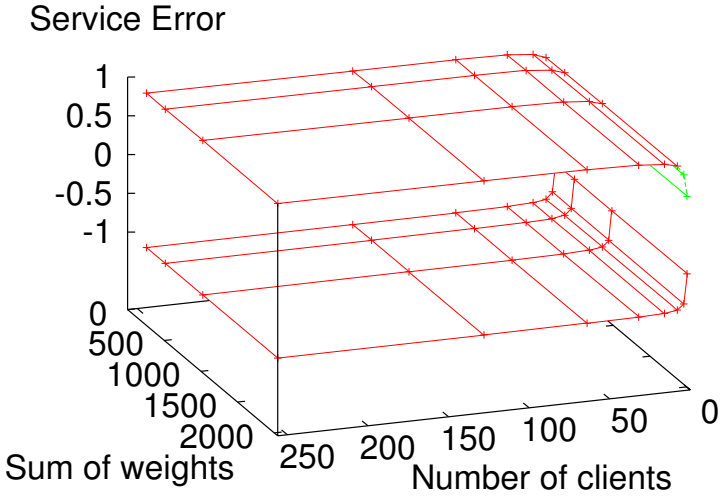

Figure 4.15: WF^2Q service error with number of clients=[2:256], total weights=[256:2048], and 50% skewed weight distribution

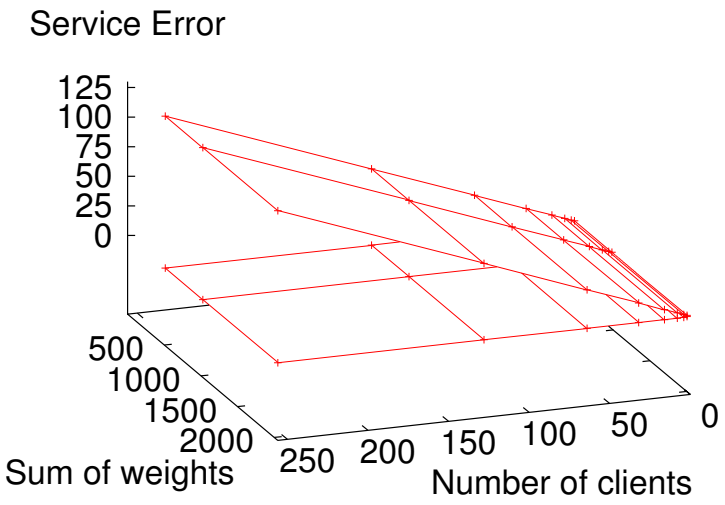

Figure 4.16: WFQ service error with number of clients=[2:256], total weights=[256:2048], and 50% skewed weight distribution

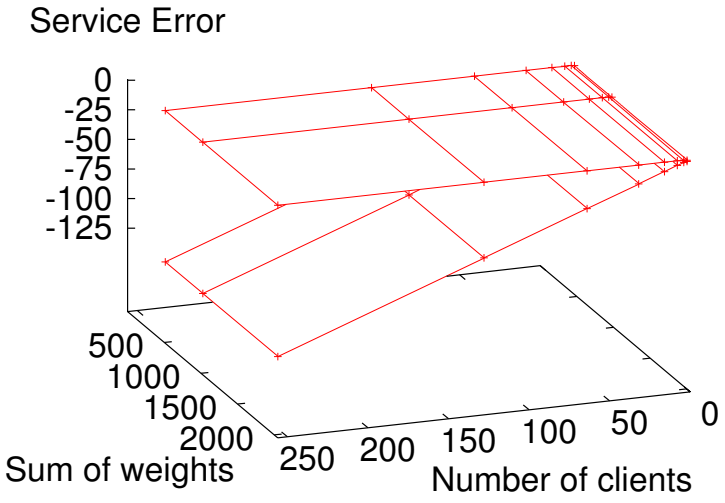

Figure 4.17: SFQ service error with number of clients=[2:256], total weights=[256:2048], and 50% skewed weight distribution

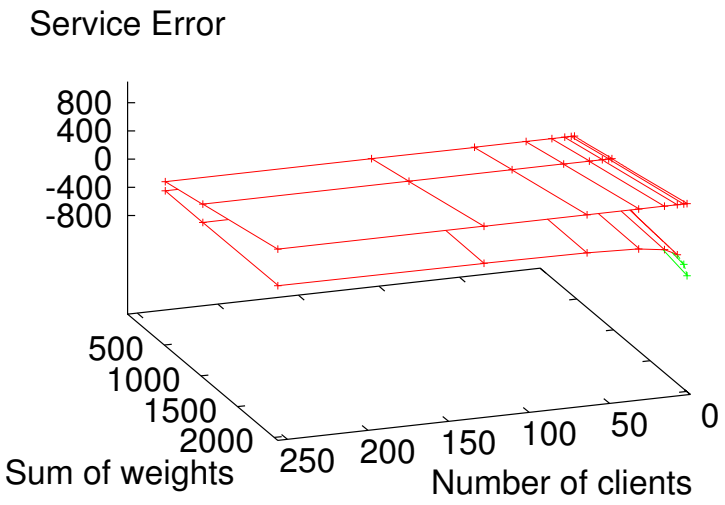

Figure 4.18: WRR service error with number of clients=[2:256], total weights=[256:2048], and 50% skewed weight distribution

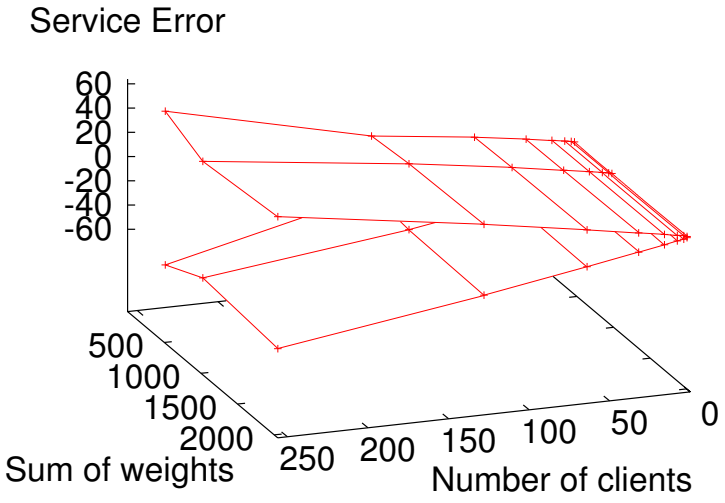

Figure 4.19: SRR service error with number of clients=[2:256], total weights=[256:2048], and 50% skewed weight distribution

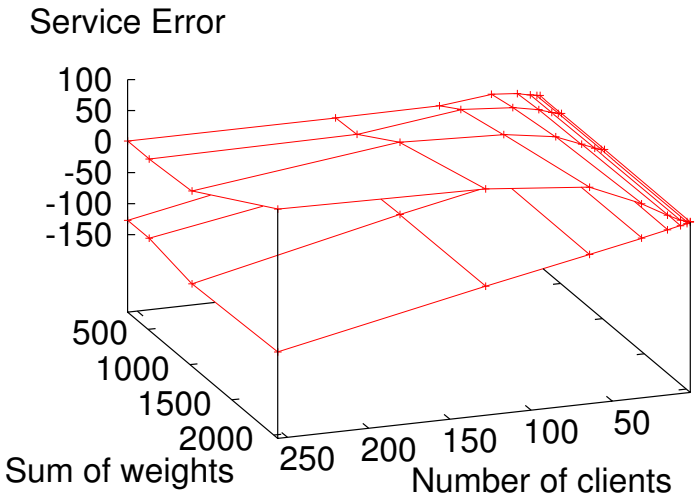

Figure 4.20: VTRR service error with number of clients=[2:256], total weights=[256:2048], and 50% skewed weight distribution

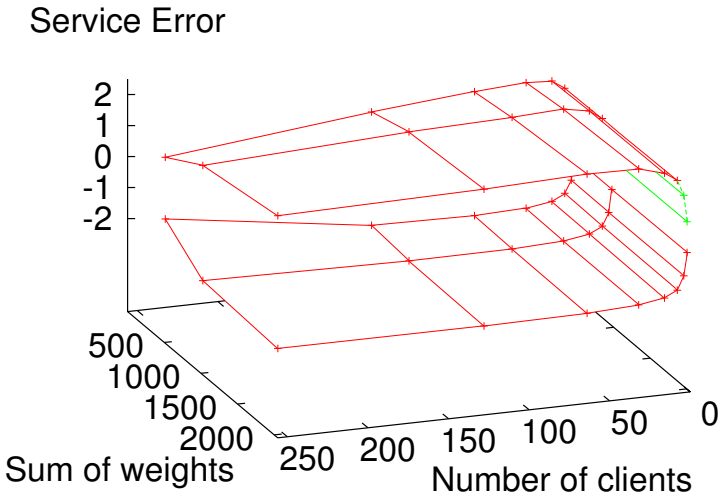

Figure 4.21: GR^3 service error with number of clients=[2:256], total weights=[256:2048], and 50% skewed weight distribution

Weight Distribution	Algorithm	Minimum	Maximum
Random	WF^2Q	-1	1
Random	WFQ	-1	2.2
Random	SFQ	-2.2	1
Random	WRR	-12845	12833
Random	SRR	-2.6	2.6
Random	$VTRR$	-27.4	143.3
Random	GR^3	-2.3	2.6
10% Skew	WF^2Q	-1	1
10% Skew	WFQ	-1	818
10% Skew	SFQ	-818	1
10% Skew	WRR	-12069	23593
10% Skew	SRR	-369	369
10% Skew	$VTRR$	-2145	10080
10% Skew	GR^3	-2.3	2.6
50% Skew	WF^2Q	-1	1
50% Skew	WFQ	-1	4095
50% Skew	SFQ	-4095	1
50% Skew	WRR	-7139	65536
50% Skew	SRR	-1968	1968
50% Skew	$VTRR$	-4593	12681
50% Skew	GR^3	-2.3	4.6

Table 4.2: Average service error range with number of clients=[32:8192] and total weights=[16384:262144]

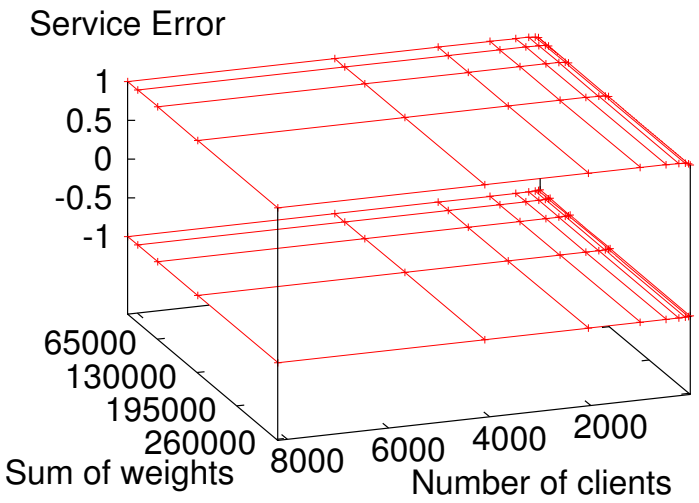

Figure 4.22: WF^2Q service error with number of clients=[32:8192], total weights=[16384:262144], and random weight distribution

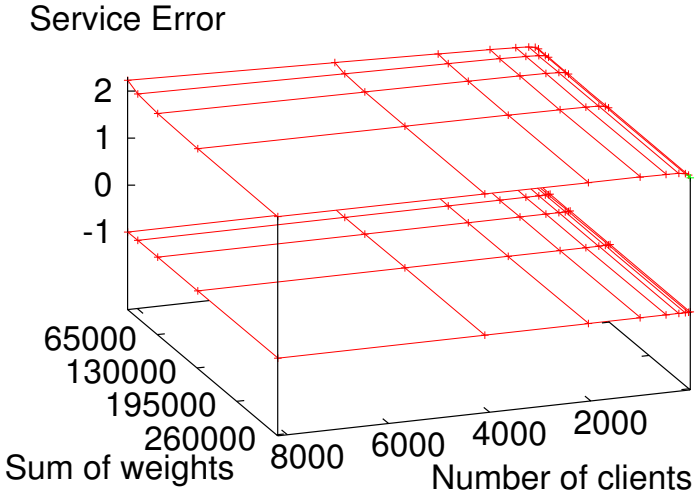

Figure 4.23: WFQ service error with number of clients=[32:8192], total weights=[16384:262144], and random weight distribution

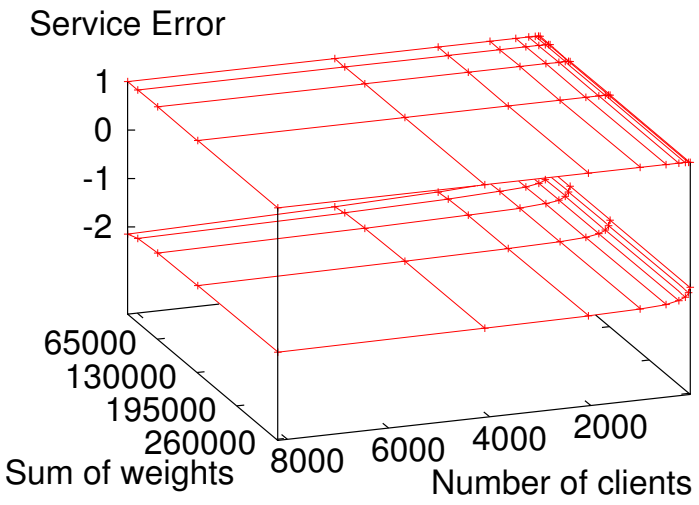

Figure 4.24: SFQ service error with number of clients=[32:8192], total weights=[16384:262144], and random weight distribution

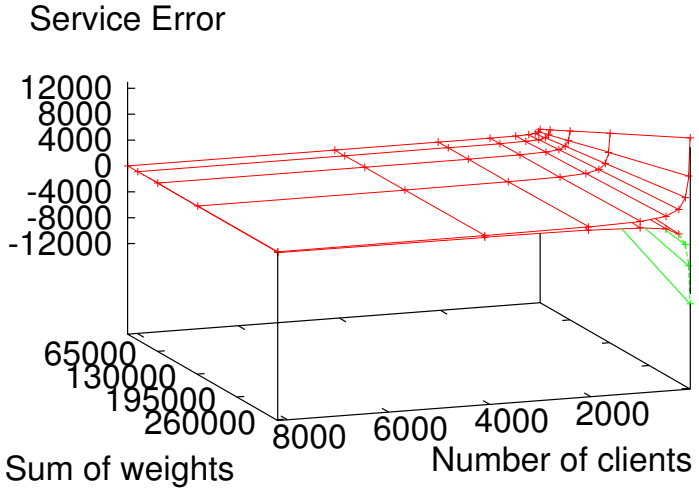

Figure 4.25: WRR service error with number of clients=[32:8192], total weights=[16384:262144], and random weight distribution

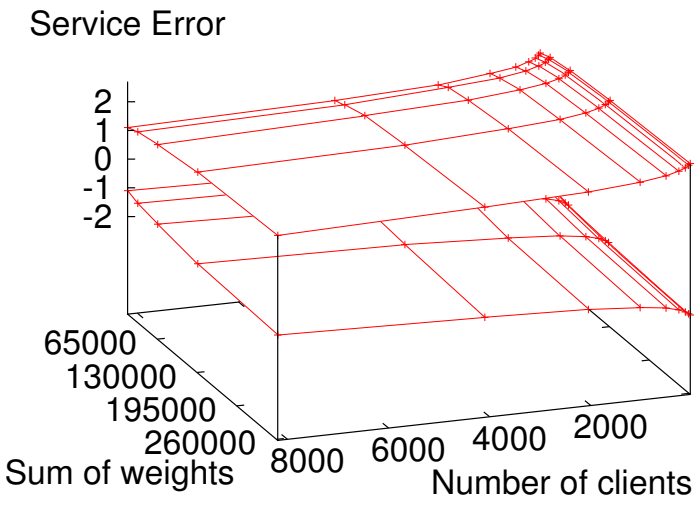

Figure 4.26: SRR service error with number of clients=[32:8192], total weights=[16384:262144], and random weight distribution

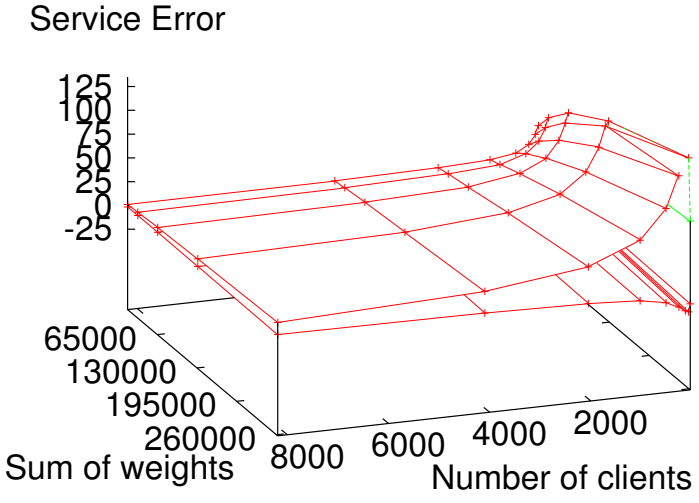

Figure 4.27: VTRR service error with number of clients=[32:8192], total weights=[16384:262144], and random weight distribution

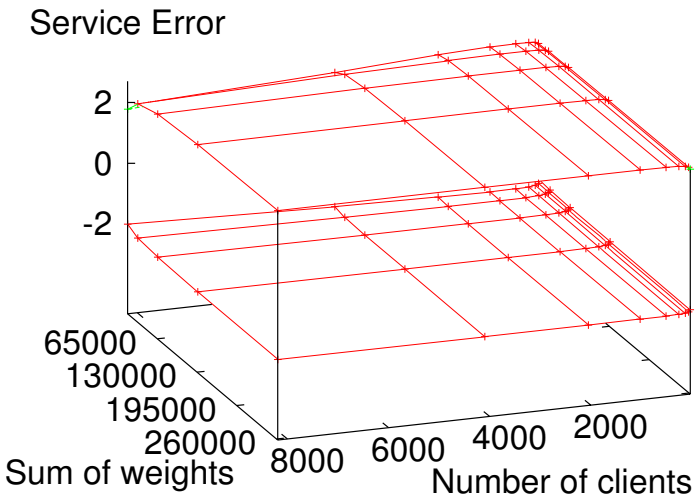

Figure 4.28: GR^3 service error with number of clients=[32:8192], total weights=[16384:262144], and random weight distribution

time error is between -1 tu and 4095 tu, which is much less than WRR. Figure 4.38 shows SFQ's service time error is between -4095 tu and 1 tu, which is almost a mirror image of WFQ. Figure 4.40 shows SRR's service error is between -1968 tu and 1968 tu. Figure 4.41 shows VTRR's service error is between -4593 tu and 12681 tu.

In comparison, Figure 4.42 shows the service time error ranges for GR^3. GR^3's service time error only ranges from -2.3 to 2.6 tu. GR^3 has a smaller error range than all of the other schedulers measured. 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 mathematically 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. WF^2Q has the smallest service errors since it is mathematically bounded between -1 and 1. Considering the total service error range of each scheduler, GR^3 provides well over two orders of magnitude better proportional

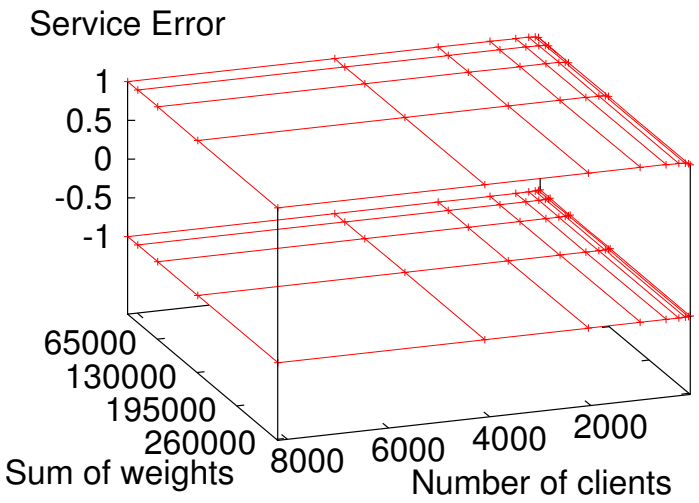

Figure 4.29: WF^{2}Q service error with number of clients=[32:8192], total weights=[16384:262144], and 10% skewed weight weight distribution

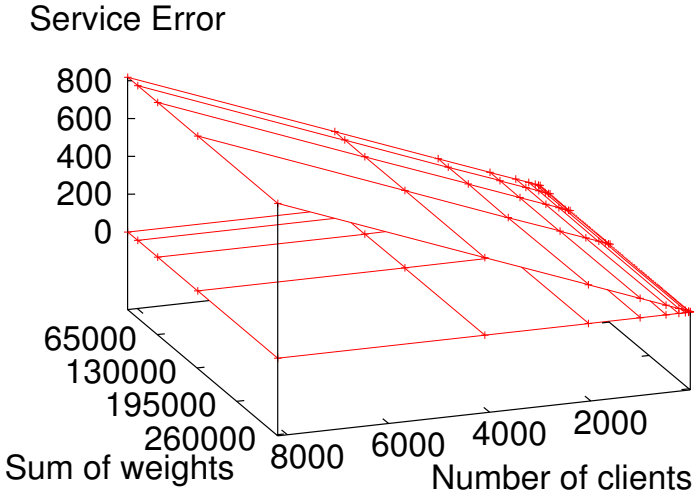

Figure 4.30: WFQ service error with number of clients=[32:8192], total weights=[16384:262144], and 10% skewed weight weight distribution

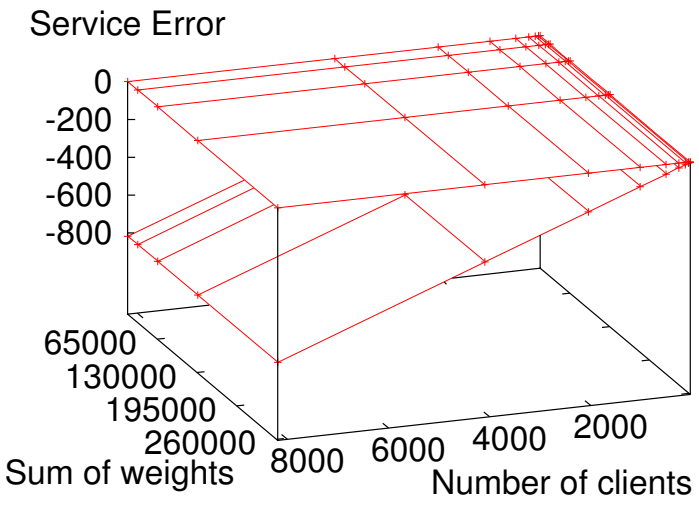

Figure 4.31: SFQ service error with number of clients=[32:8192], total weights=[16384:262144], and 10% skewed weight weight distribution

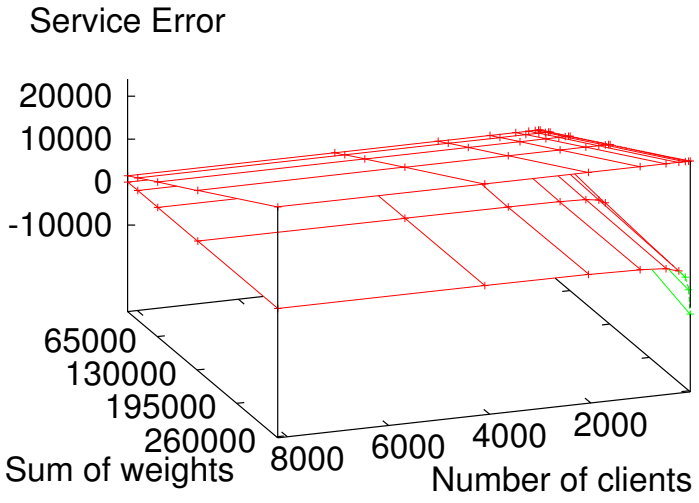

Figure 4.32: WRR service error with number of clients=[32:8192], total weights=[16384:262144], and 10% skewed weight weight distribution

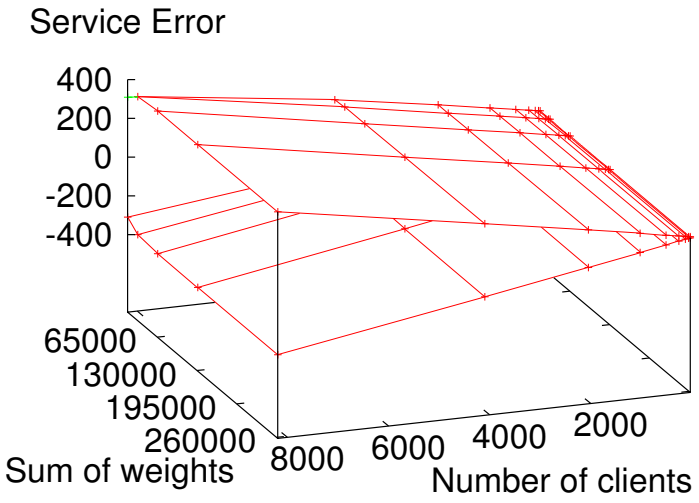

Figure 4.33: SRR service error with number of clients=[32:8192], total weights=[16384:262144], and 10% skewed weight weight distribution

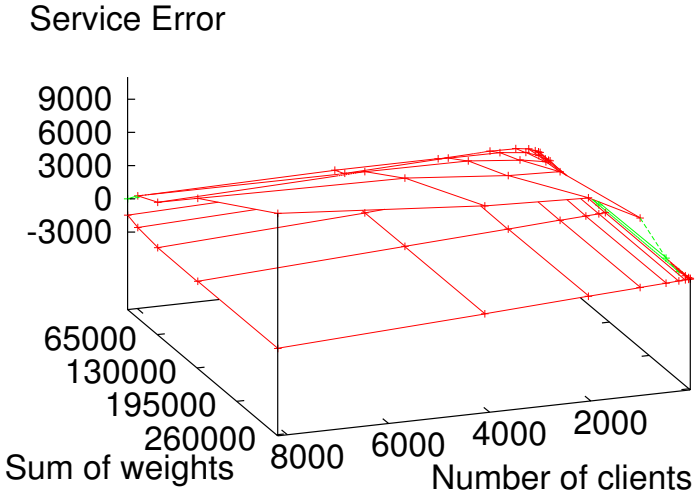

Figure 4.34: VTRR service error with number of clients=[32:8192], total weights=[16384:262144], and 10% skewed weight weight distribution

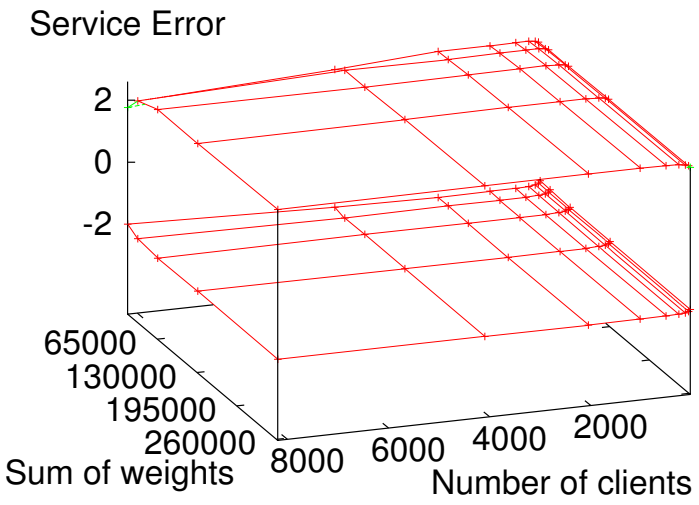

Figure 4.35: GR^3 service error with number of clients=[32:8192], total weights=[16384:262144], and 10% skewed weight weight distribution

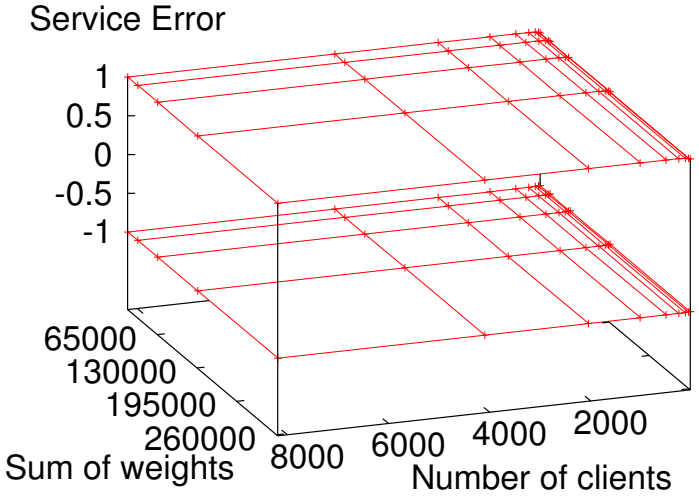

Figure 4.36: WF^2Q service error with number of clients=[32:8192], total weights=[16384:262144], and 50% skewed weight weight distribution

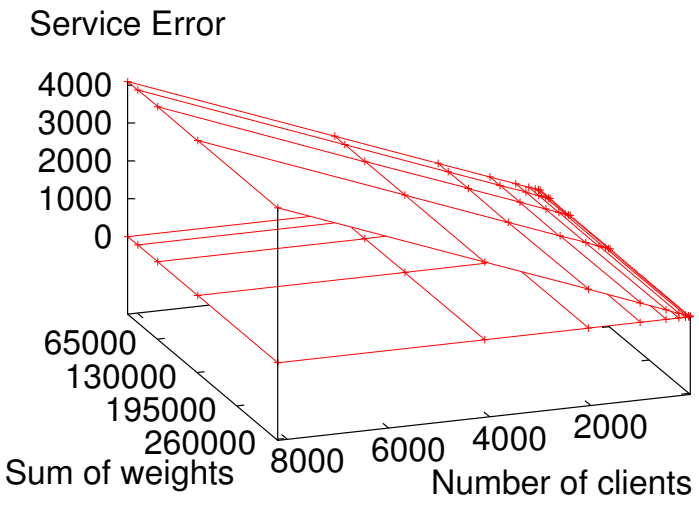

Figure 4.37: WFQ service error with number of clients=[32:8192], total weights=[16384:262144], and 50% skewed weight weight distribution

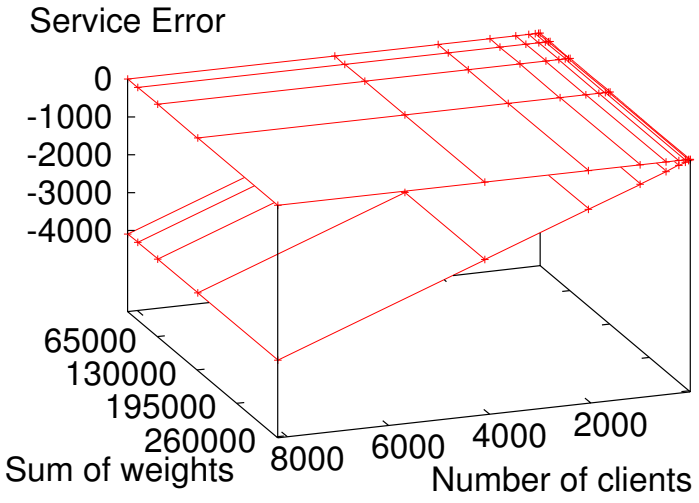

Figure 4.38: SFQ service error with number of clients=[32:8192], total weights=[16384:262144], and 50% skewed weight weight distribution

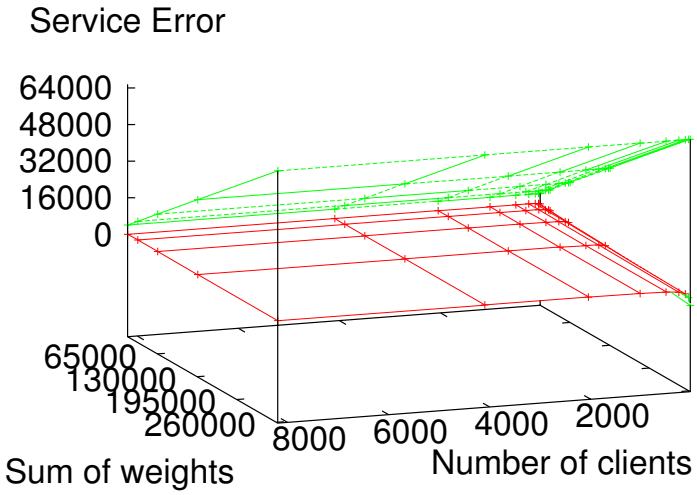

Figure 4.39: WRR service error with number of clients=[32:8192], total weights=[16384:262144], and 50% skewed weight weight distribution

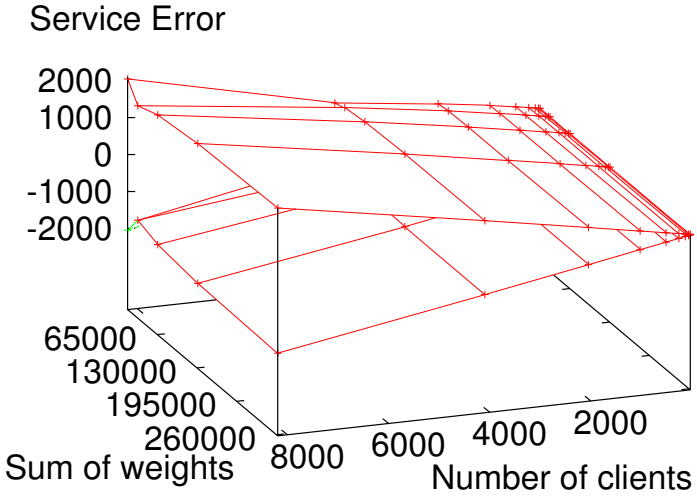

Figure 4.40: SRR service error with number of clients=[32:8192], total weights=[16384:262144], and 50% skewed weight weight distribution

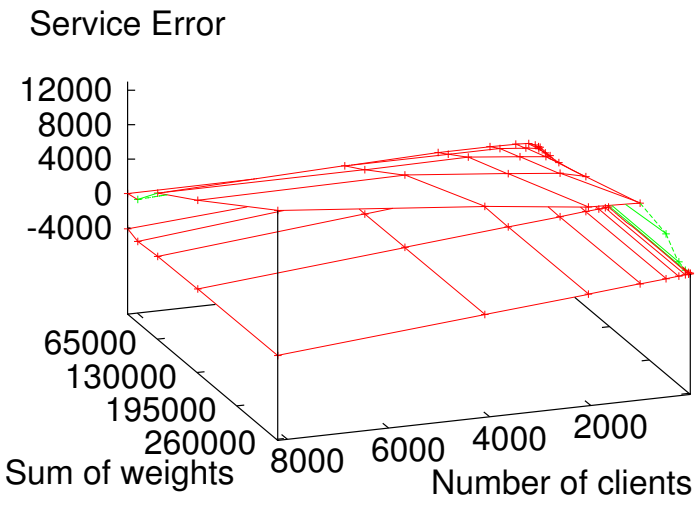

Figure 4.41: VTRR service error with number of clients=[32:8192], total weights=[16384:262144], and 50% skewed weight weight distribution

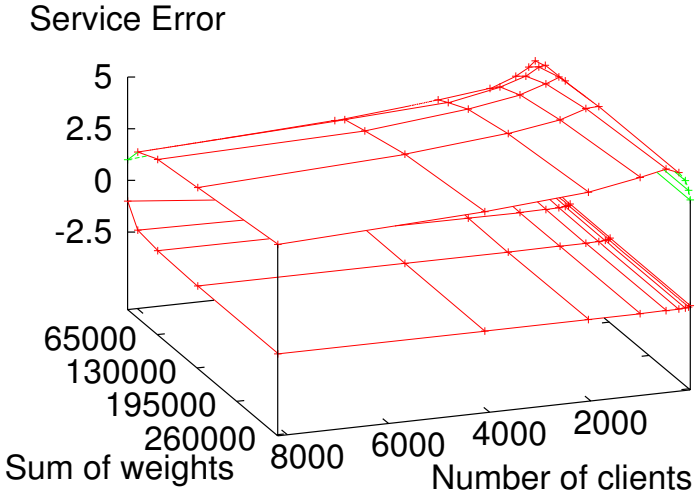

Figure 4.42: GR^3 service error with number of clients=[32:8192], total weights=[16384:262144], and 50% skewed weight weight distribution

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.

It is worth noting that as the weight skew becomes more accentuated, the error measured grows dramatically. Thus, increasing the skew from 10 to 50 percent results in more than a fivefold increase in the average 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.

4.2 Linux Kernel Measurements

4.2.1 Scheduling Overhead

To evaluate the scheduling overhead of GR^3, we implemented GR^3 in the Linux operating system and compared the overhead of our prototype GR^3 implementation against the overhead of the standard Linux 2.4 scheduler, a WFQ scheduler, and a VTRR scheduler. We conducted a series of experiments to quantify how the scheduling overhead for each scheduler varies as the number of clients increases. For the first experiment, each client 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. 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 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 4.43 shows the average execution time required by each scheduler to select a client to execute. 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 separate fixed-length array. If the number of clients ever exceeds the length of the array, a costly array reallocation must be performed. We chose an initial array size large enough to contain more than 400 clients, so this additional cost is not reflected in our measurements.

As shown in Figure 4.43, 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. Because of the importance of constant scheduling overhead in server systems, Linux has switched to Ingo Molnar's $O(1)$ scheduler in the Linux 2.5 development kernel. As a comparison, we also repeated this microbenchmark experiment with that scheduler and found that GR^3 still runs over 30 percent faster.

For the second experiment, we measured the scheduling overhead of the various schedulers for *hackbench* [19], a benchmark used in the Linux community 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 standard Linux scheduler, WFQ, VTRR, and GR^3 for 1 group up to 100 groups. Because hackbench frequently inserts and removes clients from the run

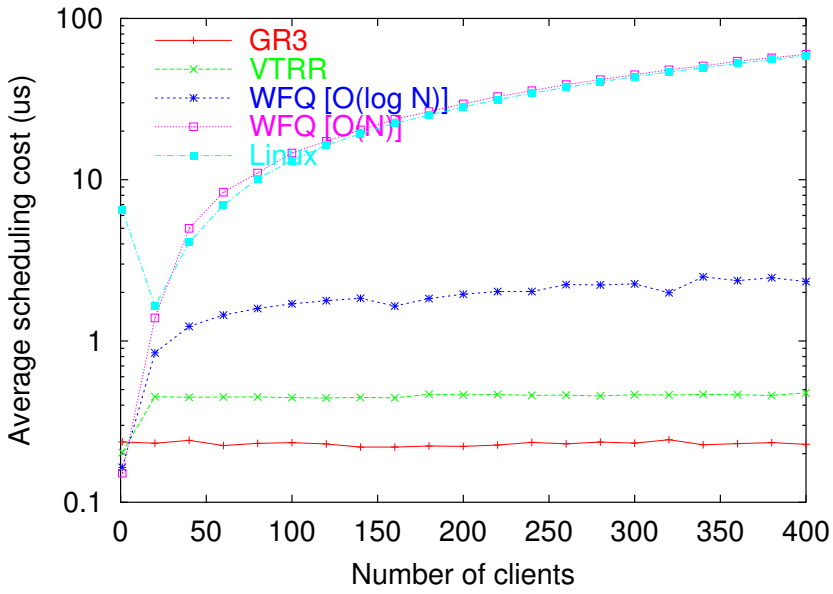

Figure 4.43: Average scheduling overhead

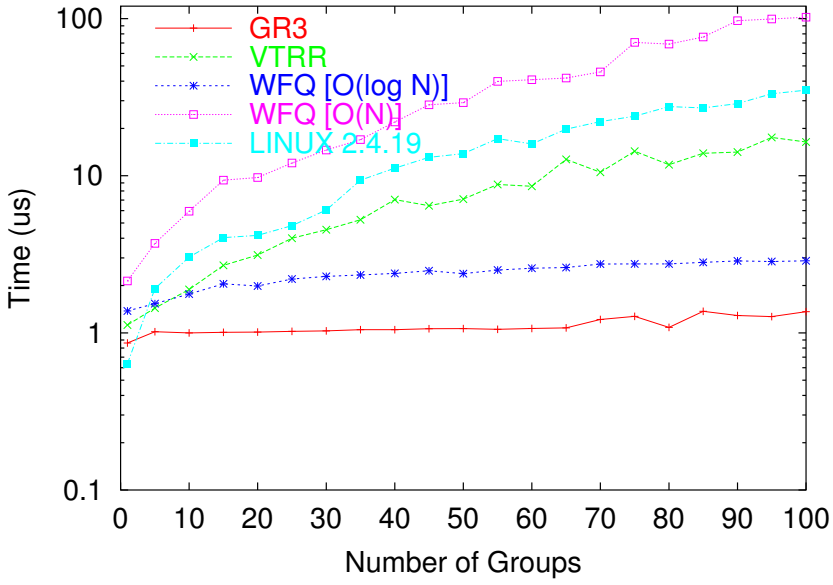

Figure 4.44: Hackbench weighted scheduling overhead

queue, the cost of client insertion and removal is a more significant factor for this benchmark.

Figure 4.44 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 4.44 is higher than the average cost per schedule in Figure 4.43 for all the schedulers measured since Figure 4.44 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. Note that while selecting a client to run using VTRR is also independent of the number of clients, insertion overhead in VTRR grows with the number of clients, resulting in much higher VTRR overhead for this benchmark.

4.2.2 Application Workloads

We demonstrate GR^3's efficient proportional sharing of resources with two experiments. First, we ran tests with an MPEG audio encoder and contrast the performance of GR^3 versus other schedulers. In the second experiment, we show that GR^3's proportional sharing holds for real applications with a larger workload.

MPEG Encoder

We briefly describe two simple tests running an MPEG audio encoder. We contrast the performance of GR^3 versus the standard 2.4 Linux scheduler, WFQ, and VTRR. We ran multiple MPEG audio encoders with different weights on each of the four schedulers. We ran two tests. The first encoder test ran five copies of the encoder with respective weights 1, 2, 3, 4, and 5. The second encoder test ran six copies of

the encoder with one encoder client assigned a weight of 10 and the other encoder clients each assigned a weight of 1. For the Linux scheduler, weights were assigned by selecting `nice` values appropriately. The encoders were instrumented with time stamp event recorders in a manner similar to how we recorded time in our micro-benchmark programs. Each encoder took its input from the same file, but wrote output to its own file. MPEG audio is encoded in chunks called frames, so our instrumented encoder records a timestamp after each frame is encoded, allowing us to easily observe the effect of resource weight on single-frame encoding time.

GR^3 provided the best overall proportional fairness for these experiments while Linux provided the worst overall proportional fairness. Figures 4.45 to 4.48 and Figures 4.49 to 4.52 show the number of frames encoded over time for the Linux scheduler, WFQ, VTRR, and GR^3 for the first and second encoder tests, respectively. For the encoders with weights 1, 2, 3, 4, and 5, only the Linux scheduler has a pronounced "staircase" effect, indicating that CPU resources are provided in irregular bursts when viewed over a short time interval. For an audio encoder, this can result in extra jitter, resulting in delays and dropouts. The Linux scheduler does not provide sharing as fairly as WFQ, VTRR, and GR^3 when viewed over a short time interval. All of the schedulers except GR^3 have the "staircase" effect for the encoder with weight 10. It can be inferred from the smoother curves of Figure 4.52 that GR^3 provides fair resource allocation at a finer granularity than the other schedulers.

Virtual Web Server

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: Linux 2.4 schedulers, WFQ, VTRR and GR^3. 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

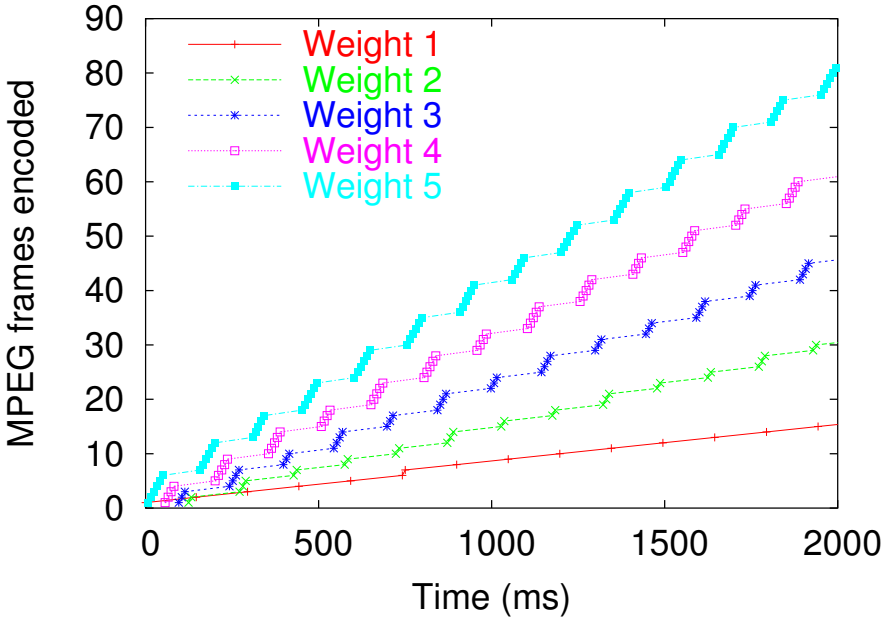

Figure 4.45: MPEG Encoding with Linux, for weights 1, 2, 3, 4, 5

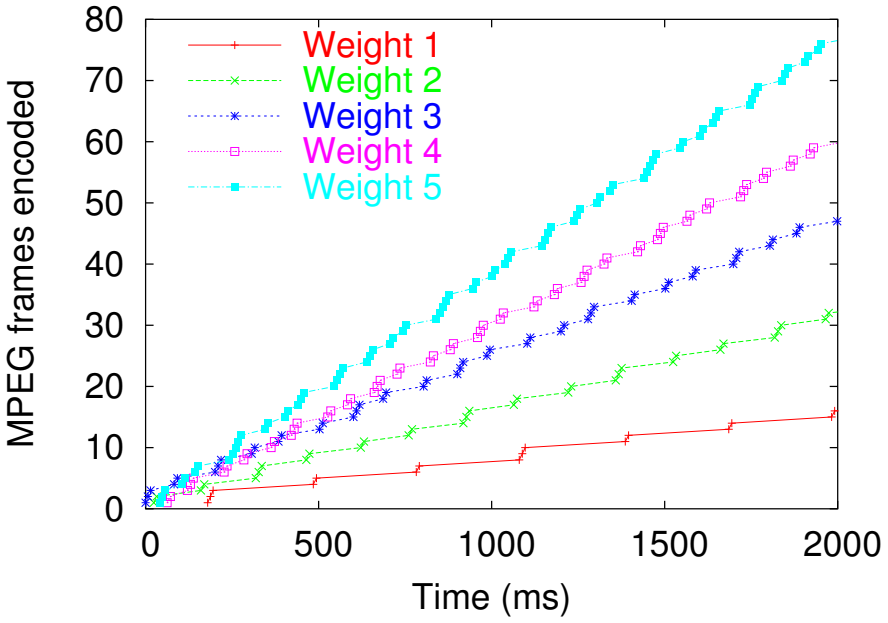

Figure 4.46: MPEG Encoding with WFQ, for weights 1, 2, 3, 4, 5

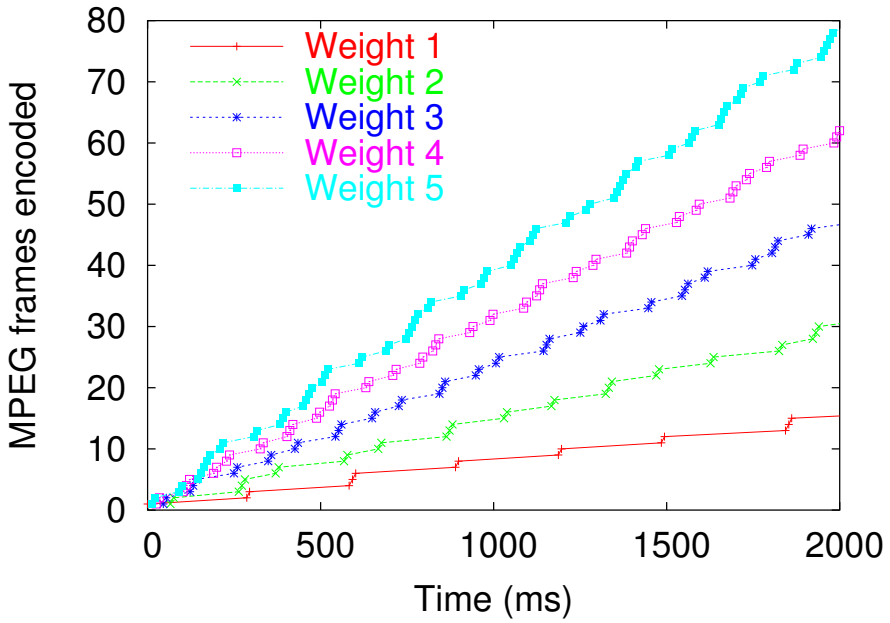

Figure 4.47: MPEG Encoding with VTRR, for weights 1, 2, 3, 4, 5

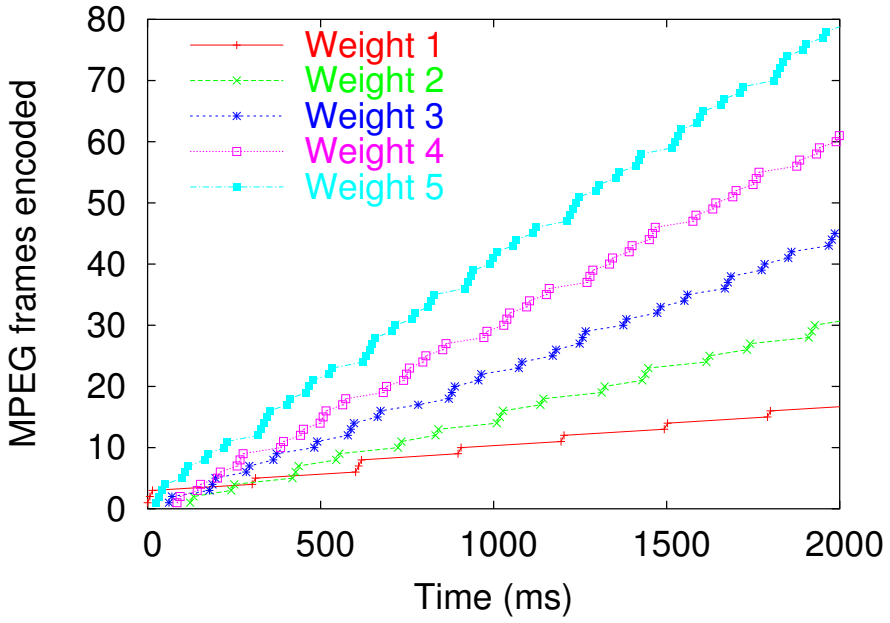

Figure 4.48: MPEG Encoding with GR^3, for weights 1, 2, 3, 4, 5

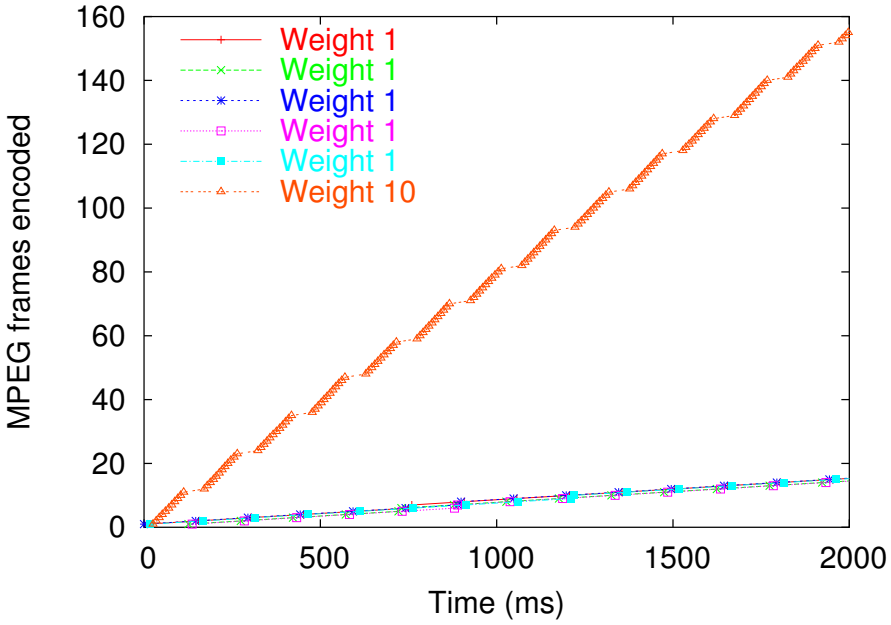

Figure 4.49: MPEG Encoding with Linux, for weights 1, 1, 1, 1, 1, 10

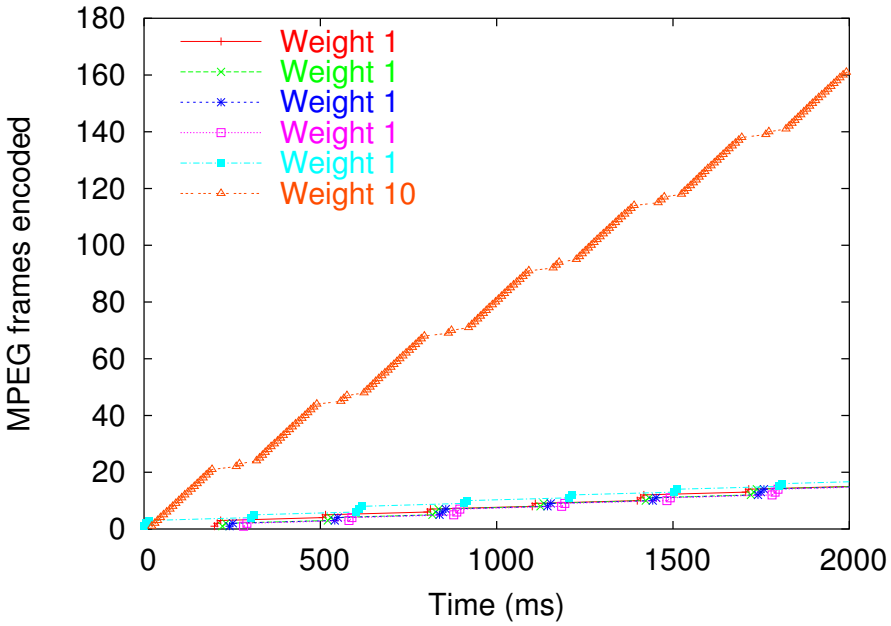

Figure 4.50: MPEG Encoding with WFQ, for weights 1, 1, 1, 1, 1, 10

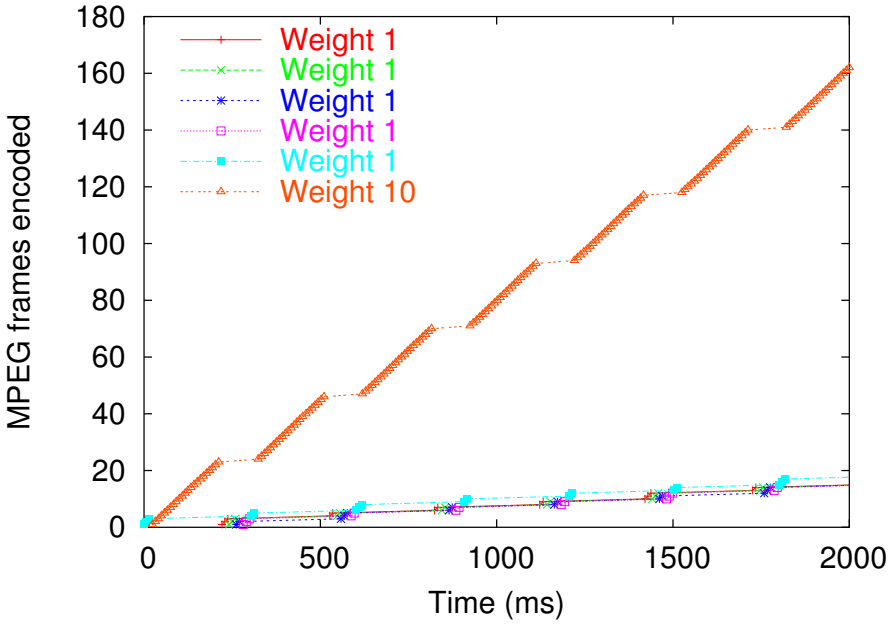

Figure 4.51: MPEG Encoding with VTRR, for weights 1, 1, 1, 1, 1, 10

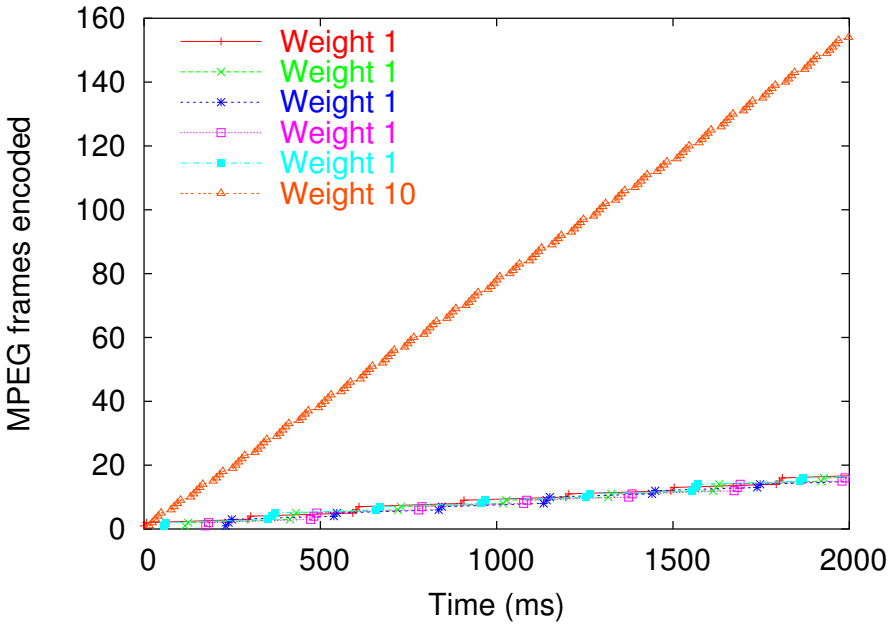

Figure 4.52: MPEG Encoding with GR^3, for weights 1, 1, 1, 1, 1, 10

simultaneously.

We ran multiple virtual servers with each one having different weights for its processes. In the first experiment, we ran five virtual servers which assigned completely random weights to processes between 1 and 10. 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 used six virtual servers, with one server having all its processes assigned weight 10 while all the other servers had processes assigned weight 1. For experiments one and two, the total load on the system consisted of 500 processes running simultaneously. For the third experiment, there are 600 processes. For the Linux scheduler, weights were assigned by selecting `nice` values appropriately. Figures 4.53 to 4.64 present the results for the three experiments. For illustration purposes, only one process from each server is shown in the figures. The figures showed the amount of process time allocated to each client over time for the Linux scheduler, WFQ, VTRR, and GR^3 for each experiment.

Linux provided the worst overall proportional fairness in all three experiments. WFQ, VTRR, and GR^3 showed good proportional fairness when the processes have a random weight distribution. As the weight distribution becomes skewed, the proportional fairness of WFQ and VTRR are worse than GR^3. In the third experiment, all of the schedulers except GR^3 have the "staircase" effect for the search engine process with weight 10. For the search engine which needs to provide interactive responsiveness to web users, the irregular bursts of CPU resources can result in extra delays in system response time. The smoother curves of Figure 4.64 indicated that GR^3 provides better overall proportional fairness than the other schedulers.

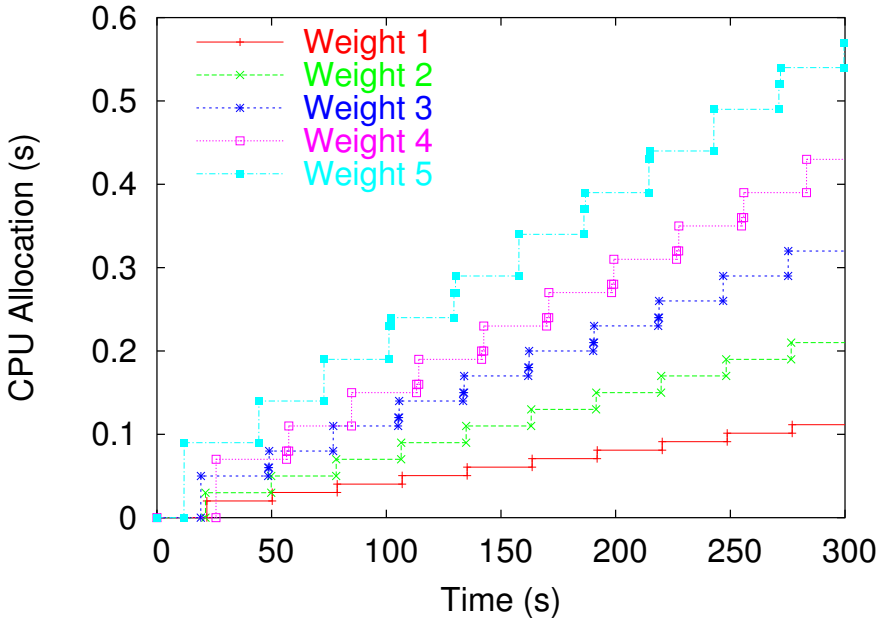

Figure 4.53: Linux CPU allocation for 5 virtual web servers with random weights

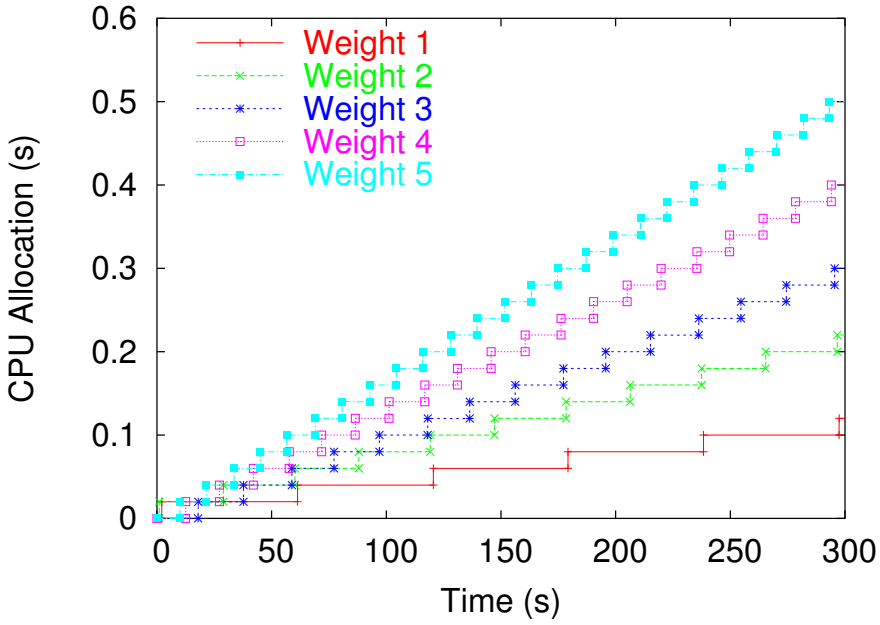

Figure 4.54: WFQ CPU allocation for 5 virtual web servers with random weights

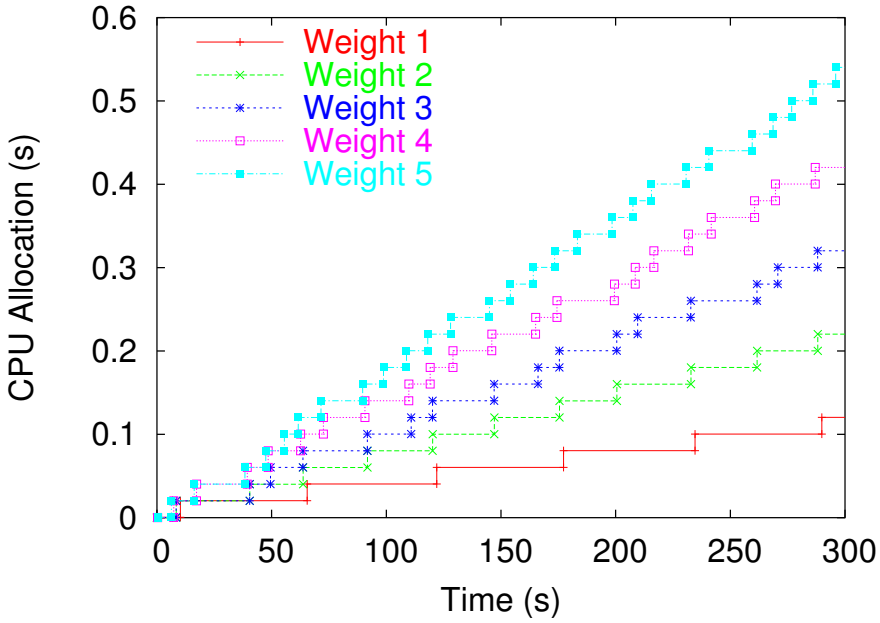

Figure 4.55: VTRR CPU allocation for 5 virtual web servers with random weights

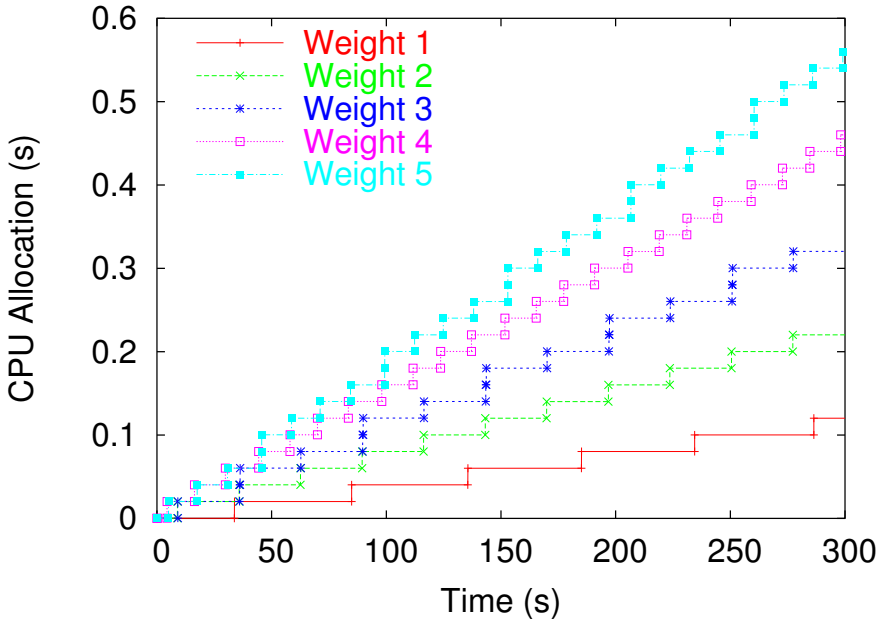

Figure 4.56: GR^3 CPU allocation for 5 virtual web servers with random weights

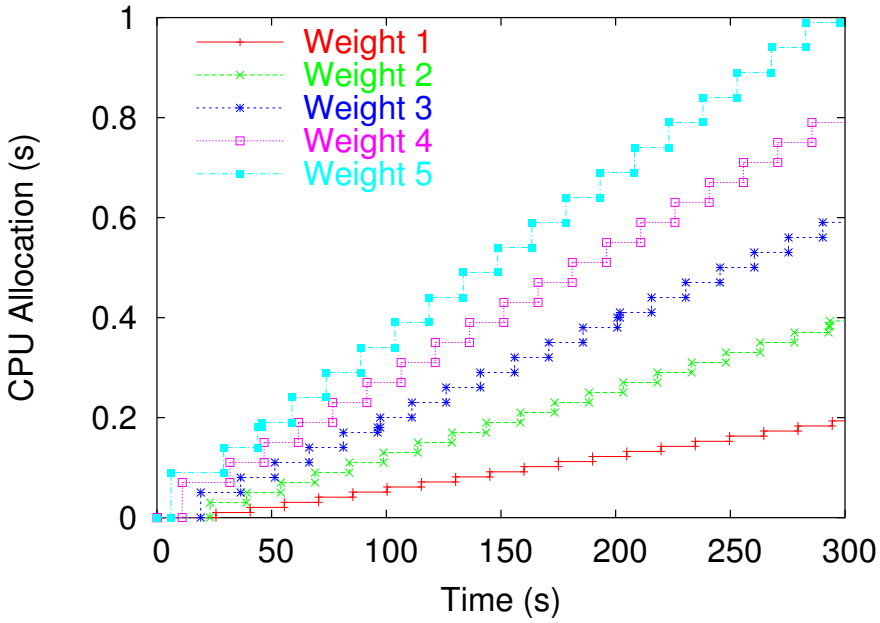

Figure 4.57: Linux CPU allocation for 5 virtual web servers with weights 1, 2, 3, 4, 5

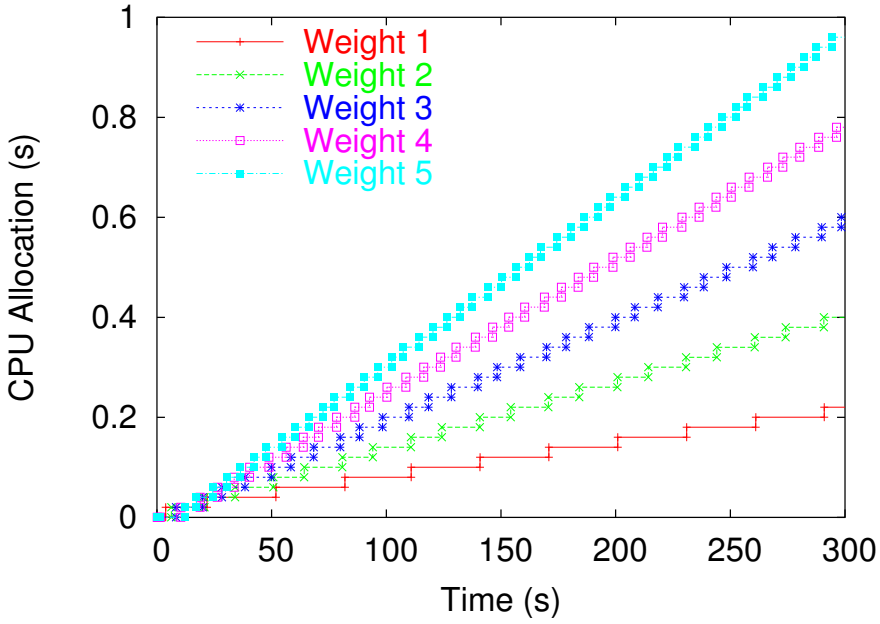

Figure 4.58: WFQ CPU allocation for 5 virtual web servers with weights 1, 2, 3, 4, 5

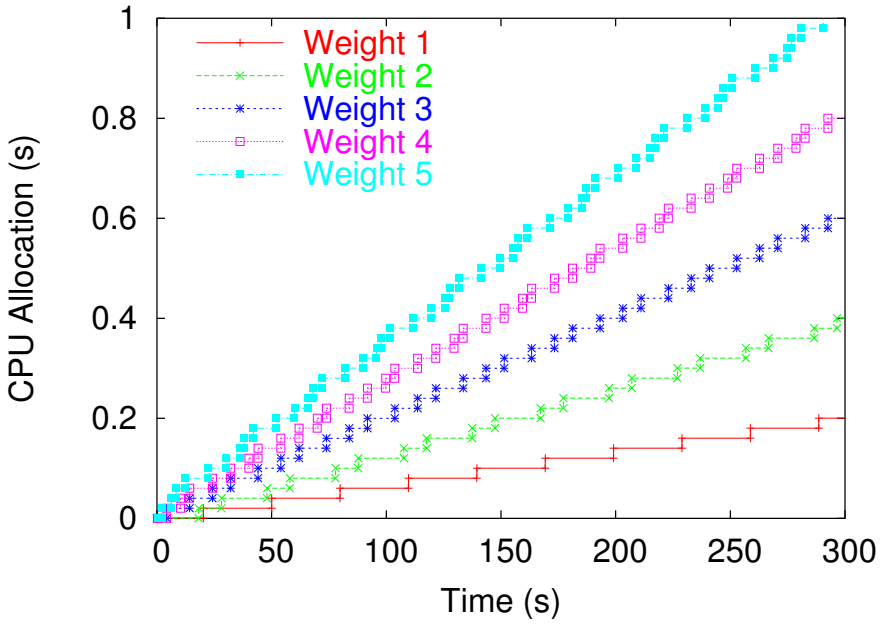

Figure 4.59: VTRR CPU allocation for 5 virtual web servers with weights 1, 2, 3, 4, 5

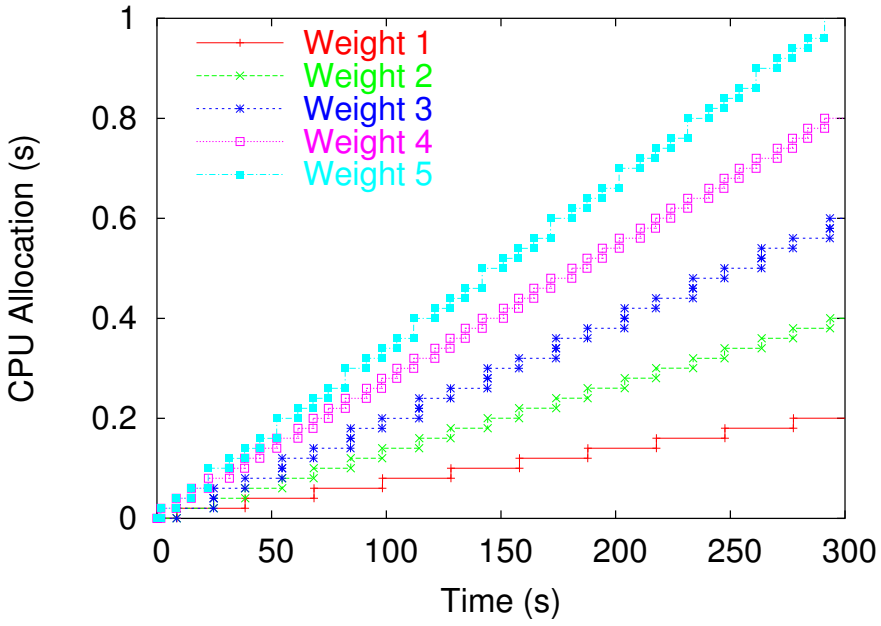

Figure 4.60: GR^3 CPU allocation for 5 virtual web servers with weights 1, 2, 3, 4, 5

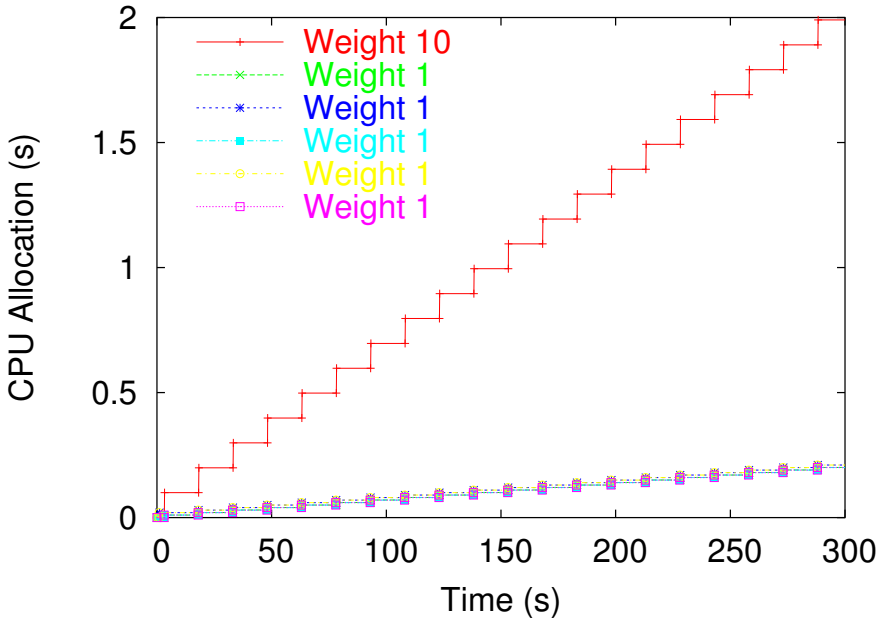

Figure 4.61: Linux CPU allocation for 6 virtual web servers with weights 10, 1, 1, 1, 1, 1

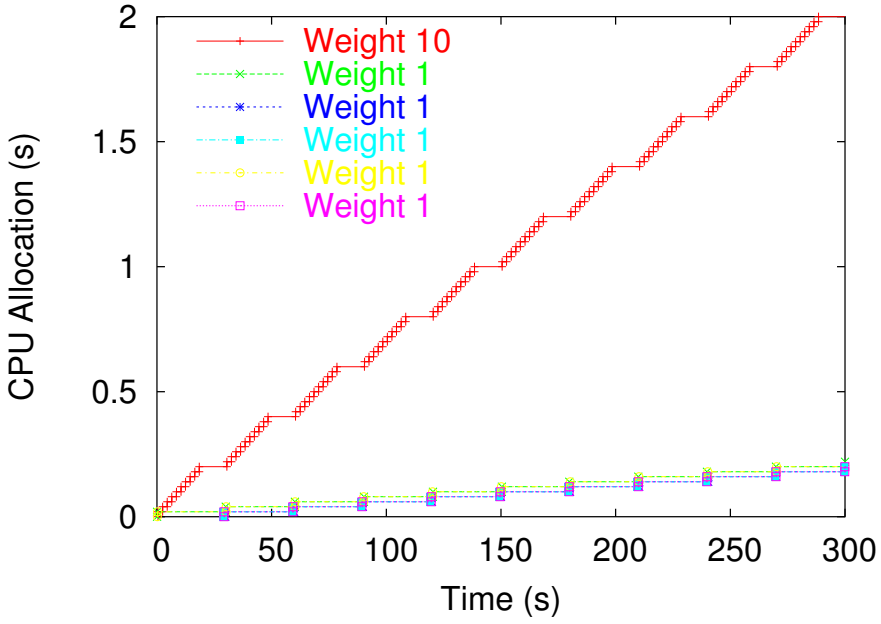

Figure 4.62: WFQ CPU allocation for 6 virtual web servers with weights 10, 1, 1, 1, 1, 1

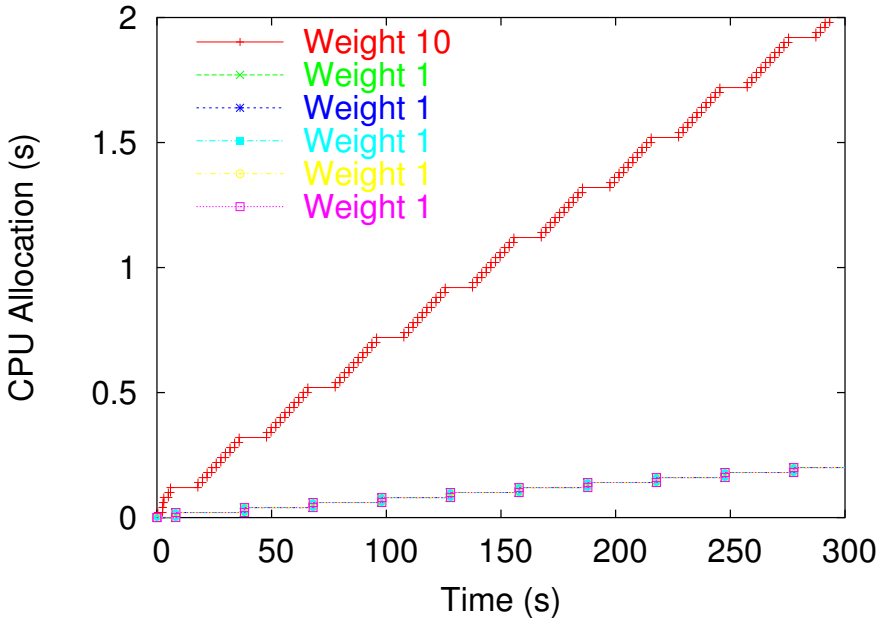

Figure 4.63: VTRR CPU allocation for 6 virtual web servers with weights 10, 1, 1, 1, 1, 1

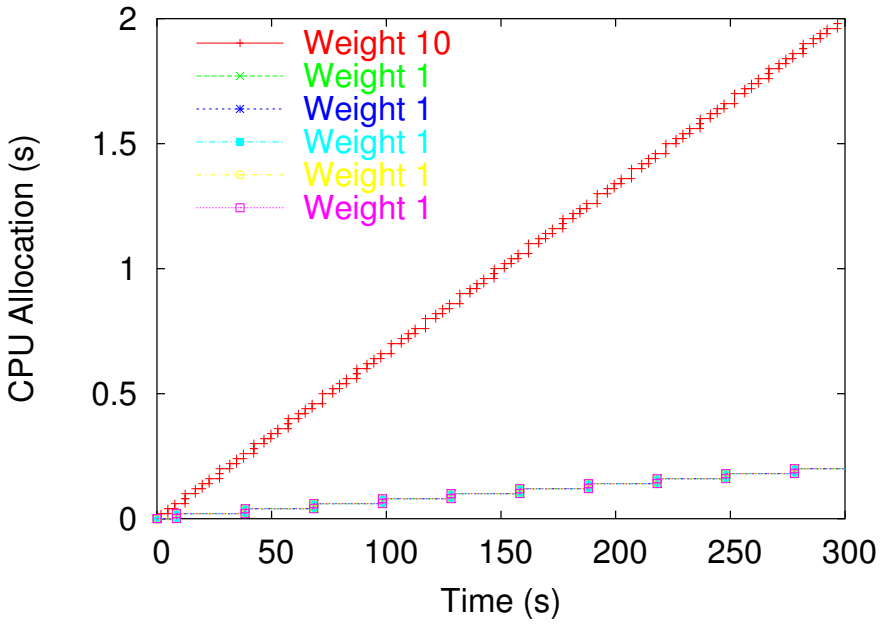

Figure 4.64: GR^3 CPU allocation for 6 virtual web servers with weights 10, 1, 1, 1, 1, 1

Chapter 5

Related Work

One of the oldest, simplest and most widely used proportional share scheduling algorithms is round-robin, which serves as the basis for a number of other 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) [20] 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 [7, 11, 12] arose in response to a need to 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 [7].

Lottery scheduling [23] gives each client a number of tickets proportional to its weight, then randomly selects a ticket and schedules the client that owns the selected ticket to run for a time quantum. Lottery scheduling requires at least $O(\log N)$ time

complexity, where N is the number of clients. Because lottery scheduling relies on the law of large numbers for providing proportional fairness, its allocation errors can be very large, typically much worse than WRR for clients with smaller weights.

Fair queueing was first proposed by Demers et. al. for network packet scheduling as Weighted Fair Queueing (WFQ) [6], with a more extensive analysis provided by Parekh and Gallager [16], and later applied by Waldspurger and Weihl to CPU scheduling as stride scheduling [23]. Other variants of WFQ such as Virtual-clock [24], SFQ [9], SPFQ [21], and Time-shift FQ [5] have also been proposed. These algorithms generally assign each client a virtual time and schedule the client with the earliest virtual time, where a client's virtual time advances at a rate inversely proportional to the client's weight when the client runs. Because this requires ordering the clients based on virtual time, scheduling requires at least $O(\log N)$ time complexity. It has been shown that WFQ guarantees that the service time error for any client never falls below -1, which means that a client can never fall behind its ideal allocation by more than a single time quantum [16]. 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 in the presence of skewed weight distributions.

Several fair queueing approaches have been proposed for reducing this $O(N)$ service time error. A hierarchical scheduling approach can be used by grouping clients in a balanced binary tree of groups and recursively applying the basic fair queueing algorithm, thereby reducing service time error to $O(\log N)$, where N is the number of clients [23]. Worst-Case Weighted Fair Queueing [1] introduced eligible virtual times and can guarantee both a lower and upper bound on error of -1 and $+1$, respectively, which means that a client can never fall behind or get ahead of its ideal allocation by more than a single time quantum. 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 complexity, where N is the number of clients.

Because of the need for faster scheduling algorithms with good fairness guarantees [4, 17], novel round-robin scheduling variants such as Virtual-Time Round-Robin (VTRR) [15] and Smoothed Round Robin (SRR) [4] have been developed that combine the benefits of constant-time scheduling overhead of round-robin with scheduling

accuracy that approximates fair queueing. These mechanisms provide proportional sharing by going round-robin through clients in special ways that run clients at different frequencies without having to reorder clients on each schedule. Unlike WRR, they can provide lower service time errors because they do not need to adjust the size of their time quanta to achieve proportional sharing. VTRR combines round-robin scheduling with a virtual time mechanism while SRR introduces a Weight Matrix and Weight Spread Sequence (WSS). Both 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. VTRR requires at least $O(\log N)$ time to insert into the run queue, where N is the number of clients. SRR requires at least $O(k)$ time to insert into the run queue, where $k = \log \phi_{max}$ and ϕ_{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 $O(kN)$.

More recently, Stratified Round Robin [17] was proposed as a low complexity solution for network packet scheduling, and possibly CPU scheduling. The algorithm uses a similar grouping strategy as GR^3, distributing all clients with weight between 2^{-k} and $2^{-(k-1)}$ into class F_k. Stratified RR splits time into scheduling slots and then makes sure to assign all the clients in class F_k one slot every scheduling interval (which is defined to consist of 2^k consecutive slots for class F_k). To account for the variation of up to a factor of two of the weights of the clients within a class, the algorithm has an intra-class aspect to it, assigning a credit of $2^k\phi_C$ to client $C \in F_k$ for each slot it is assigned, with a deficit keeping track of unused service. 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.

Stratified RR has weaker fairness guarantees and higher scheduling complexity than GR^3. Stratified RR 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. We can see this by considering $N + 1$ clients, where $N = 2^k$, C_1 having weight 0.5 and the rest with weights $2^{-(k+2)}$.

Therefore, $C_1 \in F_1$ and $C_i \in F_{k+2}$ for $1 < i \leq N+1$. The scheduling intervals are 2 slots long for F_1 and 2^{k+2} for F_{k+2}. Stratified RR will then alternate running C_1 and C_{i+1} for the first 2^{k+1} slots. At this point, the error of C_1, E_{C_1}, is $W_{C_1} - W_T \frac{\phi_1}{\Phi_T} = 2^k - 2^{k+1} \frac{2^{-1}}{2^{-1}+2^k * 2^{-(k+2)}} = 2^k - 2^{k+1} \frac{2}{3} = -\frac{2^k}{3} = -\frac{N}{3}$. Half of the 2^{k+1} slots that follow, before the next scheduling interval of F_{k+2} begins, will belong to C_1, while the other half will be unused (skipped over). Thus, effectively, C_1 runs uninterrupted for $2^k = N$ tu, recovering its negative error, but also resulting in severe burstiness and $O(N)$ service error. Client weights could be scaled to reduce this error, but with additional $O(N)$ complexity. Stratified RR requires $O(g)$ worst-case time to determine the next class that should be selected, where g is the number of groups. Although hardware support can hide this complexity assuming a reasonably small, predefined maximum number of groups [17], running Stratified RR as a CPU scheduler in software still requires $O(g)$ complexity.

GR^3 also differs from Stratified RR and other deficit round-robin variants in its distribution of deficit. In algorithms such as DRR, SRR, Stratified RR, 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 itself as employed in round-robin schemes doesn't allow for better than $O(N)$ error. In contrast, GR^3 ensures sure 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 rigurous analysis on network packet scheduling [14] suggests that $O(N)$ delay bounds are unavoidable with packet scheduling algorithms of less than $O(logN)$ time complexity. We make the observation that 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 at the same time preserving $O(g^2)$ service error.

Chapter 6

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 $O(1)$ 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. Furthermore, 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 at least O(log N) $O(log(N))$ 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. Our experiences with GR^3 show that it is simple to implement and easy to integrate into existing commercial operating systems. We have measured the performance of GR^3 using both extensive 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 WF^2Q, WFQ, SFQ, WRR, VTRR, and SRR. Our experimental results using our GR^3 Linux implementation further demonstrate that GR^3 provides accurate proportional fairness behavior on real applications with much lower scheduling overhead than other Linux schedulers, especially for larger workloads.

We have analyzed and experimentally demonstrated that GR^3 provides accurate,

low overhead proportional share scheduling for uniprocessor systems. While effective uniprocessor scheduling is crucial in any system, multiprocessor systems are increasingly common. Providing accurate, low overhead proportional share scheduling for multiprocessor systems remains an important challenge. We believe that the ideas discussed here for uniprocessor scheduling will serve as a basis for future work in addressing the scheduling problem in the context of scalable multiprocessor systems.

Bibliography

[1] J. Bennett and H. Zhang, "WF^{2}Q: Worst-case Fair Weighted Fair Queueing," in *Proceedings of INFOCOM '96*, San Francisco, CA, Mar. 1996.

[2] D. Bovet and M Cesati, *Understanding the Linux Kernel*. Sebastopol, CA: O'Reilly, 1st ed., 2001.

[3] R. Bryant and B. Hartner, "Java technology, threads, and scheduling in Linux" *IBM developerWorks Library Paper*, IBM Linux Technology Center, Jan. 2000.

[4] G. Chuanxiong, "SRR: An 0(1) Time Complexity Packet Scheduler for Flows in Multi-Service Packet Networks," in *Proceedings of ACM SIGCOMM '01*, Aug. 2001, pp. 211–222.

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

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

[7] R. Essick, "An Event-Based Fair Share Scheduler," in *Proceedings of the Winter 1990 USENIX Conference*, USENIX, Berkeley, CA, USA, Jan. 1990, pp. 147–162.

[8] E. Gafni and D. Bertsekas, "Dynamic Control of Session Input Rates in Communication Networks," in *IEEE Transactions on Automatic Control*, 29(10), 1984, pp. 1009–1016.

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

[10] E. Hahne and R. Gallager, "Round Robin Scheduling for Fair Flow Control in Data Communication Networks," Tech. Rep. LIDS-TH-1631, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Dec. 1986.

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

[12] J. Kay and P. Lauder, "A Fair Share Scheduler," *Communications of the ACM*, 31(1), Jan. 1988, pp. 44–55.

[13] L. Kleinrock, *Queueing Systems, Volume II: Computer Applications*. New York: John Wiley & Sons, 1976.

[14] J. Xu and R. Lipton, "On Fundamental Tradeoffs between Delay Bounds and Computational Complexity in Packet Scheduling Algorithms," in *Proceedings of ACM SIGCOMM '02*, Pittsburgh, PA, August 2002.

[15] J. Nieh, C. Vaill, H. Zhong, "Virtual-time round-robin: An O(1) proportional share scheduler," in *Proceedings of the 2001 USENIX Annual Technical Conference*, USENIX, Berkeley, CA, June 25–30 2001, pp. 245–259

[16] 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), June 1993, pp. 344–357.

[17] J. Pasquale and S. Ramabhadran, "Stratified Round Robin: A Low Complexity Packet Scheduler with Bandwidth Fairness and Bounded Delay," *Proceedings of ACM SIGCOMM '03*, Karlsruhe, Germany, August 2003.

[18] K. Ramakrishnan, D. Chiu, and R. Jain, "Congestion Avoidance in Computer Networks with a Connectionless Network Layer, Part IV: A Selective Binary

Feedback Scheme for General Topologies," Tech. Rep. DEC-TR-510, DEC, Nov. 1987.

[19] Hackbench: A New Multiqueue Scheduler Benchmark.

`http://www.lkml.org/archive/2001/12/11/19/index.html` Message to Linux Kernel Mailing List, December 2001.

[20] M. Shreedhar and G. Varghese, "Efficient Fair Queueing Using Deficit Round-Robin," in *Proceedings of ACM SIGCOMM '95*, 4(3), Sept. 1995.

[21] D. Stiliadis, and A. Varma, "Efficient Fair Queueing Algorithms for Packet-Switched Networks," in *IEEE/ACM Transactions on Networking*, Apr. 1998, pp. 175–185.

[22] R. Tijdeman, "The Chairman Assignment Problem," *Discrete Mathematics*, 32, 1980, pp. 323–330.

[23] C. Waldspurger, *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sept. 1995.

[24] L. Zhang, "Virtual Clock: A New Traffic Control Algorithm for Packet Switched Networks," in *ACM Transactions on Computer Systems*, 9(2), May 1991, pp. 101–125,