Group Ratio Round-Robin: An O(1) Proportional Share Scheduler

Wong Chun Chan and Jason Nieh
{wc164, nieh}@cs.columbia.edu
Department of Computer Science
Columbia University
Technical Report CUCS-012-03
April 2003

Abstract

Proportional share resource management provides a flexible and useful abstraction for multiplexing time-shared resources. However, previous proportional share mechanisms have either weak proportional sharing accuracy or high scheduling overhead. We present Group Ratio Round-Robin (GR^3), a proportional share scheduler that can provide high proportional sharing accuracy with $O(1)$ scheduling overhead. Unlike many other schedulers, a low-overhead GR^3 implementation is easy to build 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 better scheduling accuracy in practice than these other approaches for large numbers of clients.

1 Introduction

Proportional share resource management provides a flexible and useful abstraction for multiplexing scarce resources among users and applications. Because of its usefulness, many proportional share scheduling mechanisms have been developed [2, 5, 6, 8, 11, 12, 15, 17, 20, 22, 23]. In addition, higher-level abstractions have been developed on top of these proportional share mechanisms to support flexible, modular resource management policies [20, 22].

Proportional share scheduling mechanisms were first developed decades ago with the introduction of weighted round-robin scheduling [18]. Starting in the late 1980s, fair queueing algorithms were developed [2, 5, 8, 15, 17, 20, 22, 23], first for network packet scheduling and later for CPU scheduling. These algorithms provided better proportional sharing accuracy. However, the time to select a client for execution using these algorithms grows with the number of clients. Most implementations require linear time to select a client for execution, though more complex logarithmic time implementations are possible. More recently, proportional share mechanisms with constant time scheduling overhead [3, 14] have been developed, but these algorithms need to sacrifice scheduling accuracy to reduce scheduling overhead.

We introduce GR^3, a Group Ratio Round-Robin scheduler, for proportional share resource management. GR^3 combines the benefits of low overhead round-robin execution with a novel ratio-based scheduling algorithm and client grouping strategy. It provides accurate control over client computation rates, and it can schedule clients for execution in $O(1)$ time. The constant scheduling overhead makes GR^3 particularly suitable for server systems and software routers that must manage large numbers of clients. 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 CPU scheduler in Linux, and compared our GR^3 Linux prototype against schedulers commonly used in practice and research, including the standard Linux scheduler [1], Weighted Fair Queueing [5], Virtual-Time Round-Robin [14], and Smoothed Round-Robin [3]. 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 for skewed share distributions. 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 and implementation of GR^3. Section 2 provides some background on proportional fairness. Section 3 discusses related work. Section 4 presents the GR^3 scheduling algorithm. Section 5 presents performance results from both simulation studies and real kernel measurements that compare GR^3 against other well-known scheduling algorithms. Finally, we present some concluding remarks and directions for

future work.

2 Proportional Fairness

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. In this paper, we use the term share and weight interchangeably. 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. The other way is to adjust the size of the time quantum of a client so that it runs longer for a given allocation. The manner in which a scheduler determines how often a client runs and how long a client runs directly affects the accuracy and scheduling overhead of the scheduler.

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 in more technical terms. The definition we use is a simple one that suffices for our discussion; more extended definitions are presented in [7, 10, 16, 21]. Our definition draws heavily from the ideal sharing mechanism GPS [13]. To simplify the discussion, we assume that clients do not sleep or block and can consume whatever resources they are allocated.

We first define *perfect fairness*, an ideal state in which each client has received service exactly proportional to its share. We denote the proportional share of client A as S_A, and the amount of service received by client A during the time interval (t_1, t_2) as $W_A(t_1, t_2)$. Formally, a proportional sharing algorithm achieves perfect fairness for time interval (t_1, t_2) if, for any client A,

$$W_A(t_1, t_2) = (t_2 - t_1)\frac{S_A}{\sum_i S_i} \quad (1)$$

If we had an ideal system in which all clients could consume their resource allocations simultaneously, then an ideal proportional share scheduler could maintain the above relationship for all time intervals. However, in scheduling a time-multiplexed resource in time units of finite size, it is not possible for a scheduler to be perfectly proportionally fair as defined by Equation 1 for all intervals.

Although no real-world scheduling algorithm can 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. We can use a variation of Equation 1 to define the *service time error* $E_A(t_1, t_2)$ for client A over interval (t_1, t_2). The error is the difference between the amount of time allocated to the client during interval (t_1, t_2) under the given algorithm, and the amount of time that would have been allocated under an ideal scheme that maintains perfect fairness for all clients over all intervals. Service time error is computed as:

$$E_A(t_1, t_2) = W_A(t_1, t_2) - (t_2 - t_1)\frac{S_A}{\sum_i S_i} \quad (2)$$

A positive service time error indicates that a client has received more than its ideal share over an interval; a negative error indicates that a client has received less. To be precise, the error E_A measures how much time client A has received beyond its ideal allocation. The goal of a proportional share scheduler should be to minimize the allocation error between clients with minimal scheduling overhead.

3 Related Work

One of the oldest, simplest and most widely used proportional share scheduling algorithms is round-robin. Clients are placed in a queue and allowed to execute in turn. When all client shares are equal, each client is assigned the same size time quantum. In the weighted round-robin case, each client is assigned a time quantum equal to its share. Weighted round-robin (WRR) provides proportional sharing by running all clients with the same frequency but adjusting the size of their time quanta. A more recent variant called deficit round-robin [17] has been developed for network packet scheduling with similar behavior to a weighted round-robin CPU scheduler. WRR is simple to implement and schedules clients in $O(1)$ time. However, it has a relatively weak proportional fairness guarantee as its service ratio error can be quite large. Consider an example in which 3 clients A, B, and C, have shares 3, 2, and 1, respectively. WRR will execute these clients in the following order of time units: A, A, A, B, B, C. The error in this example gets as low as -1 tu and as high as $+1.5$ tu. The real trouble comes with large share values: if the shares in the previous example are changed to 3000, 2000, and 1000, the error ranges instead from -1000 to $+1500$ tu. A large error range like this illustrates the major drawback of round-robin scheduling: each client gets all service due to it all at once, while other clients get no service.

Fair-share schedulers [6, 11, 12] arose as a result of a need to provide proportional sharing among users in a way compatible with a UNIX-style time-sharing framework. In UNIX time-sharing, scheduling is done based on multi-level feedback with a set of priority queues. Each client has a priority which is adjusted as it executes. The scheduler executes the client with the highest priority. The idea of fair-share was to provide proportional sharing among clients by adjusting the priorities of clients in a suitable way. The priority adjustments were generally computed in $O(1)$ time, though in some cases, the schedulers needed to do an expensive periodic re-adjustment of all client priorities, which required $O(N)$ time, where N is the number of clients. Fair-share schedulers were compatible with UNIX scheduling frameworks and relatively easy to deploy in existing UNIX environments. However, the approaches were often ad-hoc and it is difficult to formalize the proportional fairness guarantees they provided. Empirical measurements show that these approaches only provide reasonable proportional fairness over relatively large time intervals [6]. It is almost certainly the case that the allocation errors in these approaches can be very large.

Lottery scheduling [22] provides a less ad-hoc proportional sharing approach than fair-share schedulers. Each client is given a number of tickets proportional to its share. A lottery scheduler then randomly selects a ticket and schedules the client that owns the selected ticket to run for a time quantum. Lottery scheduling provides proportional sharing by running clients at different frequencies by adjusting the position at which each client is inserted back into the queue; the same size time quantum is typically used for all clients. Lottery scheduling requires $O(N)$ time for a linear list of clients and at least $O(\log N)$ time for tree structures, 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 smaller share values.

Fair queueing was first proposed by Demers et. al. for network packet scheduling as Weighted Fair Queueing (WFQ) [5], with a more extensive analysis provided by Parekh and Gallager [15], and later applied by Waldspurger and Weihl to CPU scheduling as stride scheduling [22]. Other variants of WFQ such as Virtual-clock [23], SFQ [9], FFQ, SPFQ [19], and Time-shift FQ [4] have also been proposed. WFQ introduced the idea of a virtual finishing time (VFT) to do proportional sharing scheduling. To explain what a VFT is, we first explain the notion of virtual time. The *virtual time* of a client is a measure of the degree to which a client has received its proportional allocation relative to other clients. When a client executes, its virtual time advances at a rate inversely proportional to the client's share. Given a client's virtual time, the client's *virtual finishing time* (VFT) is defined as the virtual time the client would have after executing for one time quantum. WFQ then schedules clients by selecting the client with the smallest VFT, which requires keeping an ordered list of clients sorted by VFT. This requires $O(N)$ time for a linear list of clients and at least $O(\log N)$ time for tree structures, where N is the number of clients. Fair queueing provides proportional sharing by running clients at different frequencies by adjusting the position in at which each client is inserted back into the queue; the same size time quantum is used for all clients.

To illustrate how this works, consider again the example in which 3 clients A, B, and C, have shares 3, 2, and 1, respectively. Their initial VFTs are then 1/3, 1/2, and 1, respectively. WFQ would then execute the clients in the following order of time units: A, B, A, B, C, A. In contrast to WRR, WFQ's service time error ranges from $-5/6$ to $+1$ tu in this example, which is less than the allocation error of -1 to $+1.5$ tu for WRR. The difference between WFQ and WRR is greatly exaggerated if larger share values are chosen: if we make the shares 3000, 2000, and 1000 instead of 3, 2, and 1, WFQ has the same service time error range while WRR's error range balloons to -1000 to $+1500$ tu. 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 [15]. However, WFQ can allow a client to get far ahead of its ideal allocation and accumulate a large positive service time error especially in the presence of skewed share distributions. For example, given a client A with share 100 and 100 clients each with share 1, WFQ will run client A for 100 time quanta before running any of the other clients resulting in a service time error of $+50$ tu.

Several approaches have been proposed for reducing this service time error in the presence of skewed share distributions. One approach is to use a hierarchical scheduling approach 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 [22]. More recent fair queueing algorithms [2, 20] such as Worst-Case Weighted Fair Queueing [2] introduce 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. Unfortunately, they are more difficult to implement, and the time required to select a client to execute is at least $O(\log N)$ time, where N is the number of clients.

More recently, novel round-robin scheduling variants such as Virtual-Time Round-Robin (VTRR) [14] and Smoothed Round Robin (SRR) [3] have been developed that combine the benefits of constant-time scheduling overhead like round-robin with scheduling accuracy that approximates fair queueing. These mechanisms pro-

vide 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 VFTs used in fair queueing. It maintains a list of clients from largest to smallest share that does not change on each schedule, runs each client in turn for a time quantum until it reaches a client whose VFT indicates that it has received more than its fair allocation, then returns to the beginning of the list. SRR introduces a Weight Matrix and Weight Spread Sequence (WSS). The Weight Matrix consists of binary vectors coded from the shares of the clients. SRR then scans the elements of the Weight Matrix in a fixed order specified by WSS and selects the client to execute whose share corresponds to the matrix element selected. 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 but are typically less frequent. 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 $k = \log S_{max}$ and S_{max} is the maximum client share allowed. However like WFQ, both algorithms suffer from large service time errors especially in the case of skewed share distributions. Revisiting the example of 101 clients with one client A with share 100 and 100 clients each with share 1, the service time error for client A can range from -49.5 to 0.99 tu for VTRR and can range from -25 to 25 tu for SRR.

4 GR^3 Scheduling

GR^3 is a proportional share scheduler that schedules with $O(1)$ time complexity like recent round-robin scheduling variants but with much lower service time errors in practice. In designing GR^3, we observed that accurate, low-overhead proportional sharing is easy to achieve when all clients have equal shares, but is harder to do when clients have skewed share distributions. Based on this observation, GR^3 uses a novel client grouping strategy to organize clients into groups of similar share values 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 share values. Each group k is assigned clients with share values 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 shares, where the group share of a group is the sum of the shares of all clients in the group. Groups are selected in a round-robin manner based on the ratio of their group shares. If a group has already been selected more than its proportional share of the time, 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 share values are placed first in the list, this allows them to get more service than the lower-share 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 skips over clients which have already received their desired proportional share resource allocation.

Using this client grouping strategy, GR^3 separates scheduling in such a way that reduces the need to schedule entities with skewed share 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 share value. Even a very large 32-bit client share 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 which limits the impact of skewed share distributions on groups. For example, while it would be possible to have one group with share 100 and a few groups each with share 1 to schedule, it would not be possible to have the example in Section 3 with a group with share 100 and 100 groups with share 1 to schedule. The client grouping strategy also limits the share distributions that the intragroup scheduler needs to consider since the range of share values within a group is less than a factor of two. As a result, the intragroup scheduler never needs to schedule clients with skewed share distributions since the clients within a group must have relatively similar share values. 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.

4.1 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. In GR^3, a client has three values associated with its execution state: share, counter, and run state. A client's *share* defines its resource rights. Each client receives a resource allocation that is directly proportional to its share. A client's *counter* tracks the number of time quanta the client has recently received. 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 share, group counter, last group counter, group number, group ratio, and current client. The *group share* and *group counter* are defined as the sum of the corresponding attributes of the clients in the group run queue. The *last group counter* is the value of the group counter at the beginning of the group, and is described in more detail below. The *group number* uniquely identifies a group in the group list and determines the clients that are in the group. A group with *group number* k contains clients with share values between 2^k to $2^{k+1} - 1$. The *group ratio* is the ratio between the *group share* of the group and the next group in the group list. 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 share, total counter and current group. As discussed in Section 2, 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 share. If two groups have the same group shares, the group number is used to break the tie in ordering the groups on the group list. The *total share* is the sum of the shares of all runnable clients. The *total counter* is the sum of the counters of all runnable clients. The *current group* is the most recently selected group in the group list.

4.2 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 Section 4.3. 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.

First, we define a *scheduling cycle* as a sequence of allocations whose length is equal to the sum of all client shares. For example, for a system of three clients with shares 3, 2, and 1, a scheduling cycle is a sequence of 6 allocations. In the context of a scheduling cycle, we can explain the role of the counters in GR^3. At the beginning of each scheduling cycle, the client counters, group counters and total counter are reset to the respective client shares, group shares, and total shares. Every time a client is run, the client's counter, the group counter of the client's group, and the total counter are decremented. GR^3 uses the counters to ensure that perfect fairness is attained at the end of every scheduling cycle. At the end of the cycle, every client counter will be zero, meaning that for each client A, the number of quanta received during the cycle is exactly equal to the client share S_A. Clearly, then, each client has received service proportional to its share during the cycle.

The GR^3 intergroup scheduling algorithm uses the group ratios between groups to determine which group to select. Initially, at the beginning of a scheduling cycle, the group ratio of each group is set equal to the ratio of its group share to the group share of the next group in the group list. The group ratio of the last group in the group list is defined to be one. Since the group list is sorted from largest to smallest group share, this ratio of group shares is always greater than or equal to one. If we denote the group share of a group G as S_G and the group share of the next group in the group list as $S_{G_{next}}$, the group ratio R_G for group G is simply:

$$R_G = \frac{S_G}{S_{G_{next}}} \tag{3}$$

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 share. 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 current group counter, current client counter, and total counter are decremented by one. The current group ratio is also decremented by one. If the current group ratio becomes less than one, then the current group ratio $R_{current}$ is incremented from its previous value $R^{old}_{current}$ as follows:

$$R_{current} = R^{old}_{current} + \frac{S_{current}}{S_{G_{next}}} \tag{4}$$

where $S_{current}$ is the group share of the current group and $S_{G_{next}}$ is the group share of the next group in the group list. The next group G_{next} on the group list is then assigned to be the current group. Otherwise if the current group ratio is greater than or equal to one, the current group is reset to the largest share group at the beginning of the group list.

The GR^3 intragroup scheduling algorithm selects the current client within a group in a round-robin manner that accounts for the amount of service each client has already received. Initially, the current client within a group is set to be the first client at the beginning of the group's run queue. Like round-robin scheduling, the run queue does not need to be sorted in any manner. Once the current client has completed its time quantum, the current client is set to the next client on the group's run queue that has not exceeded its proportional share of service. While this can be done in a number of ways, GR^3 uses an approach that considers the scheduling of clients in rounds. A *round* is one pass through the run queue from the beginning of the run queue until the end of the run queue. In each round, GR^3 determines at the beginning of the round which clients still have at least as much remaining time to run in the scheduling cycle as their proportional share of service, then runs those clients during the round. To do this, GR^3 sets the last group counter

C_G^{last} equal to the group counter C_G at the beginning of a round. Given a client A with share S_A and counter C_A in a group G with group share S_G, GR^3 runs a client if the following inequality holds when $C_G^{last} > 0$:

$$\frac{S_A}{S_G} \leq \frac{C_A}{C_G^{last}} \tag{5}$$

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. However, because all clients have share values within a factor of two of each other within a group, no client will be skipped more than half the time it is considered for execution.

At the end of the scheduling cycle, when the total counter becomes zero, all the client counters, group counters, and total counter are reset to the corresponding shares. The scheduler restarts from the beginning of the largest group run queue. Note that throughout this scheduling process, the ordering of the groups on the group list and of the clients on the each group run queue do not change.

To illustrate how GR^3 works, consider again the example in which 3 clients A, B, and C, have shares 3, 2, and 1, respectively. Group number 1 would consist of client A and client B and have a group share of 5, and group number 0 would have client C and a group share of 1. Group 1 has a larger group share so it is the first group on the group list. The group ratios are 5 for group 1 and 1 for group 0. GR^3 would execute clients from group 1 for five time quanta then the client in group 0 for one time quantum. If client B happens to be before client A on the run queue of group 1, GR^3 would then execute the clients in the following repeating order of time units: B, A, A, B, A, C. In this example, the service time error for the clients as scheduled by GR^3 would range from $-5/6$ to $+2/3$ tu.

Unlike other approaches such as WFQ, GR^3 provides much lower service time error for skewed distributions. Consider again the example from Section 3 of 101 clients with a client A with share 100 and 100 clients each with share 1. GR^3 would divide the clients into two groups. Group number 6 would consist of client A and have a group share of 100, and group number 0 would have the other 100 clients and also have a group share of 100. Since both groups have the same group shares, the group number is used to order the groups so group 6 would be the first group on the group list. The group ratios are 1 for both groups. GR^3 would then alternate between the groups to execute clients for one time quantum each. For this skewed share distribution, GR^3 maintains a low service time error range of -0.990 to $+0.995$ tu.

4.3 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, change run state, and change their share assignments.

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 A with share S_A becomes runnable, it is inserted into the group k such that S_A is between 2^k and $2^{k+1} - 1$. 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 in a manner based on its previous execution history. If the client has not run in the current scheduling cycle, GR^3 inserts the client 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. If the client has run in the current scheduling cycle, GR^3 inserts the client right after the current client. This allows the newly runnable client to be serviced the next time the group is selected and is based on the rationale that the client had recently run but became not runnable just for a brief time and so should be allowed to continue running again where it left off.

When the newly runnable client is inserted into the group, the client's counter, group counter, total counter, group share, and total share need to be updated. The total share and group share are simply updated by incrementing the respective values by the client share S_A. Let S_{TOTAL}^{old} be the previous total share and S_G^{old} be the previous group share. Then the total share S_{TOTAL} and group share S_G after inserting client A are:

$$S_{TOTAL} = S_{TOTAL}^{old} + S_A \tag{6}$$

$$S_G = S_G^{old} + S_A \tag{7}$$

Since a client may become runnable in the middle of a scheduling cycle, the client counter should not simply be set equal to the client share but should instead be updated in a manner that accounts for how far the scheduler has progressed into the scheduling cycle. This provides the client with a proportionally fair allocation based on the time that it is runnable. Toward this end, the counters are updated based on the ratios of the old and new total share values. Let C_{TOTAL}^{old} be the previous total counter and C_G^{old} be the previous group counter. Then the total

counter C_{TOTAL}, client counter C_A, and group counter C_G after inserting client A are:

$$C_{TOTAL} = C^{old}_{TOTAL} \times \frac{S_{TOTAL}}{S^{old}_{TOTAL}} \quad (8)$$

$$C_A = C_{TOTAL} - C^{old}_{TOTAL} \quad (9)$$

$$C_G = C^{old}_G + C_A \quad (10)$$

Since the group share of a group changes when a client is inserted into the group, the group's relative position on the group list may change and its group ratio as well as its predecessor on the group list will have to be recomputed. Since the group share can only increase due to a client insertion, there are three cases to consider: (1) the position of the group moves closer to the front of the group list because its group share is now larger than previous groups on the group list, (2) the position of the group does not change on the group list because its group share remains less than the previous group in the group list, and (3) the group was not previously on the group list because it was empty before the newly runnable client was inserted in the group and now needs to be inserted on the group list.

In these cases, the group ratios will need to be updated by scaling it to reflect the change in shares. We first describe the general equation for updating the group ratios and then apply the equation to each of the three cases. Given a group i, we denote its current group share and group ratio as S_i and R_i, respectively, and its old group share and group ratio as S^{old}_i and R^{old}_i, respectively. Let G be a group whose group values have been updated and whose whose next group G_{next} in the group list has been updated from a previous group $G_{next-old}$. The group ratio R_G of group G is one if it is the last group in the group list, otherwise it is updated from its previous value R^{old}_G as follows:

$$R_G = R^{old}_G \times \frac{S_{G_{next-old}}}{S^{old}_G} \times \frac{S_G}{S_{G_{next}}} \quad (11)$$

When the position of a group moves closer to the front of the group list due to client insertion in the group, its group ratio, the group ratio of its predecessor group in the group list before changing the position of the group, and the group ratio of its predecessor group in the group list after changing the position of the group need to be updated. If the group becomes the first group on the group list, no predecessor group will exist or need to be updated. Given a group G whose group values have been updated from its old values due to client insertion, let G_{prev} be its previous group in the group list and G_{next} be the next group in the group list after it has moved, and let $G_{prev-old}$ be its previous group in the group list and $G_{next-old}$ be the next group in the group list before it moved. Group ratio R_G is updated as shown in Equation 11. Assuming G is not the first group on the group list, we can then substitute into Equation 11 to update the group ratios $R_{G_{prev-old}}$ and $R_{G_{prev}}$ as follows:

$$R_{G_{prev-old}} = R^{old}_{G_{prev-old}} \times \frac{S^{old}_G}{S_{G_{next-old}}} \quad (12)$$

$$R_{G_{prev}} = R^{old}_{G_{prev}} \times \frac{S_{G_{next}}}{S_G} \quad (13)$$

When the group position stays the same after client insertion, only its group ratio and the group ratio of its predecessor group in the group list need to be updated if the group is not the first group in the group list. In this case when the group position does not change, $S_{G_{next}}$ and $S_{G_{next-old}}$ from Equation 11 are the same and $R_{G_{prev-old}}$ and $R_{G_{prev}}$ from Equations 12 and 13 are the same. We can then reduce Equations 11 to 13 to update the group ratios R_G and and $R_{G_{prev}}$ as follows:

$$R_G = R^{old}_G \times \frac{S_G}{S^{old}_G} \quad (14)$$

$$R_{G_{prev}} = R^{old}_{G_{prev}} \times \frac{S^{old}_G}{S_G} \quad (15)$$

When the group was not previously on the group list and is now inserted into the group list, the group ratio of the newly inserted group G and the group ratio of its predecessor group G_{prev} in the group list need to be updated if the group is not the first group on the group list. Assuming G is not the first group on the group list, $R_{G_{prev}}$ can be updated using Equation 13. However, the newly inserted group G has no old values from Equation 11. We instead scale its group ratio by its group counter C_G which indicates how much of the group ratio should have already been completed in the current scheduling cycle. Group ratio R_G is derived from Equation 3 and computed as follows:

$$R_G = \frac{C_G}{S_{G_{next}}} \quad (16)$$

When a client becomes runnable and is inserted into a group, in most cases the current group is not affected. The one exception is if the group in which the client was inserted is the current group and the position of the group in the group list now changes as a result of its larger group share. In this case, we determine which group is to next be set to be the current group based on the position of the current group before the client insertion. Once the current client has completed its time quantum, the current group ratio is decremented by one as previously described in Section 4.2. If the current group ratio becomes less than one, the next group $G_{next-old}$ on the group list, as determined based on the group list ordering before client insertion, is assigned to be the current group. Otherwise, the current group is reset to the largest share group at the beginning of the group list.

Having described what happens when a client becomes runnable, we now discuss what happens with GR^3

when a client becomes not runnable. When a client A with share S_A becomes not runnable, it is removed from the group k such that S_A is between 2^k and $2^{k+1} - 1$. When the client is removed from its group, the group counter, total counter, group share, and total share need to be updated. The total share and group share are simply updated by decrementing the respective values by the client share S_A. Let S_{TOTAL}^{old} be the previous total share and S_G^{old} be the previous group share. Then the total share S_{TOTAL} and group share S_G after removing client A are:

$$S_{TOTAL} = S_{TOTAL}^{old} - S_A \tag{17}$$

$$S_G = S_G^{old} - S_A \tag{18}$$

Similarly, the total counter and group counter are simply updated by decrementing the respective values by the client counter C_A. Let C_{TOTAL}^{old} be the previous total counter and C_G^{old} be the previous group counter. Then the total counter C_{TOTAL}, client counter C_A, and group counter C_G after removing client A are:

$$C_{TOTAL} = C_{TOTAL}^{old} - C_A \tag{19}$$

$$C_G = C_G^{old} - C_A \tag{20}$$

Since the group share of a group changes when a client is removed from the group, the group's relative position on the group list may change and its group ratio as well as its predecessor on the group list will have to be recomputed. Since the group share can only decrease due to a client removal, there are three cases to consider: (1) the position of the group moves closer to the back of the group list, (2) the position of the group does not change on the group list, and (3) the group is now empty because its one client was removed and the group needs to be removed from the group list.

We can update the group ratios in these cases in a manner similar to the case of client insertion using Equation 11. More specifically, the group ratios for the case in which the group position moves closer to the back of the group list can be computed in the same manner as the corresponding case for client insertion by using Equations 12 to 13. Similarly, the group ratios for the case in which the group position does not change due to client removal can be computed in the same manner as the corresponding case for client insertion by using Equations 14 and 15.

When the group is now empty and needs to be removed from the group list, only the group ratio of the predecessor group in the group list need to be updated if the group is not the first group on the list. Given a group G that is to be removed with group share S_G^{old} before client removal, let G_{prev} be its previous group in the group list and G_{next} be the next group in the group list before removal. Assuming G is not the first group on the group list, we can then substitute into Equation 11 to update the group ratio $R_{G_{prev}}$ as follows:

$$R_{G_{prev}} = R_{G_{prev}}^{old} \times \frac{S_G^{old}}{S_{G_{next}}} \tag{21}$$

When a client becomes not runnable and is removed from a group, in most cases the current group is not affected. However, there are two exceptions. One exception is if the group from which the client was removed is the current group and the position of the group in the group list now changes as a result of its smaller group share. In this case, we determine which group is next to be the current group based on the position of the current group before the client removal. Once the current client has completed its time quantum, the current group ratio is decremented by one as previously described in Section 4.2. If the current group ratio becomes less than one, the next group $G_{next-old}$ on the group list, as determined based on the group list ordering before client removal, is assigned to be the current group. Otherwise, the current group is reset to the largest share group at the beginning of the group list. The other exception is if the group from which the client was removed is the current group and is now empty and removed. In this case, we again determine which group is next to be the current group based on the position of the current group before the client removal. The current group ratio is decremented by one as previously described in Section 4.2. If the current group ratio becomes less than one, the next group $G_{next-old}$ on the group list, as determined based on the group list ordering before client removal, is assigned to be the current group. Otherwise, the current group is reset to the largest share group at the beginning of the group list.

If the share of a client changes, there are two cases to consider based on the run state of the client. If the client is not runnable, only the client's share is updated and no other changes are needed. If the client is runnable and its share changes, the client's group may need to be changed and the group that the client belongs to will need to be updated appropriately. This operation can be logically simplified by removing the client from its group as in the case when a client becomes not runnable, changing the client share, and then reinserting the client back into a group based on its new share value. Removal and insertion can then be performed just as described above.

4.4 Complexity and Fairness

The primary function of a scheduler is to select a client for service when the resource is available. A key benefit of GR^3 is that it can select a client for service in $O(1)$ time. GR^3 intergroup scheduling selects a group from which to choose a client in $O(1)$ time and GR^3 intragroup scheduling selects a client for service within a group in $O(1)$ time. In selecting a group, GR^3 only has to decide whether to select the next group in the group list or to go back to the beginning of the group list and

select the first group. This decision only requires GR^3 to examine the counter of the current group which can be done in $O(1)$ time. The GR^3 intergroup scheduler does need to maintain a sorted list of groups from largest to smallest group share. However, the ordering of groups on the group list does not change in the normal process of selecting a group from which to choose a client. Once a group has been selected, GR^3 selects a client within a group in a round-robin manner that skips clients that have already received their proportional resource allocation according to Equation 5. Because all clients in a group have share values within a factor of two of each other, no client will be skipped more than half the time it is considered for execution. As a result, this intragroup scheduling decision can also be done in $O(1)$ time. Because groups do not need to be resorted during scheduling and clients within a group are not sorted, GR^3 provides an important advantage over fair queueing algorithms, which need to reinsert a client into a sorted run queue after each time it is serviced. This is at least an $O(\log N)$ time operation per scheduling decision. As a result, fair queueing algorithms require higher time complexity than GR^3.

When the total counter becomes zero, GR^3 does reset the counters of all groups and all clients in each group to their respective share values. The complete counter reset takes $O(N)$ time, where N is the number of clients. However, this reset is done at most once every N times the scheduler selects a client to execute, and much less frequently in practice. As a result, the reset of the counters is amortized over at least N client selections so that the effective running time of GR^3 to select a client for service is still $O(1)$ time.

In addition to selecting a client to execute, a scheduler must also allow clients to be dynamically created and terminated, change run state, and change scheduling parameters such as a client's weight. These scheduling operations typically occur much less frequently than client selection. Operations such as client creation, termination, and changing run state can result in a client becoming runnable or not runnable. In GR^3, making a client runnable or not runnable results in the need to insert or remove the client from a group run queue, respectively. As mentioned earlier, changing a client's share assignment can be decomposed into three steps: (1) removing the client from the group run queue, changing the client share, and then inserting the client back into a group run queue. The time complexity of all of these operations can be reduced to the time to insert or remove a client from a group run queue. In GR^3, inserting or removing a client from a group run queue can be done in $O(1)$ since there is a reference to the current client in the group and the group run queue is not sorted. However, these operations can increase or decrease the group share such that the group needs to be moved on the group list to maintain its ordering by group share value. Moving the group to its correct position on the group list can take at most $k-1$ swaps if a simple linear insertion is used, where k is the number of groups on the group list. This can easily be reduced to $O(\log k)$ by performing a binary search on the group list. This complexity is lower than recent low-overhead proportional share schedulers such as VTRR and SRR.

An $O(\log k)$ or even an $O(k)$ operation is effectively constant time in practice given that the number of groups that need to be used is typically quite small. The number of groups only depends on the range of client shares in the system. If there are clients with a wide range of shares from very large to small, more groups will need to be used. However, even in this case, the number of groups is at most bounded by the largests and smallest client shares in the system. If S_{max} and S_{min} are the largests and smallest client shares in the system, then $k \leq 1 + \log_2 \frac{S_{max}}{S_{min}}$. Note that the number of groups k is independent of the number of clients in the system and k is likely to be a small number compared to the number of clients for systems that must support large numbers of clients.

GR^3 not only provides low scheduling overhead, but it can also achieve low service time error in practice. GR^3 limits service time error by dividing up the scheduling problem into intragroup scheduling and intergroup scheduling. Because client shares within each group can not differ by more than a factor of two, GR^3 can use a simple intragroup scheduling algorithm and still bound the service time error within a group between -2 and $+1$. GR^3 intergroup scheduling does not have the same constant bound as intragroup scheduling but instead can result in worst-case service time errors of $O(k)$, where k is the number of groups. However, because the number of groups in practice is small and bounded, GR^3 intergroup scheduling can effectively provide a bound on service time error among groups as well.

5 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 WRR, WFQ, VTRR, and SRR. We used a simulator for these studies for two reasons. First, our simulator enabled us to isolate impact of the scheduling algorithms themselves and purposefully do not include the effects of other activity 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 share 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 Section 5.1.

We also conducted detailed measurements of real kernel scheduler performance by comparing our prototype GR^3 Linux implementation against both the standard Linux 2.4 scheduler and a WFQ 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 Sections 5.2 to 5.3.

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.

5.1 Simulation Studies

We built a scheduling simulator that is a user-space program which measures the service time error, described in Section 2, 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 shares S, and the number of client-share combinations. The simulator randomly assigns shares to clients and scales the share 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 share assignments. The simulator assumes that all clients are runnable at all times. This process of random share allocation and scheduler simulation is repeated for the specified number of client-share combinations. We then compute an average maximum service time error and average minimum service time error for the specified number of client-share combinations to obtain an "average-case" error range.

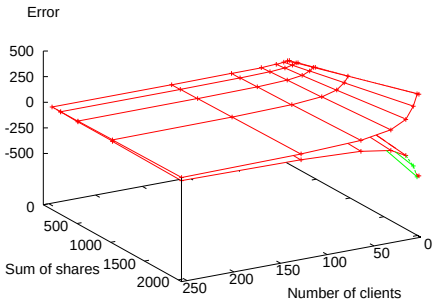

Figure 1: WRR service time error, random share allocation

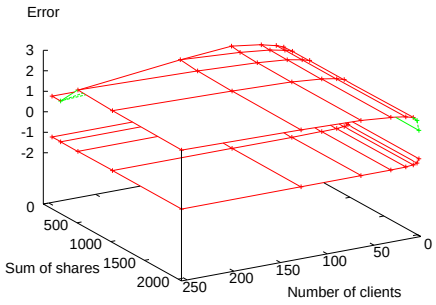

Figure 2: WFQ service time error, random share allocation

To measure proportional fairness accuracy, we ran simulations for each scheduling algorithm considered on 40 different combinations of N and S. For each set of (N, S), we ran 2500 client-share combinations and determined the resulting average error ranges. The average service time error ranges for WRR, WFQ, VTRR, SRR, and GR^3 are shown in Figures 1 to 5. 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.

Figure 1 shows the service time error ranges for WRR. Within the range of values of N and S shown, WRR's error range is between -406 tu and 407 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

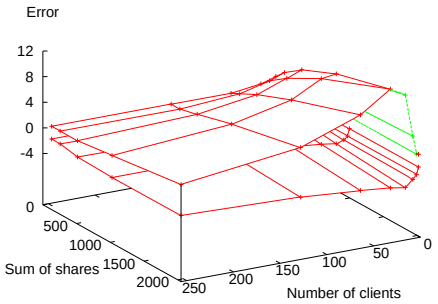

Figure 3: VTRR service time error, random share allocation

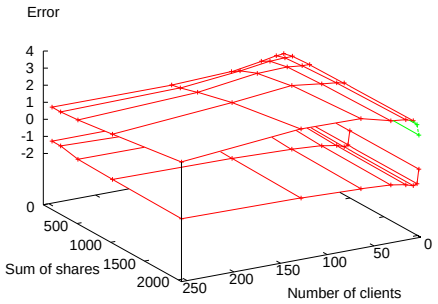

Figure 5: GR^3 service time error, random share allocation

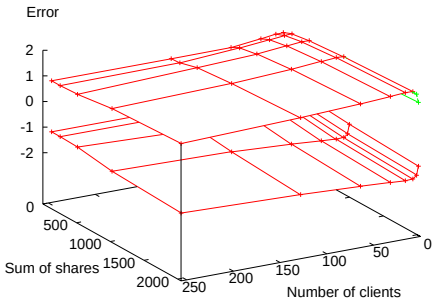

Figure 4: SRR service time error, random share allocation

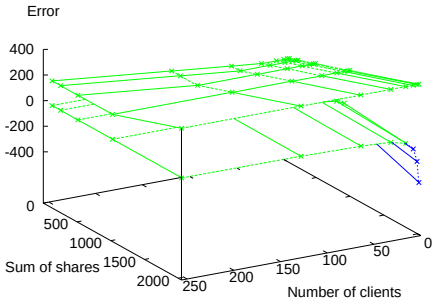

Figure 6: WRR service time error, skewed share allocation

proportional share CPU time allocation by more than 4 seconds, which is a substantial amount of service time error. Figure 2 shows the error ranges for WFQ. WFQ's error range is between -1 tu and 2 tu, which is much less than WRR. Note that another fair queueing algorithm WF^{2}Q was not simulated, but its error is mathematically bounded [2] between -1 and $+1$ tu. Figures 3 and 4 show the service error ranges for VTRR and SRR, respectively. VTRR's error range is between -3.8 to 11.9 tu. SRR's error range is between -1.7 to 1.7 tu. Although VTRR and SRR's error ranges are somewhat worse than WFQ, VTRR and WRR can schedule in $O(1)$ time.

In comparison, Figure 5 shows the service time error ranges for GR^3. GR^3's service time error only ranges from -1.3 to 1.2 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 smaller positive service error than WFQ, WFQ does have a smaller negative service time error since it is mathematically bounded below at -1. 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.

Since the proportional sharing accuracy of a scheduler is often most clearly illustrated with skewed share distributions, we repeated the simulation studies over the same range of N and S but with one of the clients given a share equal to 75 percent of S. All of the other clients were then randomly assigned shares to sum to the remaining 25 percent of S. We again considered 40 different combinations of N and S. For each set of (N, S), we ran 2500 client-share combinations and determined the resulting average error ranges. The average service time error ranges for WRR, WFQ, VTRR, SRR, and GR^3 with these skewed share distributions are shown in Figures 6 to 10. 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.

Figure 6 shows the service time error ranges for WRR. As expected, WRR's error range of -384 to 384 tu was the worst among the five scheduling algorithms. Figures 7, 8, and 9 show the service time error ranges for WFQ, VTRR, and SRR, respectively. WFQ's error range is -1 to 191 tu, VTRR's error range is -191 to 53 tu, and SRR's error range is -96 to 96 tu. Unlike the earler results for random share distributions, the service error ranges of these three schedulers are much larger for the skewed share distributions. In contrast, Figure 10 shows that the service time errors for GR^3 remain relatively low even for the skewed share distributions. GR^3's error range is only -1.3 to 3.5 tu, which is more than an order of magnitude less than the error ranges of all of the other schedulers.

The data produced by our simulations show that GR^3 has fairness properties that in practice can be much

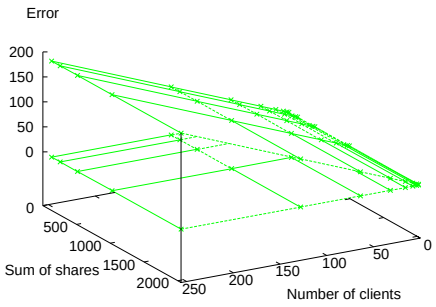

Figure 7: WFQ service time error, skewed share allocation

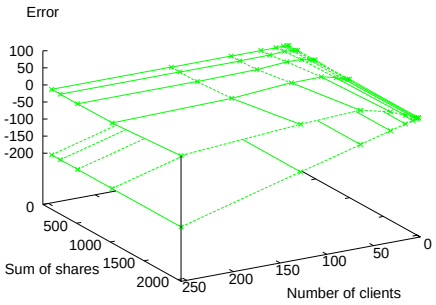

Figure 8: VTRR service time error, skewed share allocation

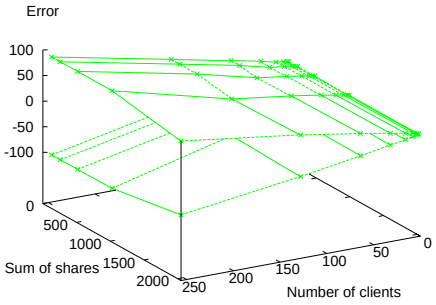

Figure 9: SRR service time error, skewed share allocation

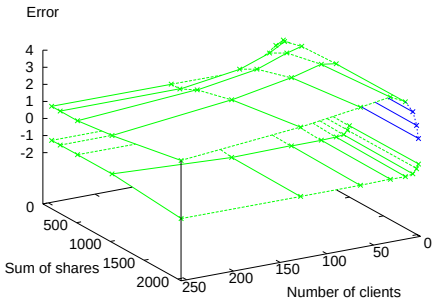

Figure 10: GR^3 service time error, skewed share allocation

better than WRR, WFQ, VTRR, and SRR. For the domain of values simulated, the service time error for GR^3 falls into an average range two orders of magnitude smaller than WRR. The service time error for GR^3 is consistently better than VTRR or SRR for both random and skewed share distributions. While GR^3 does not provide a service time error lower bound of -1 like WFQ, it has lower service time errors on average than WFQ especially for skewed share distributions, in which GR^3 can deliver more than an order of magnitude better proportional sharing accuracy. Furthermore, we show in Section 5.2 that GR^3 provides this degree of accuracy with much lower overhead than WFQ.

5.2 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 this 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 client up to 400 clients.

Figure 11 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 11, 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

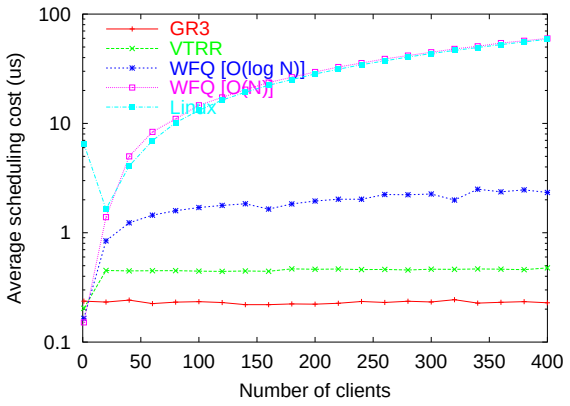

Figure 11: Average scheduling overhead

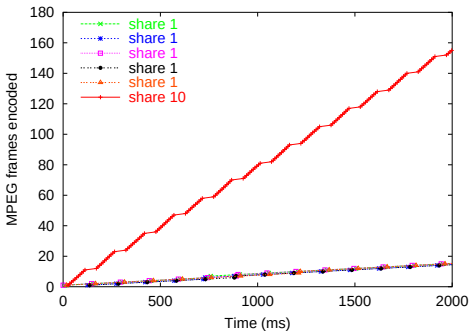

Figure 12: MPEG encoding with Linux scheduler

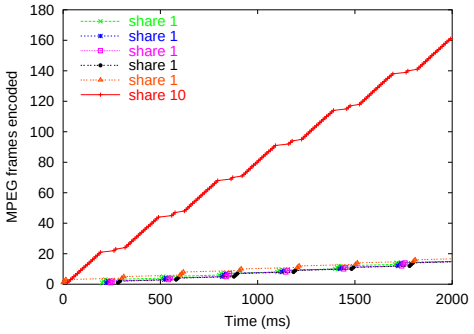

Figure 13: MPEG encoding with WFQ

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. The Linux scheduler imposes 200 times more overhead than GR^3 when scheduling a mix of 400 clients. In fact, the Linux scheduler still spends more than 50 times as long scheduling a single micro-benchmark client as GR^3 does scheduling 400 clients. GR^3 outperforms Linux even for small numbers of clients because the GR^3 scheduling code is simpler and the Linux scheduler overhead is proportional to the total number of clients in the system, not just the runnable clients. As a result, sleeping Linux system processes increase Linux scheduling overhead. GR^3 performs even better compared to Linux and WFQ for large numbers of clients because it has constant time overhead as opposed to the linear time overhead of the other schedulers. Because of the importance of low 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.

While $O(\log N)$ WFQ has much smaller overhead than Linux or $O(N)$ WFQ, it still imposes significantly more overhead than GR^3, particularly with large numbers of clients. With 400 clients, $O(\log N)$ WFQ has an overhead roughly 8 times that of GR^3. WFQ's more complex data structures require more time to maintain, and the time required to make a scheduling decision is still dependent on the number of clients, so the overhead would only continue to grow worse as more clients are added. GR^3's scheduling decisions always take the same amount of time, regardless of the number of clients.

5.3 Application Workloads

To demonstrate GR^3's efficient proportional sharing of resources on real applications, we briefly describe two of our experiments 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 shares on each of the four schedulers. We ran two tests. The first encoder test ran five copies of the encoder with respective shares 1, 2, 3, 4, and 5. The second encoder test ran six copies of the encoder with one encoder client assigned a share of 10 and the other encoder clients each assigned a share of 1. For the Linux scheduler, shares 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 share on single-frame encoding time.

GR^3 provided the best overall proportional fairness for these experiments while Linux provided the worst overall proportional fairness. Due to space constraints, we only present measurements for the second encoder

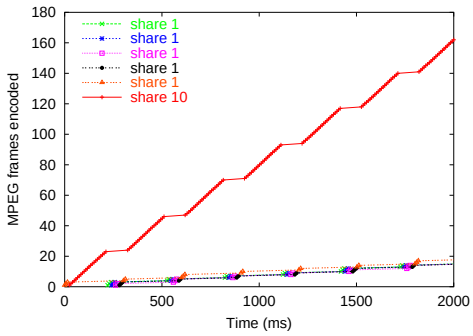

Figure 14: MPEG encoding with VTRR

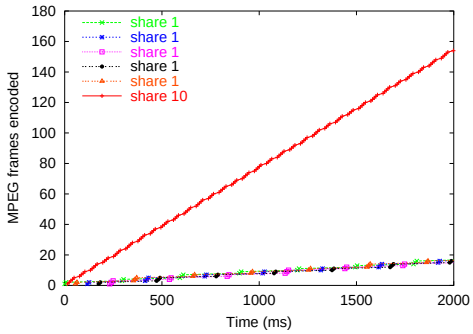

Figure 15: MPEG encoding with GR^3

test. Figures 12 to 15 show the number of frames encoded over time for the Linux scheduler, WFQ, VTRR, and GR^3. All of the schedulers except GR^3 have a pronounced "staircase" effect for the encoder with share 10, 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. It can be inferred from the smoother curves of Figure 15 that GR^3 provides fair resource allocation at a finer granularity than the other schedulers.

6 Conclusions and Future Work

We have designed, implemented, and evaluated Group Ratio Round-Robin scheduling in the Linux operating system. 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 our Linux implementation and demonstrated that GR^3 combines the benefits of accurate proportional share resource management with very low overhead. Our results show that GR^3 can provide lower scheduling overhead and better proportional fairness in practice than the Linux scheduler and proportional share schedulers such as WFQ, VTRR, and SRR. GR^3 scheduling overhead is constant, even for large numbers of clients, and is effective at handling skewed share distributions where many other proportional share schedulers are less accurate. GR^3's ability to provide low overhead proportional share resource allocation makes it a particularly promising solution for managing resources in large-scale server systems and software routers.

References

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

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] P. Goyal, X. Guo, and H. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating System," in *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, USENIX, Berkeley, CA, Oct. 1996, pp. 107–121.

[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. 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

[15] 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.

[16] 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.

[17] M. Shreedhar and G. Varghese, "Efficient Fair Queueing Using Deficit Round-Robin," in *Proceedings of ACM SIGCOMM '95*, 4(3), Sept. 1995, pp. 231–242.

[18] A. Silberschatz and P. Galvin, *Operating System Concepts*. Reading, MA, USA: Addison-Wesley, 5th ed., 1998.

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

[20] I. Stoica, H. Abdel-Wahab, and K. Jeffay, "On the Duality between Resource Reservation and Proportional Share Resource Allocation," in *Multimedia Computing and Networking Proceedings, SPIE Proceedings Series*, 3020, Feb. 1997, pp. 207–214.

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

[22] 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.

[23] 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.