A Comparison of Scheduling Algorithms for Multiprocessors

Markus Peloquin et al.

December 13, 2010

Abstract

Scheduling multiple processors in a computer system while providing real-time guarantees remains an interesting and challenging problem, despite being a topic of study for decades. Throughout these years of investigation, many algorithms have been proposed, analyzed, and evaluated. In this study, we intend to compare and evaluate three of these multiprocessor scheduling algorithms: the classic, simple algorithm global EDF, the optimal P-fair algorithm PF, and a newer, optimal algorithm LLREF that intends to improve on existing P-fair algorithms and global EDF variants. We empirically investigate the ability of global EDF to schedule randomly generated task sets as compared to PF and LLREF, which can schedule any task set with a total utilization less than the number of processors. We also investigate the overhead of all three algorithms in terms of task migrations and required scheduler invocations for randomly generated tasks. Our results verify that LLREF outperforms PF. Additionally, they show that global EDF still outperforms LLREF and PF in terms of required scheduler invocations and task migrations; however, our results also demonstrate that global EDF cannot feasibly schedule the same variety of task sets as LLREF and PF.

1 Introduction

Systems with multiple processors are ubiquitous in many domains. This includes real time systems, where there are deadlines by which tasks must complete. However, most scheduling algorithms were designed for uniprocessors, and extending them to multiprocessor systems is difficult. Even scheduling algorithms which are optimal for uniprocessors, such as EDF, are not optimal when applied to multiple processors. Furthermore, there are additional sources of overhead to consider: migrating a task between processors may incur non-negligible overhead. Also, all scheduling algorithms, whether for uniprocessor or multiprocessor systems, incur overhead due to scheduler invocations.

In this paper, we will examine three scheduling algorithms for uniform multiprocessor systems. The first scheduling algorithm we will discuss is an extension of EDF, called global EDF. The second is PF, which can schedule any set of tasks as long as total utilization is less than the number of processors. The third is LLREF, which can schedule the same set of tasks as PF, but with fewer scheduler invocations. We implement these algorithms in RTSIM, an open-source simulator. Finally,

we will finish with a presentation of our simulation results and a discussion of the schedulability as well as the number of scheduler invocations and task migrations for each of these algorithms.

2 Scheduling Algorithms

2.1 Global EDF

EDF is one of the oldest and most well-known scheduling algorithms, first described by Liu and Layland [7]. The priority of each instance of a task is determined by its absolute deadline; the task with the earliest absolute deadline will always have the highest priority. EDF is work-conserving; that is, if there is any active task, the processor will execute it rather than being idle. It has been proved that EDF is optimal for periodic tasks, and that having a utilization less than or equal to 1 is a necessary and sufficient condition for EDF to be able to schedule tasks.

Global EDF is an extension of EDF for multiple processors [4]. Like PF and LLREF, it is a dynamic priority algorithm. Similar to EDF, the jobs are ordered by earliest absolute deadline, but the P highest priority processes are executed by the P processors in every time step. Scheduling events occur only when new jobs are introduced or when a job completes.

The likelihood of migrations in global EDF depends first on how often preemptions happen. Migrations can only occur due to unfavorable scheduling events. The introduction of new jobs causes only *active* $\rightarrow$ *idle* transitions. If the introduced job has an earlier deadline than an active job, it is swapped in for the lowest priority active job. Job completion causes only *idle* $\rightarrow$ *active* transitions. If a job has completed, the highest priority idle job is chosen to execute, regardless of what CPU the job may have been executing on previously. If a job's execution state has an *active* $\rightarrow$ *idle* $\rightarrow$ *active* sequence, it is possible for the job to migrate between CPUs.

The other factor in the likelihood of migrations in global EDF is the number of CPUs. The only reason a preempted task might not be migrated is if the preempting task completed before the other active jobs. Suppose completion time is uniformly random. If we consider a uniprocessor system, migration is not possible. For $P = 2$, preempted jobs are migrated with 50% probability; for $P = 10$, 90%. As CPU counts increase, the frequency of migrations made by global EDF increases.

The schedulability bound from Liu and Layland is less applicable to global EDF. If total utilization is no more than 1, it is obviously schedulable. If utilization is greater than P, it is obviously not schedulable. There has been much work on characterizing schedulability bounds for global EDF [1]. A simpler bound is

$$\sum \frac{c_i}{d_i} \leq P - \max\left\{\frac{c_i}{d_i}\right\}(P-1).$$

Take for example a set of tasks that barely fit the bound. Let $P = 4$ and $T_i = (1, 2)$ for $i \in [1, 5]$. The condition above evaluates to $\frac{5}{2} \leq \frac{5}{2}$. If any task were longer, they might not fit, as can be seen in Figure 1.

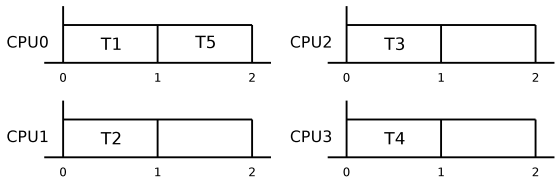


Figure 1 – Global EDF schedule of five identical tasks with utilization $\frac{1}{2}$.

2.2 PF

The PF algorithm, presented in [2, 3], is optimal for scheduling periodic tasks on multiple processors. In this regard, PF improves on global EDF because global EDF is not optimal. PF uses a "time-slicing" approach, which means it splits up tasks into unit intervals of execution, termed "slots", or ticks. PF uses this approach to approximate a "fluid scheduling", or constant rate scheduling. [6].

The basic idea of the PF algorithm is to assign slots to each task such that it is always scheduled proportionally to the utilization of the task. That is, if a task has a utilization of U, at any slot t, the task will have been scheduled for $U \cdot t$ slots between 0 and t. The authors of [2, 3] define any schedule that satisfies this requirement as *proportionately fair* or *P-fair*. They also prove that any schedule that is P-fair will meet all the deadlines for its tasks. The goal of the PF algorithm is to maintain the P-fair requirement for all tasks. One important point about the formulation of P-fairness is that the value $U \cdot t$ is not always an integer, and tasks cannot be scheduled to partial numbers of slots. A key function of the PF algorithm is to maintain whether a task has been scheduled for $\lfloor U \cdot t \rfloor$ or $\lceil U \cdot t \rceil$ slots, and make scheduling decisions according to the state of this computation.

Before describing the algorithm, it is important to state the assumptions the algorithm makes about the tasks and the properties of the resources. The first is that all tasks are independent of each other; there is no sharing of resources besides the processors. Second, the algorithm does not take into the account the overhead of migrating tasks when scheduling tasks. Finally, it assumes that the tasks are periodic and the deadlines of the tasks equal the periods.

PF maintains a measure of how close the current schedule is to meeting the P-fair requirement. The authors of [2, 3] define a *lag* function at time t for a task T_0 with utilization U_0 as follows:

$$lag(T_0, U_0, t) = U_0 \cdot t - S(T_0)$$

where $S(T_0)$ is the number of slots allocated to T_0 thus far.

PF also makes use of a "characteristic string" that describes the resource requirements for each task at each slot. The characteristic string can be calculated offline, and it is used to make optimal decisions about the future demands of tasks on each iteration of the algorithm. The authors of [2, 3] define the characteristic string for a task T_0 with utilization U_0 at time t over the characters

```
for every slot t
    allocate processors to all the urgent tasks
    allocate any remaining processors to contending tasks,
        ordered lexicographically by their characteristic strings
```

Listing 1 – The PF Algorithm

$(+, 0, -)$ as follows:

$$c(T_0, U_0, t) = sign(U_0(t+1) - \lfloor U_0 \cdot t \rfloor - 1).$$

With the lag computation and the characteristic string in hand, PF classifies each task at a time t in the following way. A task is classified as *non-urgent* if the lag for the task is strictly less than 0 and the characteristic string at the slot for t is either $-$ or 0. A task is classified as *urgent* if the lag for the task is strictly greater than 0 and the characteristic string at the slot for t is either $+$ or 0. A task that is neither non-urgent nor urgent is classified as *contending*. An invariant that falls out of this classification is that at every slot t, all urgent tasks must be scheduled and all non-urgent tasks must not be scheduled. If either of these conditions cannot be met, then the task set cannot be feasibly scheduled using PF or any other algorithm, given that PF is optimal.

After establishing this framework, the PF algorithm is quite simple and is described in Listing 1. A key nontrivial aspect of the PF algorithm is ordering the contending tasks such that the P-fairness of the schedule is maintained. This ordering is defined lexicographically according to the characteristic string of each contending task. Specifically, the ordering is $+ \geq 0 \geq -$. Additionally, when comparing the characteristic strings of two contending tasks at a particular slot t, only a certain subset of the characters must be compared. This subset is defined as the characters from slot t to the first slot when the characteristic substring has the value 0. For example, let the characteristic string of a task be "$--+0--+0$". When PF classifies this task as contending, only the sub string "$--+0$" is used to order the task relative to the other contending tasks.

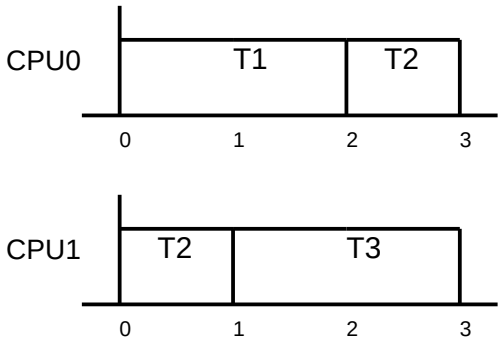


Figure 2 – Schedule for PF example

The mechanics of the PF algorithm are best explained with a simple example. The example, described in Figure 2 and Table 1, uses PF to schedule three identical tasks on two processors; each

$t = 0$		lag	character	state
x	T_1	0	$-$	contending
x	T_2	0	$-$	contending
	T_3	0	$-$	contending
$t = 1$				
x	T_1	$-1/3$	$+$	contending
	T_2	$-1/3$	$+$	contending
x	T_3	$2/3$	$+$	urgent
$t = 2$				
	T_1	$-2/3$	0	non-urgent
x	T_2	$1/3$	0	urgent
x	T_3	$1/3$	0	urgent

Table 1 – Calculations for schedule of PF example

task has a period of 3, a worst-case execution time of 2, and an initiation time of 0. Additionally, the characteristic string is the same for all three tasks, namely "$-+0-+0$...". It is important to note that this task set is infeasible with global EDF, but can be schedule by PF.

Table 1 describes the calculations and state maintained at each unit interval of time, or tick, in the algorithm. In the table, a mark is placed next to the tasks that are scheduled in that tick. Figure 2 shows the final schedule. Only ticks 0 through 2 are shown because the schedule repeats every 3 ticks.

This example illustrates an interesting case in the PF algorithm, namely when the characteristic strings of a set of contending tasks are equal. Fortunately, these ties can be broken arbitrarily. For this example, ties occur in tick 0 and tick 1. In tick 0, tasks T_1 and T_2 are chosen for execution, and in tick 1, task T_1 is scheduled over T_2. Despite breaking these ties arbitrarily, the schedule remains P-fair.

2.3 LLREF

The LLREF scheduling algorithm [5] assumes that tasks are preemptible and independent (i.e., that they have no shared resources). The cost of context switches and task migration is ignored and assumed to be negligible. The deadline is assumed to be equal to the period.

Like PF, LLREF approximates a fluid scheduling model. However, rather than using slots like PF, LLREF only schedules on a small set of events. It makes use of an abstraction called the Time and Local Execution Time Domain Plane (T-L Plane). This abstraction is used to determine when tasks must be scheduled in order to meet their deadlines. Figure 3 shows an example T-L Plane, where the x-axis is time and y-axis is the remaining execution time for a particular task.

For every two subsequent primary scheduling events, or task arrivals, triangular T-L Planes are formed where one edge is the first scheduling event, one is the diagonal of no remaining local

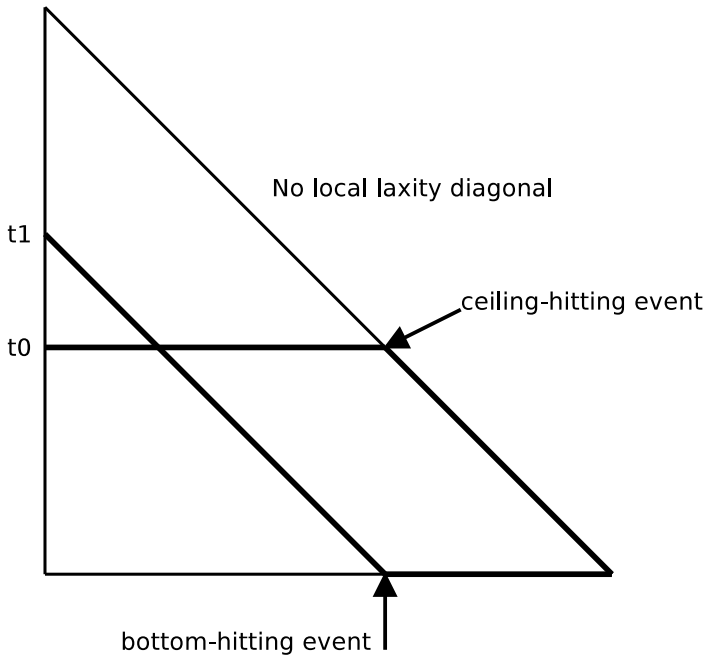


Figure 3 – T-L Plane

laxity, and the third is the horizontal at the y-value where the fluid schedule intersects the next task arrival. The two legs of the triangle are the same same length, and the line of no local laxity has a slope of -1. All T-L Planes for each time interval are superimposed, and then the algorithm is performed on each of these T-L Planes. If there is a locally feasible schedule for each plane, then all tasks can be scheduled.

When scheduling locally, only the current plane is considered. Because the borders of the T-L Plane are defined by the arrival times of tasks, a new task arrival corresponds to a primary scheduling event. On primary scheduling events, a new T-L Plane is constructed, and all tasks are rescheduled according to it.

Within each plane, tokens representing the tasks move across the plane over time, as seen in Figure 3. There are two ways that each token can move: if the task is selected, it will move to the right with a slope of -1 (e.g., t1), and if it is not selected, it will move with a slope of 0 or horizontally (e.g., t0).

Unlike PF, where scheduling occurs at each tick, in LLREF scheduling only happens within each T-L Plane at secondary scheduling events. There are two of these: when a task reaches the lower boundary of the triangle (a bottom-hitting event), and when a task hits the oblique side of the triangle (a ceiling-hitting event). A bottom-hitting event indicates that the task has executed as much as it needs to for the local T-L Plane, so it should be deselected and another task should be run instead. A ceiling-hitting event indicates that the task has zero remaining local laxity, and therefore needs to be selected immediately to meet the deadline. The tasks are otherwise selected

in order of Largest Local Remaining Execution First, hence the name of the algorithm. LLREF is not work-conserving; that is, if all tasks are at the bottom, then the processors will be idle, even if there is remaining non-local time on some tasks.

The advantage of LLREF over PF is that the scheduler does not need to be invoked as often; rather than being invoked on every time step, it only must run on primary and secondary scheduling events. It is essentially PF, but with a scheduling granularity of the interval between task arrivals rather than one tick.

2.4 The RTSIM Simulator

To evaluate and compare the algorithms, we implemented global EDF, PF, and LLREF using the RTSIM framework [8], an open-source library for simulating real-time scheduling algorithms. RTSIM builds on the simulation framework MetaSim [8], which presents a general interface to a prioritized event queue. Under RTSIM, a simulation is divided into ticks.

RTSIM provides three main abstractions: the processors managed by a *kernel* are allocated to *tasks* by a *scheduler*. For our purposes, we utilized a variant of the kernel abstraction for multiple processors and a variant of the task abstraction for modeling periodic tasks. As RTSIM users, we implemented derivations of the scheduler abstraction for all three of our scheduler algorithms.

Despite providing some useful abstractions for real-time scheduling algorithm simulation, we had to make significant modifications to RTSIM to allow us to implement PF and LLREF. Specifically, we extended the multiple processor kernel to query the scheduler on every tick for new assignments of tasks. As discussed, this behavior is required by both PF and LLREF, as they can make scheduling decisions on every tick of the simulation. We also modified the multiple processor kernel to record the various statistics we used in our evaluation, described in Section 3.1.

Additionally, our implementation of these scheduling algorithms stressed the RTSIM library in different ways, exposing a few bugs that required fixing before simulations would complete correctly. Furthermore, to be able to run our simulations for an extended period of time, we needed to fix a fair number of serious memory leaks that caused the simulation to take up enormous amounts of memory, on the order of 3 GB in 10 minutes. However, despite these issues, we performed some in-depth simulations of our algorithms, and the methodology for these simulations as well as the results are presented in the next section.

3 Results

3.1 Evaluation Methodology

We empirically evaluated global EDF, LLREF and PF by randomly generating sets of tasks and extracting three metrics from the schedules generated by each of the algorithms. The first metric evaluates the ability of the algorithm to feasibly schedule a given set of tasks. To measure schedula-

bility, we generated task sets with a utilization less than the number of processors. This guaranteed that both LLREF and PF would be able to schedule the task sets. However, since global EDF is non-optimal, there is no guarantee that global EDF would be able to schedule these task sets [2]. Whenever a task set could not be scheduled by global EDF, this infeasibility was recorded. Section 3.2 describes the algorithm used to generate these task sets.

The second metric measures the total number of task migrations made by a specific schedule for a set of tasks. The third metric measures the total number of scheduler invocations required to produce the final schedule. The task migrations and scheduler invocations characterize the overhead of each algorithm. We chose these metrics because the authors of LLREF discussed these metrics as ways to characterize the overhead of scheduling algorithms.

In our simulations, a task migration occurs when a task instance is descheduled from one processor and is later rescheduled on a different processor. For each algorithm, scheduler invocations occur at different times. For global EDF, the scheduler was invoked on every arrival of a task instance and on every completion of a task instance. At these points, global EDF makes decisions on the next task subset to schedule based on the absolute deadlines. For LLREF, there are both primary and secondary scheduling events. For PF, the scheduler is invoked every clock tick; this means the number of scheduler invocations for PF is equal to the number of clock ticks in the simulation of a specific task set.

3.2 Task Set Generation

The following algorithm in Listing 2 was used to generate the task sets. It is based on the algorithm presented by Baker [1] in an evaluation of global EDF feasibility tests. The basic idea of the algorithm is to iteratively add tasks with a random period and worst-case execution time until the utilization exceeds the total number of processors. The task that pushed the utilization over the bound is then removed to get the final task set.

However, this approach does not necessarily result in task sets with reasonably small hyperperiods, and we would have liked to simulate all task sets for a single hyperperiod. To make some task sets reasonable to simulate, the algorithm takes two approaches. The first is to remove tasks from the task set until the hyperperiod is less than 100000 simulation ticks or the number of tasks is less than one more than the number of processors. The second approach limits the number of attempts to construct a reasonable task set. If a task set is not found within these limited number of tries, a task set with a feasible utilization under PF and LLREF is accepted, but it is only simulated for 100000. This limitation was introduced to make our simulations practical and allow us to gather a large amount of data.

The `genPeriod` and `genUtil` procedures in the algorithm of Listing 2 introduce the random variation into task set generation, and draw pseudo-random numbers from the distributions used by Baker [1]. Specifically, `genPeriod` draws an integer uniformly between 1 and 1000.

```
maxRunTime = 100000
maxTries = 100000
tasksets = { }
for i = 0 to iterations
    for procs in [ 2, 4, 8 ]
        for dist in [ Uniform, Bimodal, ExpMean25, ExpMean50 ]
            taskset = { }; attempts = 0; done = false
            while( !done ) {
                totalUtilization = 0.0
                while( totalUtilization < procs ) {
                    period = genPeriod()
                    util = genUtil(dist)
                    wcet = ceil(period*util)
                    totalUtilization += wcet / period
                    taskset.add(createTask(wcet, period))
                }
                "remove the last task added"
                if( attempts >= maxTries
                    && taskset.size() >= procs + 1 ) {
                    tasksets.add(taskset)
                    break
                }
                while( lcm(taskset) > maxRunTime
                       && taskset.size() > procs + 1 ) {
                    "remove the last task added"
                }
                if( taskset.size() < procs + 1 ) {
                    attempts++; continue;
                }
                done = true
```

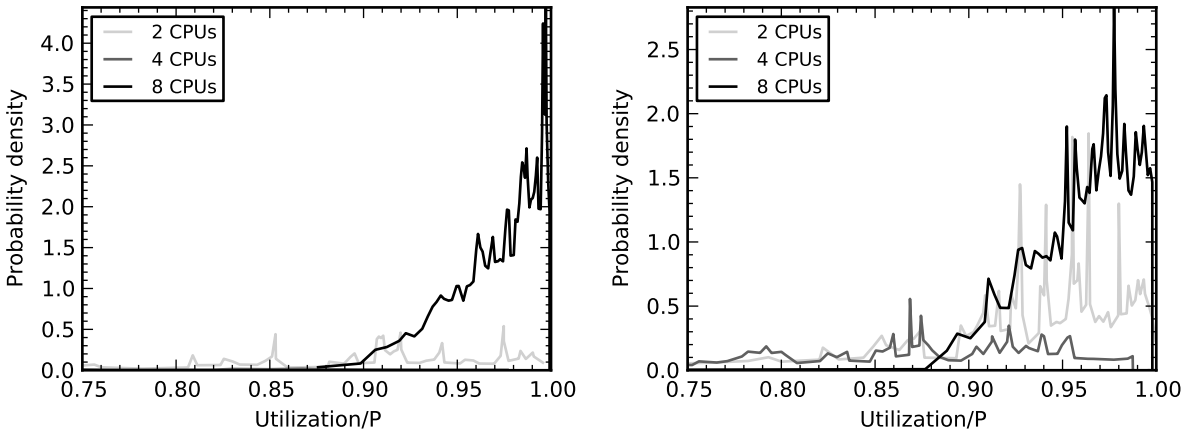
Listing 2 – Algorithm for generating random task sets

`genUtil` is more complicated and depends on the current iteration of the *dist* loop. For the `Uniform` distribution, the utilization is draw uniformly between $1/period$ and 0.999. The `Bimodal` distribution is created from two uniform distributions: one between $1/period$ and 0.5 and the other between 0.5 and 0.999. The utilization is chosen from the second distribution with probability 1/9. Also, if $1/period$ is greater than 0.5, a new lower bound is selected from a uniform distribution between 0.001 and 0.3. The `ExpMean25` and `ExpMean50` are exponential distributions with means of 0.25 and 0.50 respectively. For these distributions, when the drawn utilization is greater than 0.999, the utilization is clipped to 0.999. Finally, it should be noted that we decided to give all tasks an initialization time of zero to simplify the analysis of our results.

We generated 1100 task sets for each combination of these parameters, resulting in 13200 task sets. We input each of these task sets into RTSIM and invoked each of the three algorithms to create schedules for these tasks.

3.3 Schedulability

Schedulability was checked by running the tests until they failed and just recording whether they were feasible or not. We did not pay attention to such details as how long it took for a job to miss or with what frequency misses would occur.



(a) Exponential distribution, $\mu = .5$

(b) Uniform distribution

Figure 4 – Measured probability density of the infeasible task groups on EDF. The graphs were normalized for system utilization. The Exponential distribution was characteristic of those not shown.

Since PF and LLREF are optimal, there is no sense checking their schedulability. In fact, our simulator was always able to successfully schedule the tasks with PF and LLREF when their combined utilization was less than P, which was expected.

Global EDF had some unsurprising, but critical, problems. As shown in Table 2, its probabilistic guarantees are quite low. In a couple instances (with exponential distributions), task sets with less than 50% utilization could still miss deadlines.

CPUs	Worst miss	μ	σ	Misses
2	1.28	1.82	.161	15%
4	2.27	3.44	.352	14%
8	7.01	7.67	.232	96%

Table 2 – EDF feasibility statistics for a uniform distribution of utilizations. All numbers, save *misses*, are in terms of utilization. *Worst miss* is the lowest utilization that missed. μ and σ are the mean and standard deviation of utilizations that caused misses. This data shows the same trends as the other distributions.

The number of misses also drastically changes as the number of CPUs increases. This is intuitively what one would expect since with $P = 1$, there would have been no misses with full utilization. However, note that schedulability gets slightly better from two to four CPUs.

The degraded performance with increased CPUs is again shown in Figure 4. As the data in the table suggests, scheduling gets harder as P increases. There is also an inversion between two and four CPUs in both graphs; the four CPU curve is not even visible in Figure 4a.

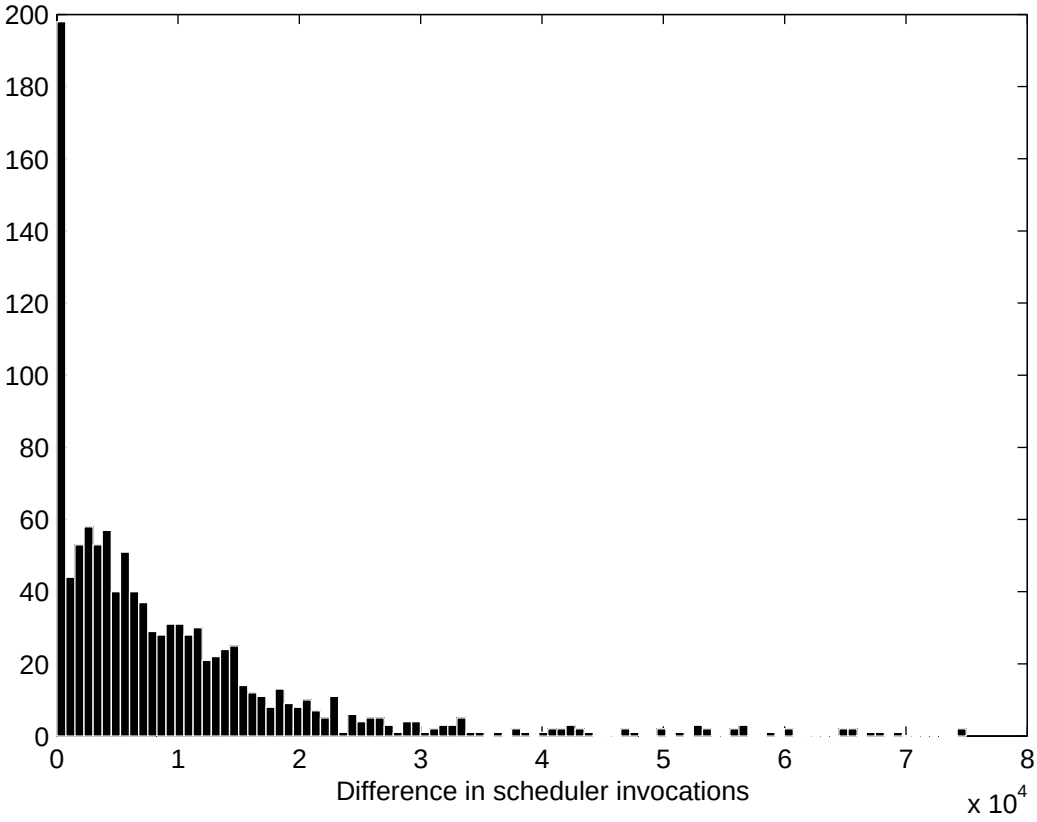


Figure 5 – Histogram of the difference in number of scheduler invocations between EDF and LLREF on 4 processors

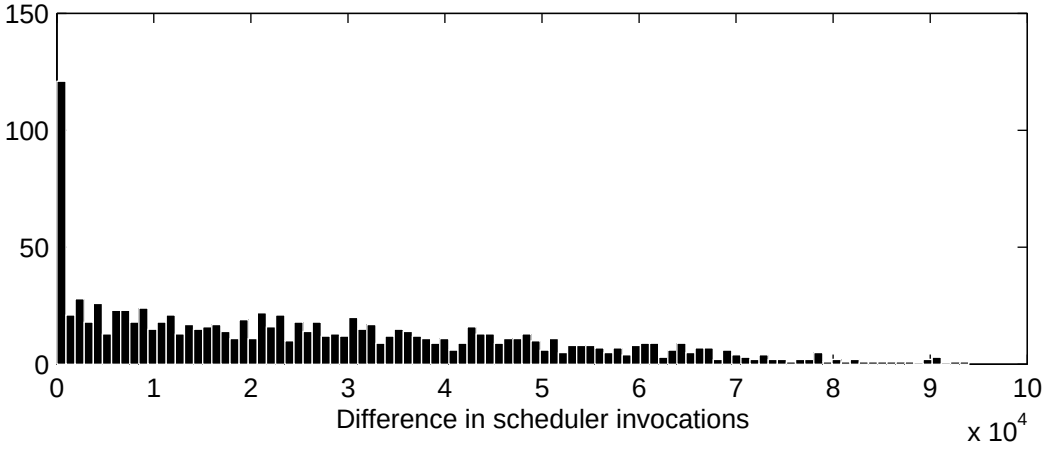


Figure 6 – Histogram of difference in number of scheduler invocations between PF and LLREF on 4 processors

3.4 Scheduler Invocations

As can be seen in Figures 5 and 6, LLREF always has at least as many invocations as EDF, and PF always has at least as many invocations as LLREF. Since they are ordered, we do not show the comparison between EDF and PF.

For brevity, we show only the comparison of scheduler invocations for 4 processors; the results for 2 and 8 processors are similar. The main difference is that the spike at 0 is higher, indicating that the number of times where the performance of EDF is equal to that of LLREF, and those where LLREF is equal to PF, increases with number of processors. Additionally, the observed trends did not differ significantly with the different distributions used for generating worst-case execution times.

3.5 Task Migrations

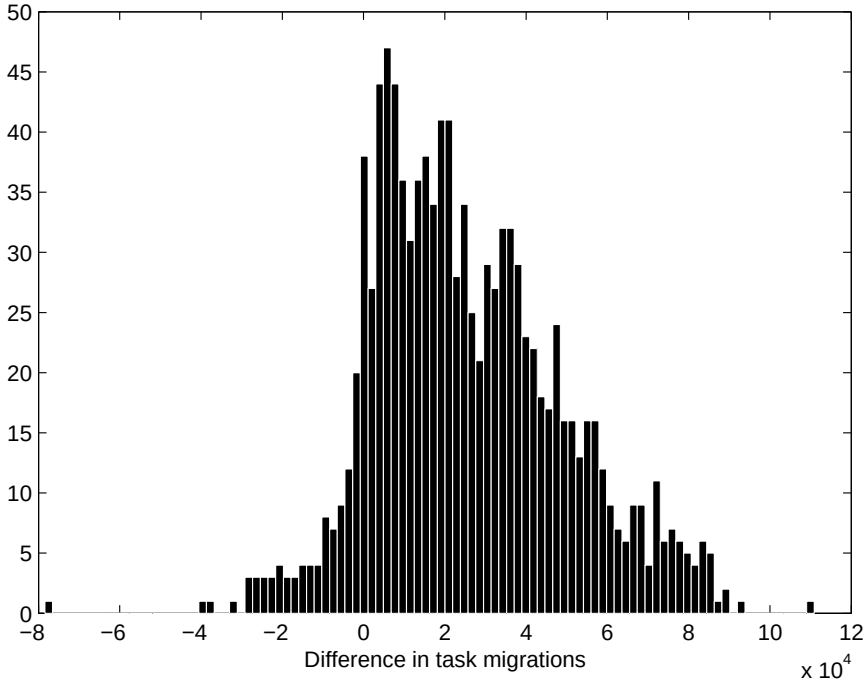


Figure 7 – Histogram of the difference in number of task migrations between PF and LLREF on 4 processors

Similar to our observations for scheduler invocations, the difference in task migrations between two algorithms did not change with the different distributions for worst-case execution time. However, our observations also did not change for different numbers of processors. Therefore, we only present the schedules that were created for 4 processors with worst-case execution times drawn from the `Uniform` distribution.

The graphs in Figure 7, 8, and 9 present histograms for the difference in the number of task migrations between two of the three algorithms. The difference in number of task migrations between LLREF and PF is presented in Figure 7. It is important to note in this plot that when the difference is less than 0, PF required fewer task migrations than LLREF, and when the difference is greater than 0, LLREF required fewer task migrations than PF. Examining Figure 7, we observe that the schedules produced by LLREF almost always require less task migrations than PF; in only about 30 schedules did PF outperform LLREF in terms of task migrations. We also observe that for a significant number of the schedules, LLREF required far fewer migrations than PF.

In Figure 8, the difference between PF and EDF in terms of number of task migrations for each schedule is displayed. It is important to note that no histogram bin lies to the left of zero in this figure, which implies that the number of task migrations required by PF is always greater than the number of task migrations required by EDF. We also observe that EDF almost always required significantly fewer migrations than PF.

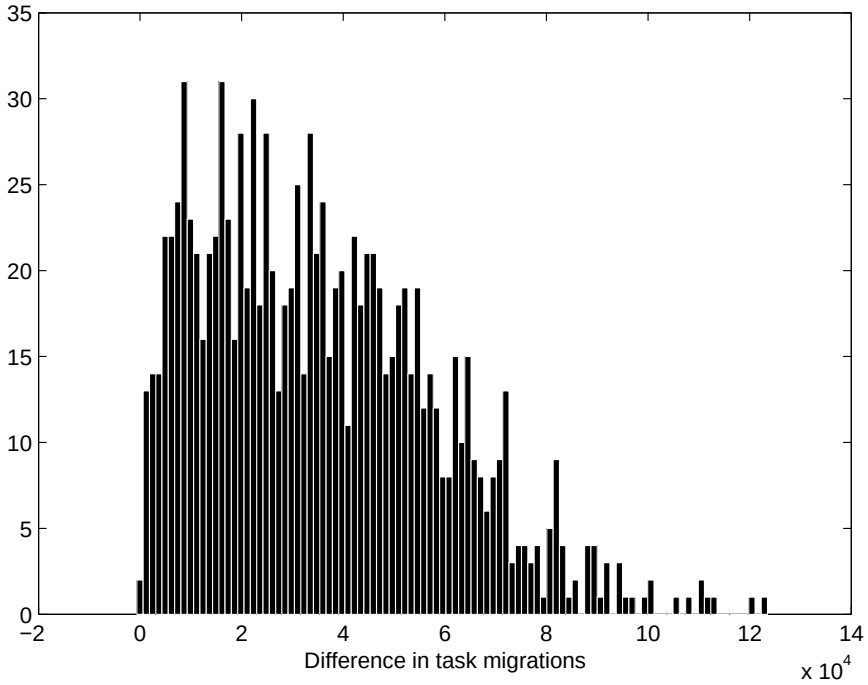


Figure 8 – Histogram of the difference in number of task migrations between PF and EDF on 4 processors

Figure 9 presents the difference in number of task migrations between LLREF and EDF. When the difference is less than 0, LLREF required fewer task migrations than EDF. When the difference is greater than 0, LLREF required more migrations than EDF. We observe that for most schedules LLREF required a similar amount of task migrations when compared to EDF.

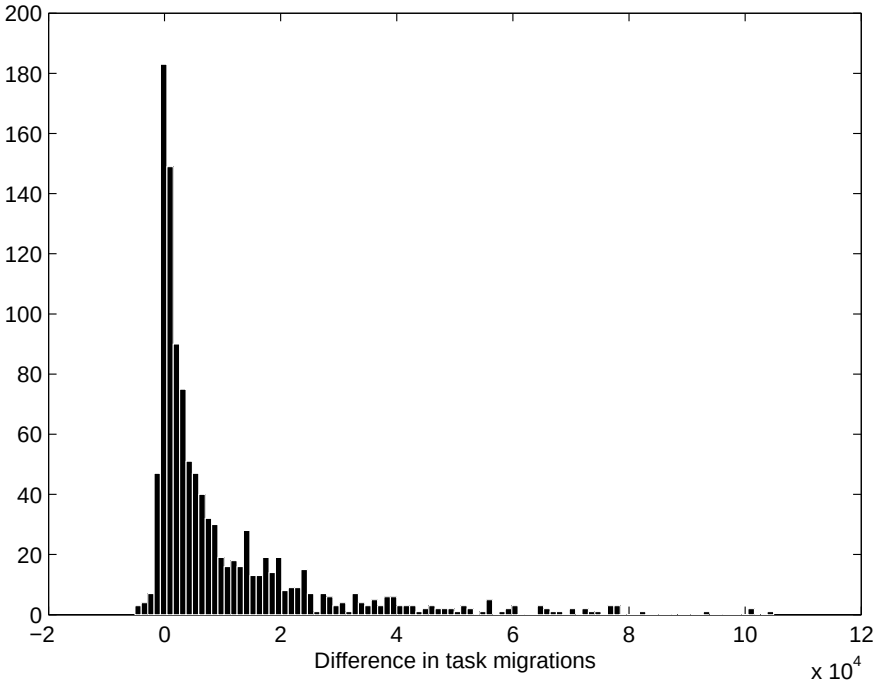


Figure 9 – Histogram of the difference in number of task migrations between LLREF and EDF on 4 processors

4 Discussion

4.1 Schedulability

Global EDF is far simpler than PF and LLREF, but it pays for it in schedulability. Where PF and LLREF are optimal, global EDF can only make loose guarantees. In all cases, increasing the CPU count increases the utilization level where missed deadlines may occur. However, normalizing these values shows that the system utilization does not necessarily increase with the CPUs. It might be that with four CPUs, global EDF strikes a better balance between the number of tasks (negative) and the available resources (positive). Certainly, moving from one to two CPUs is problematic, but going to four makes the system more fluid. This is more likely than that all four of the distributions are biased.

4.2 Scheduler Invocations

Scheduler invocations are a source of overhead, and in a real system, time spent scheduling detracts from the time available to run the tasks. Even in cases where the scheduler can be run offline, a higher number of scheduler invocations will lengthen the time required to find a feasible schedule. For this reason, it is desirable to minimize the number of times the scheduler is invoked.

In PF, the scheduler is invoked at every tick. Since LLREF is designed to improve on PF by minimizing the number of scheduler invocations, it only schedules on primary or secondary events. Therefore, if primary events occur on a relatively high percentage of ticks, the number of invocations will be similar to PF. This will occur if tasks have short periods, as LLREF must schedule on all primary events.

As the number of processors increases, the distributions of the difference between LLREF and PF remain fairly consistent. The exception to this is that the spike at 0 increases in size with number of processors. The reason for this is that when the tests are run for more processors, the task sets contain more tasks. The probability that one of these tasks will have a short period, and thus lead to a large number of scheduler invocations, is then higher. This may not be the case in real systems. In addition, as the number of tasks increases, the likelihood that any particular tick will be a multiple of one of the task periods increases, leading to more invocations in general. Since PF already is invoked in every time step, this only affects LLREF.

Global EDF has the fewest scheduler invocations; they only occur when a task arrives or completes. This is a subset of the invocations in LLREF, since LLREF considers task arrivals as primary events, and local task completion is only one of the possible secondary scheduling events. However, the trade-off there is that LLREF can schedule sets of tasks which are not feasible using global EDF. Our results show that, as expected, EDF always has fewer invocations than LLREF. As with the comparison of LLREF and PF, the number of cases where the schedulers have equal numbers of invocations increases with the number of processors. This has the same cause; if there is a task deadline at every tick, then both EDF and LLREF will schedule at every tick. Thus, for some sets of tasks, all three algorithms will be identical when considering number of scheduler invocations.

It should be noted that the amount of work done per invocation is not the same for each algorithm. Even within LLREF, primary scheduling events involve more work than secondary scheduling events. The number of events does at least provide an upper bound on the number of task migrations, however. Examining the work done in each scheduler invocation was outside the scope of this project.

4.3 Task Migrations

A goal of any multiprocessor scheduling algorithm should be to minimize the number of task migrations required to schedule a task set. A task migration requires making all the resources available to the new processor. Depending on the architecture of the system, this could be quite costly.

For all three algorithms, task migrations can only occur when the scheduler is invoked. Therefore, we observe generally similar trends as with scheduler invocations. However, these trends show different behavior due to the fact that a task migration does not necessarily occur on every scheduler invocation. The method each algorithm uses for determining when to migrate a task accounts for

these deviations from the scheduler invocation trends as well as the differences in behavior between the algorithms.

Task migrations could theoretically occur at every tick in PF. Since LLREF requires fewer scheduler invocations, it makes sense that LLREF would almost always require fewer task migrations as well. However, our simulation results show that the trends for scheduler invocations do not translate directly to the trends for task migrations when comparing LLREF and PF.

Specifically, we observe that there is a larger variation in the number of task migrations. This variation can be attributed to our LLREF implementation's bias to keep tasks that have hit the execution ceiling on the same processor while our PF implementation does not make any special accommodations to minimize task migrations. The original authors of LLREF and PF did not include any functionality in the algorithms to minimize task migrations. Therefore, we are not surprised that the differences between these algorithms in terms of task migration performance depend partially on the implementation.

Just as with scheduler invocations, global EDF requires the fewest task migrations. This makes intuitive sense because global EDF will only migrate a task when a preemption occurs. When comparing EDF to PF, we observed that PF always requires more task migrations than EDF. This can be attributed to PF requiring more scheduler invocations. Additionally, the PF algorithm does not make any effort to ensure that a task scheduled in two consecutive ticks remains on the same processor, and thus, this behavior constributes to the higher number of task migrations.

When comparing LLREF to EDF, we see that LLREF performs similarly to EDF in terms of task migrations, and most likely is outperformed by EDF because EDF requires fewer scheduler invocations. Additionally, LLREF must sometimes migrate tasks in order to execute all of them; in some cases, a task set that has many migrations is simply not schedulable with EDF.

5 Conclusion

We examined three scheduling algorithms for uniform multiprocessor systems: global EDF, PF, and LLREF. Global EDF, while simple and straightforward, is a non-optimal multiprocessor scheduling algorithm. We analyzed this shortcoming and showed that many task sets are not schedulable by global EDF. However, global EDF still demonstrated its merits with its low overhead in terms of task migrations and scheduler invocations. PF improved on global EDF by feasibly scheduling a large number of task sets that global EDF could not schedule but performed far worse than global EDF in terms of overhead. LLREF found a middle ground between global EDF and PF. In every aspect we evaluated, LLREF performed better than PF. In terms of overhead, LLREF performed worse than global EDF, but makes strides to reduce this performance gap when compared to PF. In the end, choosing whether to use one of these algorithms will require deciding whether optimality or low scheduler overhead is more important.

References

[1] Baker, T. P. A comparison of global and partitioned edf schedulability tests for multiprocessors. Tech. rep., In International Conf. on Real-Time and Network Systems, 2005.

[2] Baruah, S. K., Cohen, N. K., Plaxton, C. G., and Varvel, D. A. Proportionate progress: a notion of fairness in resource allocation. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing* (New York, NY, USA, 1993), STOC '93, ACM, pp. 345–354.

[3] Baruah, S. K., Gehrke, J., and Plaxton, C. G. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Symposium on Parallel Processing* (Washington, DC, USA, 1995), IPPS '95, IEEE Computer Society, pp. 280–288.

[4] Brandenburg, B. B., Calandrino, J. M., and Anderson, J. H. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *IEEE Real-Time Systems Symposium* (2008), pp. 157–169.

[5] Cho, H., Ravindran, B., and Jensen, E. D. An optimal real-time scheduling algorithm for multiprocessors. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 101–110.

[6] Holman, P., and Anderson, J. Adapting Pfair scheduling for symmetric multiprocessors. *Journal of Embedded Computing 1*, 4 (2005), 543–564.

[7] Liu, C. L., and Layland, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM 20* (January 1973), 46–61.

[8] Retis Lab, Scuola Superiore Sant'Anna. *The RTSIM project*. http://rtsim.sssup.it.