Charge-Based Proportional Scheduling

Umesh Maheshwari

Technical Memo MIT/LCS/TM-529
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139
January, 1995

Abstract

Most priority-based schedulers lack the ability to control the relative execution rates of applications. A recent scheme, called *lottery scheduling* [WW94], uses randomization to control the execution rates of threads in proportion to the tickets allocated to them. However, randomization does not afford sufficient control over short periods of time; *e.g.*, it would fail to provide the intended execution rates for threads that run for less than 10 timeslices. This paper presents a new scheme that controls execution rates over much smaller intervals and provides better service guarantees. Simulation results prove its advantage over lottery scheduling. The scheme is based on charging threads for CPU usage and occasionally skipping some threads to keep the usage close to the intended proportion. Unlike earlier charge-based schemes, which adjust the priorities of running threads, this scheme schedules threads in round-robin order. Despite its improved quality of service, the scheduler processing overhead is low.
Keywords:
scheduling, resource allocation, proportional share

1 Introduction

Most thread schedulers are based on some variant of *priority* scheduling. Such schedulers provide crude control over the relative execution of application threads. As long as a higher priority thread is runnable, the lower priority threads are ignored. Most systems try to solve this problem by adding usage-charging techniques, which reduce the priorities of threads with processor usage. While this allows some degree of fairness, it does not provide control over the relative execution rates of applications. The effect of usage-charging varies wildly with the values of parameters chosen, and depends unpredictably on the particular runtime situation. For instance, if the charge parameter is not set correctly, two high priority threads can keep alternating and starve a lower priority thread.

Proportional scheduling has been proposed for predictable control over execution rates [WW94]. Here, threads are allocated tickets in order that runnable threads will execute at rates proportional to their tickets. Proportional execution rates are useful in situations where services of varying importance must proceed concurrently. This is the case, for example, in a server handling multiple clients, a multi-media application processing audio and video, or a multi-threaded scientific computation where accuracy is governed by the amount of processing.

The implementation of proportional scheduling in [WW94] is called *lottery* scheduling. It uses randomization to select threads in proportion to the tickets allocated to them. However, randomization does not afford sufficient control on the execution rates over short periods of time such as 10 timeslices. As a result, threads that live for a short time, as are common in graphical user interfaces, may fail to receive the intended execution rates. Further, randomization provides poor service guarantees: even if a newly runnable thread has a large number of shares, it may not be scheduled for an unbounded time, albeit the probability of this happening decreases with time. This makes lottery scheduling unsuitable for handling high-priority tasks, which priority scheduling handles quite well.

I present a new implementation of proportional scheduling that is based on charging threads for processor usage. Threads are considered for scheduling in round-robin order, and some are skipped to keep their execution rates close to the intended proportion. The scheme is different from usage-charging schemes added on top of priority scheduling, which adjust the priorities of running threads and schedule the thread with the highest priority. The fair share scheduler proposed in [KL88] also works through adjusting priorities, and therefore suffers from unpredictable execution rates. Actually, this work is focused on providing fairness on a per-user basis over longer periods of time, not on controlling the execution rates of individual threads.

Charge-based proportional scheduling is capable of providing proportional execution rates over periods as short as 10 timeslices. The deterministic nature of the scheme provides significantly strong service guarantees and handles high-priority threads just as well priority scheduling. Despite these advantages, the scheme has very little processing overhead — less than lottery scheduling in most cases.

This paper focuses on efficient and fine-grained control over execution rates of individual threads. The simplicity of the scheme lends itself to the addition of techniques proposed for other aspects of a full-fledged scheduler. For example, many of the supplementary techniques proposed for lottery scheduling, such as modular decomposition, are also applicable to the proposed scheme.

The paper is organized as follows. Section 2 introduces the charge-based algorithm in a simple setting, and Section 3 discusses issues such as creation and deletion of threads in a realistic situation. Section 4 gives simulation results to prove the advantage of the charge-based scheme. Section 5 compares the scheme with earlier schemes, especially lottery scheduling.

2 Charge-Based Scheduling

This section describes the basic algorithm behind the proposed scheme. It deals with scheduling a given set of runnable threads to achieve some desirable ratio of execution rates. Issues such as creation or blocking of threads are discussed in Section 3.

Each thread is allocated some *shares*, with the intention that runnable threads will execute at rates proportional to their shares. (Shares are like tickets in lottery scheduling.) The scheduler does not change the allocation of shares by itself; it manipulates another value associated with each thread: the thread's *account*. The scheduler initializes the account of a thread with the thread's shares. It schedules timeslices, or *quanta*, to runnable threads in round-robin order. Each time a thread receives a quantum, the scheduler deducts a charge from its account. During the round-robin scheduling, any thread with a non-positive account is skipped. When there are no threads left with positive account, the scheduler refunds the accounts of all runnable threads with the number of shares they have. The concept of a refund is similar to "aging" of the usage charge in priority scheduling [LMKQ89], except that a refund is targeted to achieve a more predictable effect. The algorithm is depicted in Figure 1. Setting the appropriate level of charge is crucial for the efficacy of the algorithm and is discussed in Section 2.1.

The result of the above scheduling is that, over a sufficiently long period of time, threads receive quanta in ratio of their shares. For example, consider two threads, S with 3 shares and T with 2. The scheduling of these threads is shown below. Here, the notation $[T_a]$ means that T's account was refunded up to a; T_a means that T received a quantum and its account dropped down to a; (T_a) means that T was skipped, so its account remained at a. Assuming a charge of 1 per quantum, T and S are scheduled as follows:

$$[S_3\,T_2]\,S_2\,T_1\,S_1\,T_0\,S_0\,(T_0)\,[S_3\,T_2]\,S_2\,T_1\,S_1\,T_0\,S_0(\,T_0)\,\ldots$$

```
While true do
    found = false
    For each runnable thread T do
        If T.account > 0 then
            Run T for a quantum
            found = true
            T.account = T.account - charge
        end
    end
    If not found then
        For each runnable thread T do
            T.account = T.account + T.share
        end
    end
end
```

Figure 1: Charge-based proportional scheduling.

which results in the sequence, $STSTS, STSTS, \ldots$. Here, the threads are refunded after every 5 quanta, of which S receives 3 and T receives 2.

Analysis

I define the execution rate of a thread over a period as the ratio of the quanta allocated to the thread and the total quanta scheduled in that period. Given the shares allocated to various threads, s_i, the intended execution rate of a thread with s shares is $s/\Sigma s_i$. Over any given period of time, the actual execution rate may deviate from the intended rate. In both lottery scheduling and charge-based scheduling, the actual execution rates get closer to the intended rates over longer periods of time. The smaller the period required to limit the deviation in the rates, the better the scheme.

In this paper, I measure time periods in quanta because it factors out the differences in the absolute time interval used for a quantum on different systems. I refer to a pass made by the scheduler over the set of runnable threads as a *round*. In Figure 1, a round is one iteration of the `while` loop. In any round, each thread is either given one quantum or skipped. The period between two refunds is a *term*. In the example where S has 3 shares and T has 2 and the charge is 1, a term involves 3 rounds, with T skipped in the third. In general, if the charge, *c*, is greater than 1, the accounts of some threads may be negative at the end of a term (ranging from $-(c-1)$ to 0). Therefore, the actual execution rates over a term may differ from the intended rates. The duration of a term can vary between $\Sigma\lceil\frac{s_i}{c}\rceil$ and $\Sigma\lfloor\frac{s_i}{c}\rfloor$ depending on the values of the accounts at the beginning of the term.

Consider a period of time such that the accounts of all threads at the end of the period are identical to their values at the beginning. I refer to such a period as a *cycle*. (A special case is when the accounts are all zero at the beginning as well as at the end.) By definition, the refund granted to a thread during a cycle must be equal to the charge applied to it. Since the refund is proportional to the allocated shares and the charge is proportional to the number of quanta received, during each cycle, the threads receive quanta in exact proportion to their shares. Thus, the actual execution rates over the duration of a cycle are exactly equal to the intended rates.

In the example with 3 and 2 shares, each term constitutes a cycle, but this need not be true in general. Given the shares allocated to various threads, s_i, and the charge, c, the length of the cycle in quanta can be computed analytically, although the computation may be non-trivial. In the simple case when the charge is 1, a period of Σs_i quanta constitutes a cycle.

2.1 How much to Charge

Applying a charge of 1 works fine only as long as the shares held by runnable threads are small integers. The problem is obvious from the following example, where S has 30 shares and T has 20:

$$[S_{30}\,T_{20}]\,S_{29}\,T_{19}\;\ldots\;S_{10}\,T_0\,S_9\,(T_0)\;\ldots\;S_0\,(T_0)$$

which results in the sequence, $STSTSTSTSTSTSTSTSTSTSTSSSSSSSSSS, \ldots$. Here, for the first 20 rounds, both S and T are scheduled once per round. For the next 10 rounds, only S is scheduled. Thus, the period of time over which S and T receive proportional execution rates (the cycle) is as long as 30 quanta. If S terminates in only 10 quanta, it would not live to see its advantage over T.

I have considered two ways to compute the appropriate level of charge in the general case. While the first is more intuitive, the second results in more uniform scheduling as well as a simpler scheduling algorithm. Analysis and simulation results show that even the first performs better than lottery scheduling.

2.1.1 Fixed-Term Charging

This technique aims to provide approximately proportional execution rates over short periods by constraining the term size. It fixes a target term size and computes a charge such that most terms are about that long. Given a target term size, k, a suitable value for the charge is $\lceil\frac{\Sigma s_i}{k}\rceil$. (All computation proposed in this paper is integral.) Here, k can be chosen to be a power of 2 for quick integer division.

With this charge applied for each quantum, all threads will have non-positive accounts in about k quanta. The exact term size may differ from k depending on the rounding-off of the charge, the divisibility of the individual share values by the charge, and the number of threads. First, consider a simple example where the term size is indeed k. If S has 30 shares and T has 20, and the term size is 10, then the charge should be $\frac{(30+20)}{10}$, or 5. The scheduling happens as follows:

$$[S_{30}\,T_{20}]\,S_{25}\,T_{15}\,S_{20}\,T_{10}\,S_{15}\,T_5\,S_{10}\,T_0\,S_5\,(T_0)\,S_0\,(T_0)$$

which results in the sequence, $STSTSTSTSS, \ldots$. In the above example the accounts of S and T at the end of the term are zero, so the term constitutes a cycle. In general, the accounts of threads whose share values are not divisible by the charge will be negative at the end of the term. Although such terms do not provide exactly proportional rates, they come close in practice. Furthermore, the execution rates over a larger period of time indeed converge to the allocated shares. This happens because negative accounts at the end of one term affect scheduling during the next terms in a self-corrective manner. As an example, consider thread S with 7 shares and thread T with 4, and an intended term size of 6. The desired charge is $\frac{(7+4)}{6}$ or about 2. The first term is scheduled as follows:

$$[S_7\,T_4]\,S_5\,T_2\,S_3\,T_0\,S_1\,(T_0)\,S_{-1}\,(T_0)$$

In this term, S received 4 quantum and T received 2. The next term is scheduled as follows:

$$[S_6\,T_4]\,S_4\,T_2\,S_2\,(T_0)\,S_0\,(T_0)$$

Note that the refund added 7 to S's account of -1. In this term, S received 3 quantum and T received 2. Thus, the quanta received in the two terms combined are in proportion to the allocated shares. This is to be expected since the two terms form a cycle. In addition, the execution rates within each term is close to the intended rate.

There is a tradeoff involved in choosing the target term size, k. A small k results in shorter terms, which is desirable for fine-grained control over execution rates, but results in larger deviations from the intended rates over the term. Simulation results in Section 4 show that setting k to 10 results in deviation of less than 0.05 in execution rates over a 10 quanta period.

2.1.2 Maximum-Share Charging

Here, the charge is set so that the threads are refunded after every round. That is, each round acts as a term. To exhaust all accounts within a round, the charge is set to the maximum share of any thread. Since all threads are refunded after each round, the account of the thread with the maximum shares remains constant. Therefore, this thread is run in each round. The accounts of other threads may remain negative even after a refund; such a thread is not run until successive refunds increase its account above zero. If S has 30 shares and T has 20, the charge is 30 and the scheduling happens as follows:

$$[S_{30}\,T_{20}]\,S_0\,T_{-10}\,[S_{30}\,T_{10}]\,S_0\,T_{-20}\,[S_{30}\,T_0]\,S_0\,(T_0)\;\ldots$$

which results in the sequence, $STSTS$, ... This automatically gives the smallest possible cycle size of 5 and is better than the fixed-term technique. Even when the cycle size in the two techniques is the same, this one results in a more uniform distribution of quanta, resulting in less deviation from the intended rates over sub-cycle periods. As an example, consider the scheduling of S with 5 shares and T with 2. Using maximum-share, the charge is 5:

$$[S_5\, T_2]\, S_0\, T_{-3}\, [S_5\, T_{-1}]\, S_0\, (T_{-1})\, [S_5\, T_1]\, S_0\, T_{-4}\, [S_5\, T_{-2}]$$

$$S_0\, (T_{-2})\, [S_5\, T_0]\, S_0\, (T_0)\, \ldots$$

resulting in the sequence $STSSTSS$, $STSSTSS$, ... On the other hand, using a target term-size of 7 (or 10), the charge would be 1:

$$[S_5\, T_2]\, S_4\, T_1\, S_3\, (T_0)\, S_2\, (T_0)\, S_1\, (T_0)\, \ldots$$

resulting in the sequence: $STSTSSS$, $STSTSSS$, ... Here, T's quanta are clumped together into the the first 2 rounds.

Analysis

Perhaps surprisingly, the maximum-share algorithm given above is similar to Bresenham's digitized line-drawing algorithm, which is used heavily in computer graphics [Bre65]. The line-drawing algorithm plots pixels in a 2-D grid so as to approximate a line between two points in the grid. Suppose the line is to be drawn from the point $(0, 0)$ to (X, Y). The algorithm works by maintaining variables x and y for the last point plotted. If $X > Y$, the algorithm increments x at each step, and decides whether or not to increment y depending on their current values. Figure 2 shows the result of drawing a line from $(0, 0)$ to $(10, 4)$. The algorithm is effective: it limits the deviation of pixels from the intended line due to quantization error. It is also very efficient: it involves only integer addition and subtraction.

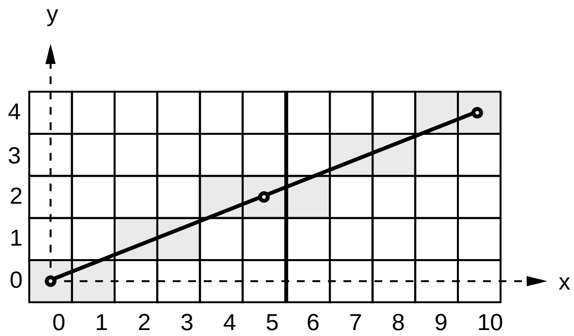

Figure 2: Digitized line drawing.

Just as the line-drawing algorithm approximates a line by plotting pixels, proportional scheduling approximates the intended execution rates by allocating quanta, except that the latter may involve more than 2 variables (threads). The counterparts of the constants X and Y are the allocated shares, and those of the variables x and y are the accounts. The counterpart of the test $(X > Y)$ is the selection of the maximum share. Like the line-drawing algorithm, maximum-share charge-based scheduling is both effective in limiting deviation and efficient in limiting processing overhead.

Performance

The following analyzes the processing overhead of the scheduler. The goal is to find the amount of work done per quanta scheduled. Suppose there are n threads and thread T_i has s_i shares. Let the maximum shares be s_m. Then, roughly speaking, in s_m consecutive rounds, thread T_i is scheduled s_i times. Thus, total quanta scheduled in s_m rounds are Σs_i. The work done during these rounds is $n * s_m$ units, where each unit of work involves reading an account, adding or subtracting to it, and writing it back. The work done per quantum allocated is $n * s_m/\Sigma s_i$, or s_m/s_a, where s_a is the average shares held by the threads. Note that this value would be a small constant for most distributions of shares; for example, it is 2 for a uniform distribution of shares. In the worst case, when the shares are highly skewed, it is upper-bounded by n. The overhead in real time is expected to be low because the unit of work considered above has very little overhead.

3 Dynamic Changes

The previous section focused on scheduling a fixed set of runnable threads according to the shares allocated to them. This section considers issues regarding addition and removal of runnable threads. Since maximum-share charging is more effective and simpler than fixed-term charging, this section concentrates on only that technique.

3.1 Creation/Termination of Threads

There are several options to handle new runnable threads. Probably the simplest is to incorporate the new thread right away so that it gets to run as part of the ongoing round (and therefore, in the ongoing term). A potential problem with this approach is that a continual supply of low-share threads can keep a round going for ever. For instance, one low-share thread could create another low-share thread and then terminate, while keeping the number of threads at any time roughly the same. This can starve higher-share threads. Further, it is unclear as to when the charge should be updated to reflect the new thread: immediately, or when the next term starts. Neither alternative is perfect, although the choice is not expected to have substantial impact on performance.

Another option is to wait until the ongoing term is over before incorporating the new thread into the run queue. The wait can be implemented by simply initializing the account of the new thread to zero, so that the thread is skipped until the next refund. This technique is more suitable for maximum-share charging because there the terms are short (one round each), so the wait is shorter. Even so, it is undesirable for a new high-share thread to have to wait until the round is over.

It is possible to get the better of the above options with the following approach:

1. If the new thread has more shares than any other runnable thread, it is scheduled immediately in the ongoing round. The charge is updated and the difference (if any) applied to the threads that have already run during the current round.

2. Otherwise, the new thread waits until the ongoing term has finished. (Its account is initialized to zero.) The charge is updated when the next term begins.

The above approach incorporates a desirable element of priority-based scheduling: often, a high-priority thread must be run immediately in response to an external event. The desirable effect can be achieved by allocating a large number of shares to such a thread, so that the above approach will ensure that the thread is run immediately. It may seem at first that this approach might cause a high-share thread to wait for lower-share threads if another, higher-share, thread is runnable. Actually, this is unlikely because the higher-share thread must have resulted in a high charge, so that the accounts of the low-share threads are expected to be negative for most rounds (terms) and such threads will be skipped.

The removal of a runnable thread is simple. The charge is updated at the end of the ongoing term. In maximum-share charging, after the thread with maximum shares is removed, the accounts of the remaining threads may need to be refunded multiple times (say, m) before any of them becomes positive. This procedure can be expedited by computing m using integer division and then applying m times the normal refund at once.

3.2 Blocking/Unblocking of Threads

A simple way to handle blocking and unblocking of threads would be to treat them exactly as termination and creation of threads. Under this approach, when a thread blocks, its account is forgotten and is reinitialized when the thread unblocks. However, this can distort execution rates away from the intended proportions if some threads block and unblock often. For example, a low-share thread that blocks and unblocks after every round (term) will get to run in every round.

Therefore, a better approach is to retain the account of a blocked thread. Unlike newly created threads, there is no danger of a continual sequence of unblocking threads; therefore, an unblocked thread can be incorporated into the ongoing round. As in the case of new threads, when a maximum-share thread unblocks, it is scheduled immediately.

A pertinent issue is the amount of refund to grant a thread when it unblocks. If a thread unblocks in the same term as it blocked in, it is not granted any refund. Otherwise, the thread is granted a single refund. (A more fancy scheme can be imagined that grants refund in proportion to the terms for which the thread blocked, while ensuring that the resultant account of the thread is not more than its shares.) This approach requires a mechanism to check for the condition when a thread blocks and unblocks in the same term, which can be implemented using a term counter.

The charge is updated as in the case of creation and deletion of threads. When a thread blocks, the charge is updated when the next term begins. When a thread unblocks, the charge is updated immediately and the difference (if any) is applied retroactively to threads that have already run in the ongoing round.

3.3 Fractional Quanta

If a thread blocks or yields the CPU after using only a fraction of a quantum, it may receive less than its proportional share of the CPU time. Charge-based schemes can easily fix this by prorating the charge applied according to the fraction of the quantum actually used by the thread.

One implication of prorating the charge is that maximum-share charging might not actually exhaust the accounts of some threads in one round. If a thread with a positive account at the end is still runnable (say, because it yielded instead of blocking), it will be scheduled again before the next refund, as desired.

Prorating of the charge can be implemented without resorting to floating point numbers. A fixed number of lower order bits, b, in the charge and the account variables can be used to track the fractional usage. Each quanta is divided into 2^b *subquanta*. The charge to be applied is then computed based on the per-quantum charge and the actual number of subquanta used.

4 Simulation

This section contains simulation results for lottery, fixed-term charge-based, and maximum-share charge-based schedulers. The goal is to assess the proportionality of execution rates over small periods.

4.1 Model and Implementation

The model simulates a simplistic scenario where a number of threads with different shares are introduced at the beginning and allowed to run without blocking. The model also ignores the processing overhead due to the scheduler and the clock interrupt handler in accounting for time. However, it is apparent from the charge-based algorithm that it involves only a few integer operations and loads and stores for each runnable thread considered in round-robin order. Therefore, I believe that ignoring the overhead of charge-based scheduling does not distort the execution rates in real time.

Each scheduler is simulated as a function: the input is the shares allocated to various threads, and the output is a sequence identifying the threads that should be scheduled in successive quanta.

I found that lottery scheduling was sensitive to the quality of the random number generator used, especially when dealing with small number of tickets, such as two threads with 1 and 2 tickets. Poor random number generators resulted in perceptibly worse deviations in execution rates. I used the Park-Miller pseudo-random number generator [PM88], which is fast and yet of high quality and is the one used in [WW94].

For fixed-term charging, the target term size was fixed at 10. The choice of the target term size favors certain combinations of shares. For example, a term size of 10 is well suited for 20-30 shares, because then the charge is 5, which exactly divides both 20 and 30. I have avoided such favored combinations in the results presented.

4.2 Measurables

I have used the following variables to present the output from the experiments. Note that time is measured in quanta.

The *cumulative quanta*, q_t, received by a thread at any time t is simply the number of quanta scheduled to the thread by that time.

The *execution rate*, $e_{p,t}$, of a thread over a period p and at time t is the ratio of the quanta received by the thread over a period p starting at time t to the total number of quanta in that period.

$$e_{p,t} = (q_{t+p-1} - q_{t-1})/p$$

The intended execution rate, $\overline{e}$, of a thread is the ratio of the shares held by the thread to the total number of shares held by runnable threads.

$$\overline{e} = s/\Sigma s_i$$

The *mean absolute deviation* of the execution rate, d_p, over a period p is the expected deviation of the execution rate over a period p from the intended rate.

$$d_p = E(|e_{p,t} - \overline{e}|) = \sum_{t=1}^{N} |e_{p,t} - \overline{e}|/N$$

The value of N used in the experiments was 1000.

4.3 Results

The following results were obtained from scheduling two threads with 50 and 20 shares. Figures 3-5 plot the cumulative quanta received (q_t) by the two threads against total quanta scheduled (t). The dotted lines represent the intended allocation. By and large, all of the three schemes seem to adhere to the intended allocation. The max-share charge-based scheme follows the intended lines the best, which is not surprising, given its similarity with the line-drawing algorithm.

Figures 6-8 plot the execution rates over a period of 10 quanta ($e_{10,t}$) of the two threads against time (t). It is apparent that lottery scheduling can result in quite erratic execution rates when they are measured over 10 quanta. For example, at times the execution rate of the second thread dropped to zero (it was not scheduled for more than 10 quanta). On the other hand, there are times when the execution rate of the second thread was more than that of the first. Since the outcome of lottery scheduling is not deterministic, it must be noted here that the result shown is only typical: results in other runs were sometimes worse and sometimes better. Both of the charge-based schemes seem to be distinctly better than lottery scheduling. Further, the maximum-share scheme outperforms the fixed-term scheme.

Figure 9 provides the definitive comparison between the various schemes. It plots the deviation in the execution rates (d_p) against the period (p) over which the execution rates are measured. When only two threads are scheduled, the absolute deviations in the execution rates for the two are equal by definition. Hence, only one curve is shown per scheme. Note that the mean deviation for lottery scheduling is largely free of a chance factor (unlike execution rates) because it is computed by averaging deviations over 1000 periods.

As expected, the deviations decrease as the period over which the execution rates are measured is increased, modulo some fluctuations in the case of the charge-based schemes due to the periodicity of their quantization errors. The mean deviation in the execution rates for lottery scheduling over 10 quanta is about 0.11, while that for maximum-share charging is 0.025. Again, maximum-share charging has lower deviations than fixed-term charging. The deviation for maximum-share charging drops to zero at every multiple of 7 because the scheme effectively achieves the smallest possible cycle size, which is 7.

As an example of scheduling of more than two threads, Figures 10 and 11 show the cumulative quanta received by four threads with shares 47, 31, 23, and 11.

5 Related Work

The fair-share scheduler proposed in [KL88] is based on usage charging but it works through adjusting priorities and running the process of highest priority. Processes with higher priority get charged more for same execution time. This work is focused on providing fairness on a per-user basis over longer periods of time, not on controlling the execution rates of individual threads.

At the time this paper was originally written (January 1995), the only existing work aimed at proportional schedul-

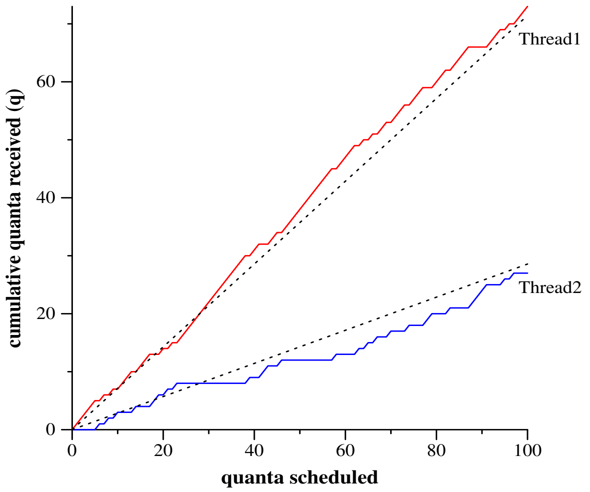

Figure 3: Lottery scheduling (50:20)

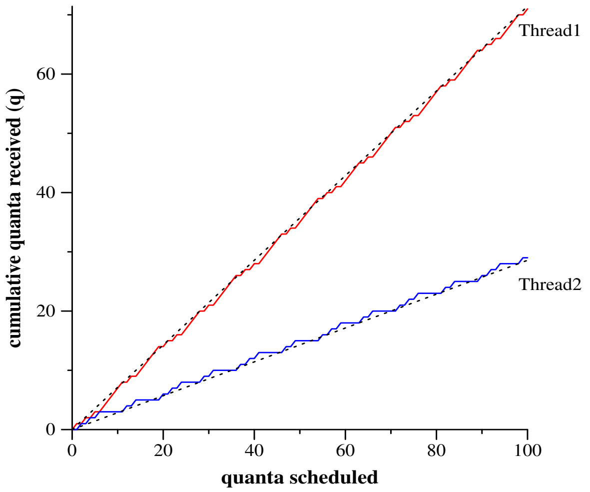

Figure 4: Fixed-term charging (50:20)

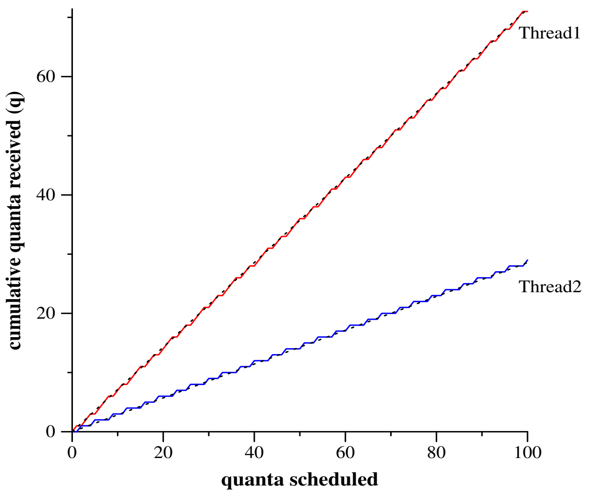

Figure 5: Maximum-share charging (50:20)

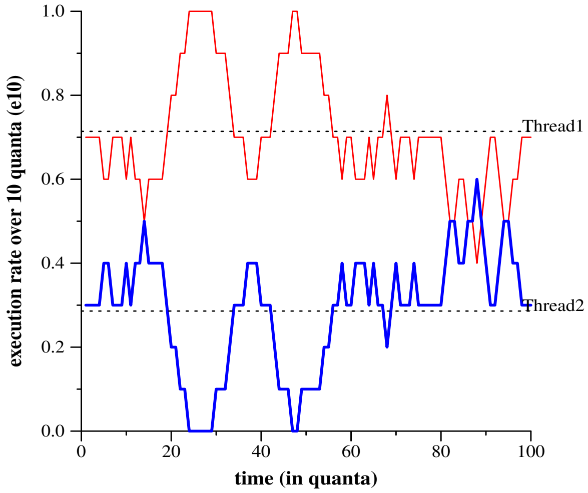

Figure 6: Lottery scheduling (50:20)

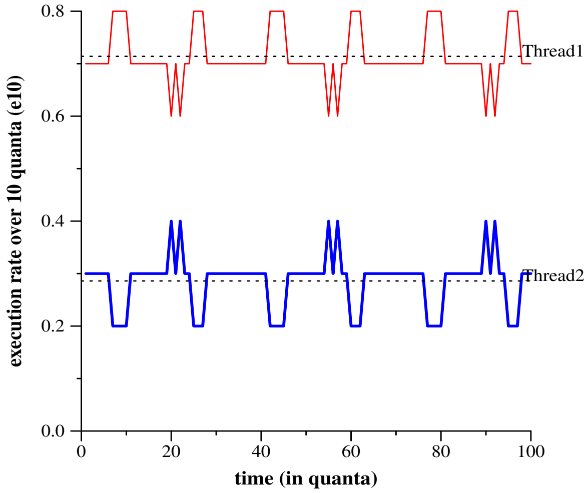

Figure 7: Fixed-term charging (50:20)

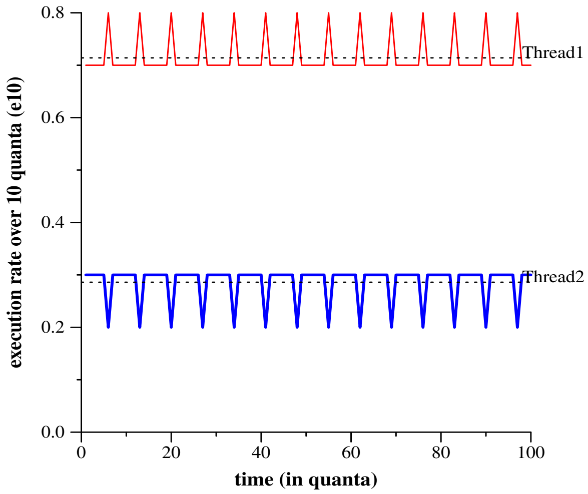

Figure 8: Maximum-share charging (50:20)

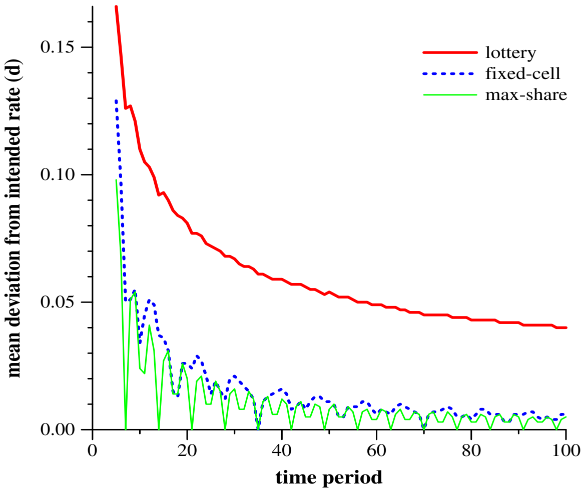

Figure 9: Deviation from intended rates (50:20)

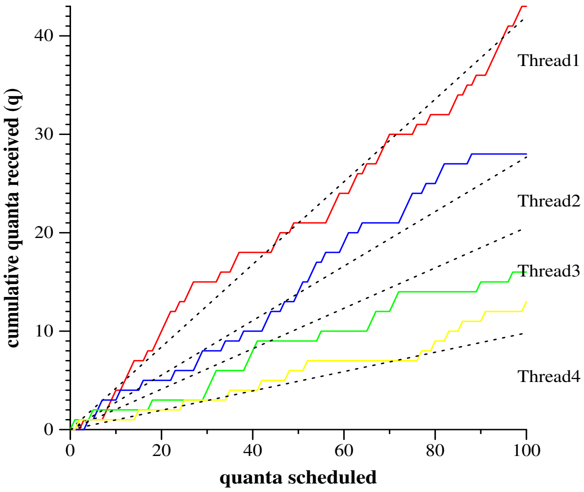

Figure 10: Lottery scheduling (47:31:23:11)

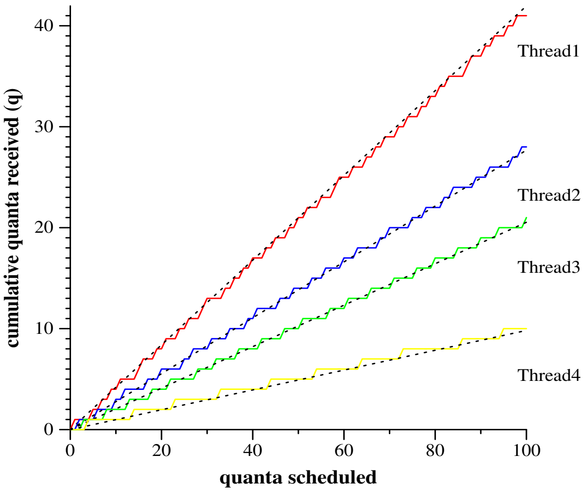

Figure 11: Maximum-share charging (47:31:23:11)

ing over small intervals was lottery scheduling. Since then three other deterministic schemes have appeared [FS95, SAW95, WW95]. A comparison between them and charge-based scheduling is found in [WW95].

Comparison with Lottery Scheduling

The charge-based scheme provides proportional execution rates over smaller time periods than provided by lottery scheduling. This is evident from simulation results, which show that the charge-based scheme achieves lower deviations over 10 quanta than what lottery scheduling achieves over 100 quanta. The advantage of the charge-based scheme is intuitive because it corrects for skews in the past allocation of quanta to move closer to the desired allocation, while lottery scheduling relies on Bernoulli trials, which are independent of past decisions.

The deterministic nature of charge-based scheduling provides significantly stronger service guarantees. In lottery scheduling, there is a finite chance that even a thread with a large number of shares may not run for a while. Consider the situation when there are 8 threads with 10 shares each, and a thread with 20 shares is created. The expected number of quanta for which the 20-share thread will have to wait before it gets its first quantum is $(\Sigma s_i/s) - 1$, or 4. There is a 10.7% probability that the 20-share thread will have to wait for 10 or more quanta. In maximum-share charge-based scheduling, the maximum-share thread runs in every round. Even other threads are guaranteed to run every so often depending on their shares. When a new maximum-share thread starts up, it is guaranteed to run immediately.

Lottery scheduling requires generating a random number (modulo Σs_i) and then checking for the winner by running down the list of runnable threads. A binary search tree can be used to decrease the search time from linear to logarithmic in the number of threads, but I believe that it would have a higher constant overhead. In charge-based scheduling, for each quantum scheduled, threads are considered in round-robin order until a thread with a positive account is found. As discussed in Section 2.1.2, the average amount of work done is $n * s_m/\Sigma s_i$, where each unit involves reading an account, adding or subtracting to it, and writing it back. Without experimental results it is difficult to predict with certainty which scheme would result in lower overhead, and whether the difference is significant. I expect the charge-based scheme to be better because it avoids random number generation.

Fortunately, many of the supplementary techniques proposed for lottery scheduling in [WW94] are also applicable to the charge-based scheme presented in this paper. Note that the charge-based scheme only replaces the randomized-selection aspect of lottery scheduling with a deterministic accounting method. For example, the charge-based scheme admits the same kind of modular decomposition as lottery scheduling. As is true of tickets in lottery scheduling, the shares used in this scheme can be associated with a currency, and each currency backed by shares in some more primitive currency. Similarly, priority inversion problems can be avoided by a transfer of shares.

One advantage of lottery scheduling over the charge-based scheme is that it does not require any special treatment for changes to the set of runnable threads, and is therefore simpler in this aspect.

6 Conclusions

I have presented a scheduling algorithm that controls the relative execution rates of threads in proportion to the shares allocated to them. The algorithm is based on charging threads for CPU usage, but unlike previous usage-charging schemes that schedule the highest priority thread, it schedules threads in round-robin order with selective skipping. Further, I proposed two competitive methods for setting the charge that should be applied for each quantum of processor usage. In particular, setting the charge to the maximum number of shares held by any runnable thread results in very low deviation in the execution rates.

Simulation results have shown that the proposed algorithm is effective: it keeps the execution rates close to the intended proportions over periods as short as 10 quanta. At the same time, the algorithm is efficient: the threads are considered in round-robin order and the scheduling decision requires only a few integer operations.

While proportional scheduling has distinct advantages over priority scheduling, previous implementations such as lottery scheduling had some drawbacks that made it unlikely for them to replace priority scheduling. I believe that this work pushes proportional scheduling a step forward as a better alternative than priority scheduling for general-purpose computing.

References

[Bre65] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[FS95] L. L. Fong and M. S. Squillante. Time-functions: A general approach to controllable resource management. Working draft, IBM Research Division, T. J. Watson Research Center, March 1995.

[KL88] J. Kay and P. Lauder. A fair share scheduler. *CACM*, 31(1):44–55, January 1988.

[LMKQ89] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, 1989.

[PM88] S. K. Park and K. W. Miller. Random Number Generators: Good ones are hard to find. *CACM*, October 1988.

[SAW95] I. Stoica and H. Abdel-Wahab. A new approach to proportional share resource allocation. Technical Report 95-05, Department of Computer Science, Old Dominion University, Norfolk, VA, April 1995.

[WW94] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First OSDI*, November 1994.

[WW95] Carl A. Waldspurger and William E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Memo MIT/LCS/TM-528, MIT Laboratory for Computer Science, Cambridge, MA, June 1995.