Comparing Lottery and EEVDF Scheduling Algorithm for Real-time Applications

Enamul Hoque, Tanima Dey
Department of Computer Science
School of Engineering and Applied Sciences
University of Virginia
Charlottesville, VA 22904
eh6p@virginia.edu, td8h@virginia.edu

ABSTRACT
In this paper, we study two proportional-share resource management algorithms, Lottery Scheduling and Earliest Eligible Virtual Deadline First (EEVDF), and compare their performances in scheduling real time jobs having specific deadlines. Lottery scheduling is not used to schedule real time jobs; we test whether we can use it for this purpose by designing the ticket allocation policy according to real-time constraints. On the other hand, EEVDF is designed to support real time jobs. But there is no quantitative evaluation of EEVDF for scheduling real time jobs in the literature, so we evaluate its performance for real time jobs. We evaluate the performance of these two algorithms by implementing them in Java and simulating with synthetic jobs having real time deadlines and compare the performance of the lottery scheduling with that of EEVDF. From the results of the evaluations, we find that lottery scheduling performs competitively with EEVDF scheduling when the real time jobs have large periodic deadlines.

1. INTRODUCTION
Real time applications are a major portion of today’s operating system’s workload. The main characteristic of these applications is that the computation involved in each process must be completed within a specific deadline. Example of such workload includes, multimedia applications, file transfers, Internet Telephony etc.

In general scheduling algorithms are of two types, preemptive and non-preemptive. Preemptive scheduling algorithms allocate a resource to a particular process for a fixed amount of time called quantum. If the process needs the resource beyond that time, then it has to wait till the scheduler allocates it the resource again. In non-preemptive or cooperative scheduling, the process is allowed to use the resource as long as it needs once it is scheduled to use the resource.

There are mainly two types of real time applications, hard and soft. In the hard real time applications, not only the correctness of the operations matters but the application must finish its computation within the specified amount of time. If it finishes after that period, it is said to have missed its deadline and the computation done in that time is considered to be useless. Example of hard real time systems includes the anti-lock brakes in automobiles where it must work within a specific time to stop the vehicle. On the other hand, the soft real time applications might miss deadlines occasionally and provide decreased quality of service. Example of such applications includes streaming videos in which if the application misses certain number of deadlines, it might not be noticeable to human eyes as long as it does not create any flickers.

Another important aspect in real time systems is to design scheduling algorithms for both periodic and aperiodic jobs. The scheduling algorithm for such systems should ensure that the periodic jobs can meet the deadline and provide fast response to the aperiodic jobs so that its performance is not hampered for scheduling periodic jobs only.

Static real time scheduling algorithms fix the priority of the jobs before they are added to the ready queue, Such as, rate monotonic scheduling [6]. On the other hand, dynamic real time scheduling algorithms decide the priority or the order in which the jobs will be scheduled, on the fly, by considering their deadlines or periods, such as, EDF [8].

In this paper, we have compared the performance of two proportional-share resource management algorithms, lottery scheduling (LS) [14] and earliest eligible virtual deadline first (EEVDF) scheduling [12]. We are considering periodic real time jobs each with specific deadline and computation time as workloads to be scheduled by these algorithms. Lottery scheduling provides randomized mechanism ensuring probabilistic fairness among all the competing clients for a resource in the proportion of their shares. On the other hand, EEVDF ensures such fairness deterministically with the added feature of providing strong timeliness or deadline guarantees for the service time received by a client. In our case, the clients are the real time jobs.

In both lottery and EEVDF scheduling algorithm, there is a set of clients that compete for a time shared resource like processor, I/O etc. The contended resource is allocated to the scheduled job for time quantum of fixed size. At the beginning of each time quantum, a client is selected by lottery (LS) or based on a determin
istic parameter lag time (EEVDF). EEVDF guarantees that the difference between the time a client should get according to its share and the time it actually gets is bounded by the size of a quantum. But lottery scheduling does not provide any such guarantees because it is not designed for real time applications or jobs. The hypothesis we want to test is whether we can use lottery scheduling for real time jobs.

We design the ticket allocation algorithm of lottery scheduling in such a way that it helps real time jobs to maintain deadlines. The tickets are allocated to each job dynamically proportional to how close its deadline is and then the scheduler can perform the lottery to get the winner, i.e. which job gets the resource for the next quantum. But as the lottery scheduling algorithm is randomized, it is not guaranteed that all jobs will make their deadlines. On the contrary, EEVDF is designed to perform better for such periodic jobs. We compare the performance of the lottery scheduling with that of EEVDF and find that lottery scheduling with our dynamic ticket allocation scheme performs competitively with EEVDF scheduling when the jobs have large periodic deadlines. We compare the performance of the lottery scheduling with dynamic ticket allocation algorithm with that with static ticket allocation and the results show that the dynamic allocation policy helps the real time jobs to satisfy their deadlines better. We also evaluate EEVDF scheduling for real time jobs which is missing in the original paper [12].

The paper is organized as follows: Section 2 describes the related work for real time scheduling algorithms. Section 3 describes lottery and EEVDF algorithm in details. Section 4 describes the major parts and related components of the project. The description of the experiments, the final results, comparisons of the algorithms’ performances and the analysis of the findings are done in Section 5. We summarize our results and draw conclusion in Section 6.

2. RELATED WORK

In this section, we discuss some of the most popular real time scheduling algorithms and some works done on lottery and EEVDF scheduling. There are numerous works done in this particular research area and still there are ample opportunities.

The two most common algorithms are Rate Monotonic (RM) [6] and Earliest Deadline First (EDF) [8]. RM uses a fixed priority scheduling in which priority is assigned based on the cycle duration of the job. The shorter the cycle duration is, the higher is the job’s priority. However, the assumptions made by RM are not practical, because it considers the jobs to have constant run-time [6] [11].

On the other hand, in EDF scheduling, the processes having the closer deadline are assigned higher priorities than the other processes. Whenever a scheduling event occur, the job with the highest priority, i.e., having the closest deadline, will be searched in the ready queue to schedule it so that it can meet the real time requirements. EDF guarantees that all deadlines are met provided that the total CPU utilization is not more than 100%. And it is a dynamic scheduling algorithm [8].

This is also known to be included in general class of deadline algorithms, which are optimal scheduling algorithms and schedules an active job, at each instant of time t, whose deadline is closest to t [5].

Another real time scheduling algorithm is Least Slack Time (LST), also known as Least Laxity First (LLF) algorithm which is based on slack time [1]. Slack time is defined to be the amount of time left after the job is started now. LST assigns priority based on the slack time, highest priority for least slack time. This algorithm can be suboptimal and can be used for processor utilization up to 100% like EDF.

In [10], the authors describe two scheduling algorithms for scheduling a set of jobs characterized by worst-case computation times, deadlines, and resource requirements on a multiprocessor system. The authors of [7] have described a new algorithm for supporting both periodic and aperiodic jobs in hard real time computing systems where the deadlines of periodic jobs are guaranteed by a deterministic feasibility check. A different approach to hard real time scheduling has been taken in [4] by the authors, where they have designed a distance-constrained job model and its scheduling scheme which is applicable to real-time periodic job scheduling and communication.

The authors of lottery scheduling later proposed another proportional-share resource management algorithm, called Stride Scheduling [16], based upon virtual time measurement that enables the scheduler to determine the process to be selected next. Comparison between these two proportional share algorithms has been made in [15]. These two papers do not discuss whether these algorithms are applicable for scheduling real time jobs or not. Detail descriptions and theoretical analysis of EEVDF are provided in [12], [13] and [3]. But none of these provide performance evaluation of EEVDF for real time jobs.

3. LOTTERY AND EEVDF SCHEDULING

In this section, we describe the lottery and EEVDF scheduling algorithms briefly.

3.1 Lottery Scheduling

The traditional schedulers in operating systems do not consider the different relative execution rates of the applications and usually follow the priority scheduling to ensure that the high prioritized applications do not suffer because of the low priority ones. But this approach causes the low priority jobs to degrade performance. Other approaches, such as, fair share, microeconomic scheduler, etc. address successfully some of the problems with the absolute priority schedulers and thus ensure fairness, but they are very complex and have high overheads. These are the motivations for designing lottery scheduling. It is a novel approach taken by the authors to solve this problem of having jobs of different execution rates so that these are assigned the resource accordingly.

Lottery scheduling is a randomized resource allocation mechanism which provides flexibility and responsive control over different service rates of large number.
of applications. It efficiently implements the proportional-share resource management, the resource consumption rates of active computations are proportional to the relative shares that they are allocated. It ensures probabilistic fairness and removes the problem of starvation which is very much common for priority based scheduling. It also supports modular resource management so that modules running concurrently can isolate their resource management policies from one another. It is such simple and general that it is applicable to a wide range of diverse resources, such, CPU processing time, I/O bandwidth, memory, access to semaphore locks etc. for scheduling.

In lottery scheduling, the resource rights are provided to the appropriate client in terms of lottery tickets. The allocation of a particular resource will be provided to the client first by holding the lottery in which the winning ticket number will be determined. Then the client holding the winning ticket will get to use that resource for until next lottery is held, i.e. the allocated time quantum expires. The lottery tickets are relative and vary in proportion to the contention for that particular resource. The allocation of the resources, i.e., expected allocation of the resources to the competing clients is proportional to the number of tickets they hold. The number of lotteries, won by a client has binomial distribution and required for a client’s first win has geometric distribution [14].

3.2 Earliest Eligible Virtual Deadline First Scheduling

Earliest Eligible Virtual Deadline First (EEVDF) is a scheduling algorithm that can satisfy both proportional-share and real time constraints. EEVDF retains all the advantages of the proportional-share schedulers and also provides strong timeliness guarantees for the computation time received by a job. To use it as a proportional-share algorithm, we have to assign shares to each job from the higher level. In that case, EEVDF guarantees that the difference between the computation times that a job should receive based on its share and the computation time it actually receives in the real system is bounded by a specific value. To use EEVDF to schedule jobs with real-time constraints, we have to assign shares to each job inversely proportional to its deadline. These two features make EEVDF appropriate for scheduling continuous media, interactive and batch applications.

Suppose, there are a total of \( n \) jobs where the \( i \)th job has period \( p_i \), deadline \( d_i \) (here we assume \( p_i = d_i \)), arrival time \( a_i \) and computation time \( c_i \), where \( i = 1, 2, 3, \ldots, n \). At time \( t \), we calculate the weight \( w_i \) and share \( f_i \) of the \( i \)th job by Equation (1) and Equation (2) respectively. At time \( t \), the algorithm calculates virtual time \( V(t) \) by Equation (3), where \( t_{\text{start}} \) is the time when the scheduler starts running.

\[
w_i = \frac{c_i}{a_i + d_i - t} \tag{1}
\]

\[
f_i = \frac{w_i}{\sum_{i=1}^{n} w_i} \tag{2}
\]

\[
V(t) = \frac{t - t_{\text{start-time}}}{\sum_{i=1}^{n} \frac{a_i}{w_i}} \tag{3}
\]

The basic idea of the algorithm is to associate with each job an eligible time, \( e_i \). A job becomes eligible to start at time \( t \) when the computation time that it should receive equals the computation time it has actually received before issuing the current request. For scheduling the jobs, EEVDF uses the Virtual Eligible Time, \( V(e) \) and Virtual Deadline, \( V(d) \) of each job which are defined by Equation (4) and Equation (5) respectively. Here \( a_i \) represents the computation time that the \( i \)th job has obtained up to time \( t \).

\[
V(e_i) = V(a_i) + \frac{a_i}{f_i} \tag{4}
\]

\[
V(d_i) = V(e_i) + \frac{c_i}{f_i} \tag{5}
\]

EEVDF always schedules a job that has the smallest virtual deadline. So it gives jobs having closer deadlines, earlier arrival times and more remaining computation times more priority to get scheduled. The pseudo code of the algorithm is presented in Figure 1.

4. DESCRIPTION OF THE PROJECT

In this section, we describe the major components of the project.

4.1 Ticket Allocation Algorithm for Lottery Scheduling

The lottery scheduling algorithm provides proportional-share resource management. The abstraction of clients in the original lottery scheduling is considered as jobs in our case. In order to adapt lottery scheduling to be applicable to real time applications, we have to ensure that the allocation of tickets to a job is properly done so that the individual instances of the real time applications, i.e. jobs, can meet the periodic deadlines.

![Figure 1: The EEVDF Scheduling Algorithm](image-url)
We design the ticket allocation policy of lottery scheduling to suit real-time applications. First, we implement the lottery scheduling algorithm with static ticket allocation policy. Jobs are assigned tickets inversely proportional to their deadlines when they arrive and the number of tickets do not change afterwards. Then, we modify the ticket allocation policy to allocate tickets to jobs dynamically. We define the laxity of a job to be the difference between the finish time (arrivalTime + deadline) of the job and the current time of the system. The modification is that the number of tickets allocated to a job at a particular time is proportional to how near the deadline of the job is at that moment. That is, we assign the tickets inversely proportional to the laxity. The number of allocated tickets will be high when the laxity is small and vice versa. In both cases, after the tickets have been assigned, the lottery is held and the winning ticket is determined. Then the job holding the winning ticket is selected for having the resource. In our case, the resource is the CPU time.

The lottery scheduling algorithm with dynamic ticket allocation policy is given in Figure 2. The allocation of the tickets to each job is dynamically adjusted. The scheduler component of our implementation wakes up every time after the current time quantum expires and runs the algorithm described above to allocate the tickets to each job and continue scheduling.

### 4.2 Data Structures

There are two main data structures that we use to implement the two algorithms: Job and Ready Queue. The Job data structure has a number of fields including the job id, period, finish time and computation time. The finish time is calculated by adding the period to the arrival time of the job to the system. There is a timer associated with each job whose value is equal to the period of the job. The Ready Queue is used to hold all the jobs which have arrived in real time and waiting to get scheduled. When the scheduler wakes up it determines the next job to schedule according to the appropriate algorithm (either lottery or EEVDF) and removes the scheduled job from Ready Queue. When the timer for the associated job expires, a new instance of the job is created and added to the Ready Queue.

### 4.3 Random Number Generator

We use Park-Miller pseudo-random number generator [9] which is also known as Lehmer Random number generator. It is a multiplicative linear congruential generator and the general form is given by Equation (6).

\[
S_{n+1} = A \times S_n \mod (2^{31} - 1)
\]

Here, \(S_{n+1}\) is the next seed and \(S_n\) is the current seed for the random number generator. The initial seed \(S_0\) should be relatively prime to \((2^{31} - 1)\) and the value of \(A = 16807\) for the minimal standard. In our case, \(A = 48271\).

The random number generator returns a pseudo-random real number uniformly distributed between \(0.0\) and \(1.0\). The period is \((m - 1)\), where \(m = 2, 147, 483, 647\) and the smallest and largest possible values are \((1/m)\) and \(1 - (1/m)\) respectively. We generate 10, 100 and 1000 random numbers using this random number generator, within range from 0 to 100 and the frequency distribution of the random numbers is shown in Figure 3. We see from the graph that as the number of samples increases, the distribution becomes more uniform. In our simulation, we generate 300 and 150 random numbers in a single run of the simulation for quantum size = 1 second and quantum size = 2 seconds respectively, whose distribution is shown in Figure 4.

### 5. EXPERIMENTS AND ANALYSIS OF THE RESULTS

We evaluate and compare the algorithms by simulation. We implement the simulator in Java programming language. The simulator takes as input a set of periodic jobs and schedules them according to the lottery scheduling or EEVDF algorithm. We run the simulator by varying quantum size, number of jobs, periods and computation times of the jobs.

The metric that we are considering for measuring the relative performance of the algorithms is percentage of
missed deadlines. We maintain an array which counts the number of deadline misses, indexed by the job’s id. Whenever the scheduler wakes up, if it finds that the difference between the finish time of the job and the current time of the system has become less than or equal to zero, then it increments the number of deadline misses by one. Also, if it finds the summation of current time and quantum size to exceed the finish time of the job, it increases the count of deadline misses. The second case implies that, even if the job gets selected by the scheduler to get the next CPU cycle, still it cannot finish its job within the allocated time, therefore missing its deadline. In both of the cases, the scheduler removes such jobs (if there are multiple such jobs) from the ready queue and then takes its scheduling decision. First we evaluate the algorithm using a small set of periodic jobs. Next we run large scale simulations.

5.1 Small Scale Simulation

For small scale simulation, we create six jobs each having deadline and computation time as shown in the Table 1. For each algorithm, we run the simulation with this workload 3 times and take the maximum deadline miss ratio. We select this workload, because it has some jobs with computation times equal to quantum size (job 1 and 2 in case of quantum size = 2 seconds) and some jobs with computation times greater than quantum size, so they need more than one quanta to finish computation. Also the workload contains jobs with different periods. In our simulation, we always assume period of a job is equal to the deadline and finish time to be equal to the sum of arrival time and period of the job.

First we compare the performance between lottery scheduling having static ticket allocation policy and having dynamic ticket allocation policy. For the workload of Table 1, the performance comparison of these two is shown in Figure 5 for quantum size = 1 second and Figure 6 for quantum size = 2 seconds, allowing pre-emption. From the graphs, we see that in general, the periodic jobs miss more deadlines for the static policy than the dynamic policy. For both quantum size = 1 second and quantum size = 2 seconds, jobs with period = 10 seconds miss deadlines much less for the dynamic policy than the other because we allocate the tickets based on laxity. Since the jobs with period = 10 seconds have harder deadlines, these jobs get more tickets with more time elapsed and the probability of their winning the lottery increases, so does their chance of getting scheduled. We infer from the results of less deadline misses for lottery scheduling with dynamic ticket allocation policy that the ticket allocation scheme based on laxity works successfully. Since the static ticket allocation policy does not change number of their tickets based on laxity, so these jobs miss more deadlines than the dynamic policy. For the jobs with period = 20 seconds and period = 30 seconds, the percentage deadline misses is almost the same for both static and dynamic policy.

The result of comparison between static and dynamic lottery scheduling algorithm allowing non-pre-emption is shown in Figure 7. From the figure, we see that the percentage deadline misses is much smaller for the dynamic policy than the static for jobs with period = 10 seconds. The jobs with period = 20 seconds and period = 30 seconds, do not miss their deadlines at all for the lottery scheduler with dynamic ticket allocation. From these results, we infer that to use lottery scheduling for real time jobs, we have to allocate tickets dynamically based on the laxity of the jobs.
Next we compare the performance between lottery scheduling with static and dynamic ticket allocation policy and EEVDF scheduling for the small workload which is shown in Figure 8 for quantum size = 1 second and Figure 9 for quantum size = 2 seconds. From the figures we see that for the dynamic lottery scheduler, the jobs have much higher deadline miss percentage for all the periods. This is because of the randomness of the lottery scheduler to select the winning job to get the contended resource. EEVDF does not have any randomness in the selection of job to be scheduled. We can see some deadline misses in case of EEVDF as well. These misses are mainly due to the fact that not all of these jobs can be scheduled, the arrival rate is greater than the service rate in this case.

Next we compare the performance between the dynamic lottery and EEVDF scheduling for the small workload, shown in Figure 10, the case of non-preemption. From the figure we come to the same conclusion as in the previous case of preemptive scheduler. In general, we infer from the graphs that the dynamic lottery scheduler works competitively with EEVDF scheduler for jobs having large periods.

### 5.2 Large Scale Simulation

For large scale simulation, we generate a set of synthetic job sets with varying periods and computation times. We follow similar mechanisms as mentioned in [2] to generate the job sets. That is, we generate period and computation time of each job in such a way that the total utilization is less than 1. Formally, if we need to generate a set of $n$ periodic jobs, then we choose period $p_i$ and computation time $c_i$ for the $i$th job according to the constraints of Equation (7). The utilizations of all the jobs have same distribution. We vary the number of jobs in each job set from 10 to 60. For each job set, we run the simulation for 5 minutes. We do not consider job sets with more than 60 periodic jobs, because
we can generalize their performance for more than 60 jobs from the results we have. The metric we use to compare their performance is the ratio of total number of jobs that miss their deadlines and total number of jobs that arrive during the simulation period. For each task set, we run the algorithms three times and take the maximum deadline miss ratio.

First we run the simulation for preemptive scheduling with quantum size = 1 second. The performance of both the algorithms is shown in Figure 11. From this figure, we see that overall deadline miss ratio increases for both the algorithms with the increase in total number of jobs. This is because the computation time of some jobs is less than the quantum size. For these jobs, the processor remains idle for the remaining time of a quantum and the scheduler does not wake up to schedule the next job. With increase in number of jobs, this processor idle time keeps increasing and as a result more jobs miss their deadlines. For all cases, EEVDF performs better than the lottery scheduling.

Then we run the simulation for non-preemptive scheduling. The performance of both the algorithms is shown in Figure 12. From this figure, we see that the overall percentage deadline misses for EEVDF remains constant (2%) for all the sets of jobs. Actually this constant value should be considered as an upper bound, because we take ceiling of the percentage deadline miss ratio. So EEVDF performs very well for our generated task sets. This is because the total utilization of the jobs is less than 1. So there should not be too many misses. In case of dynamic lottery scheduling also the deadline miss ratio remains close to a constant level, although there are ups and downs which we believe is due to the randomized nature of the algorithm. Here also EEVDF
always performs better than the lottery scheduling with dynamic ticket allocation policy.

6. CONCLUSION

In summary, we show that the lottery scheduling which is a proportional-share algorithm can be designed to support real time applications. We propose one heuristic of allocating more tickets to the jobs in proportion to the closeness of the deadlines. We evaluate the lottery scheduling and EEVDF scheduling and find that lottery scheduling with dynamic ticket allocation policy works better than the static ticket allocation lottery for such periodic real time jobs and competitively with EEVDF algorithm for jobs with large periods. As there is always some randomness involved in the lottery scheduling algorithm, it can never guarantee successful scheduling for hard real time jobs. But our results show that by allocating tickets based on deadlines, lottery scheduling can be used to schedule soft real time jobs, depending on the appropriate quantum size and computation time. The proper model of such lottery scheduling for soft real time jobs requires further studies and experiments.

Our evaluation of EEVDF shows that it always maintains overall deadline miss ratio of all the jobs less than 2 percent in case where the overall utilization of the tasks are less than 1. It will be interesting to see how EEVDF performs in case of jobs with varying distribution of utilizations. Overall, we hope by using our simulator, it is possible to test the performance of these two algorithms exhaustively and by analyzing the results, we can come to more precise conclusion about their performances.

Acknowledgment

We would like to express gratitude towards our CS656 course instructor Prof. Marty Humphrey for helping us to find the right problem statement for the class project and directing us to the right path to finish the project successfully.

7. REFERENCES

Implementation, pages 1–12, November 1994.
