UNIVERSITÉ LIBRE DE BRUXELLES

Faculty of Science

Computer Science Dept.

INFO-F-404 : REAL-TIME OPERATING SYSTEMS

PROJECT 1: LEAST LAXITY FIRST

Authors

Chaste Gauvain, Ooms Aurélien

Monday 28th October, 2013

Academic year 2013 - 2014

Abstract

Study of the performance of LLF scheduling algorithm on systems with periodic, synchronous tasks and constrained deadlines. We consider systems of n periodic, synchronous and independent tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ with constrained deadlines embedded on a uniprocessor device. The work is divided in 3 parts: implementation of a LLF scheduling algorithm simulator, implementation of a system generator and study of LFF's performances.

Contents

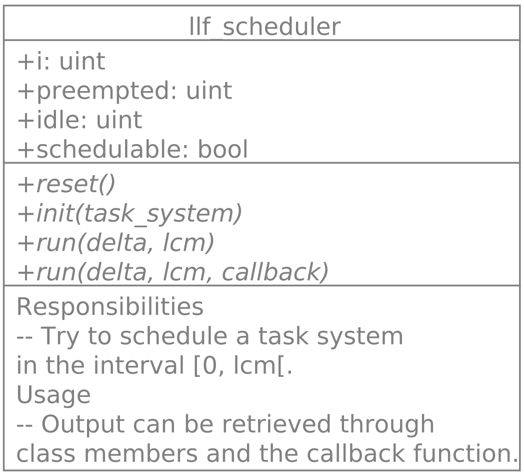

Figure 1: Interface of a scheduler

Source Code 1.1: Use case of the scheduler interface

```
os::llf_scheduler_event_based<os::task_system_t, os::job_t>
    scheduler;
scheduler.reset();
scheduler.init(task_system);
scheduler.run(d, lcm, callback);
```

1 Simulator

We implemented two versions of the simulator, one time based and one event based.

1.1 General structure

Figure 1 describes the interface of both implementations. A use case can be seen in Source Code 1.1.

✒ All the functionalities are splitted into small tools with aim to better flexibility, so for example in Source Code 1.1 you can see that the *lowest common multiple* is computed outside of the scheduler.

1.2 Time based

We first implemented the time based simulator for it's simplicity (See file).

A run of the scheduler looks like Algorithm 1.1

```
Data: system, lcm, d
schedulable ← true ;
preempted ← 0 ;
idle ← 0 ;
j ← null ;
queue ← ∅ ;
for i ← 0 to lcm do
    check for new jobs;
    if j = null ∧ llj ≠ null then
        j ← llj;                      /* llj := least laxity job */
    end
    else if j ≠ null ∧ d mod i = 0 ∧ j ≠ llj then
        preempted ← preempted + 1;
        j ← llj;
    end
    if j ≠ null then
        if i > startdeadline[j] then
            idle ← idle + 1;
            schedulable ← false;
            break;
        end
        else if timeleft[j] > 1 then
            startdeadline[j] ← startdeadline[j] + 1;
        end
        else
            delete j;
        end
    end
    else
        idle ← idle + 1;
    end
end
```

Algorithm 1.1: Run of the time based scheduler

Jobs are queued in an *std::multimap<uint, os::job_t>* where the key is the point in time where the job should imperatively be scheduled (start deadline) or else the system is not schedulable. This is better (from the implementation point of view) than directly considering the slack time because the start deadline will only increment for the current job whereas slack times would decrement for all idle jobs (not optimal for priority queues).

Δ_r has been interpreted as : *priorities of jobs are checked with a frequency of* $\frac{1}{\Delta_r}$*. However, if a job finishes, the cpu is left free and a new job can be handled without regards to* Δ_r*.*

1.3 Event based

The event based scheduler can be easily obtained by modifying the time based scheduler, the only changes to make are:

1. adding an event priority queue
2. adding triggers for events
3. computing the i steps as $i = hpe.i$ where hpe is the *highest priority event*

An example of the differences of implementations can be seen in Source Code A.1 and Source Code A.2.

The event loop can be seen in Source Code A.3.

(Link to the source code.)

1.4 Time vs Event

There are pros and cons considering both implementations. Here are listed some observations:

1. The event based approach consumes more memory because of the (small) overhead of the event queue.
2. The event based approach is much faster for $\Delta_r > 1$ (asymptotic Δ_r speed up factor).
3. The time based approach has to check very often if new jobs appeared. This is very inconvenient in the case of long long period tasks (1 check per time unit).

Source Code 2.1: The *floor_min_uniform* function

```
template<typename F, typename I, typename J>
F floor_min_uniform(F v, const I p, const J min, const F u){
    J i = v * p * u;
    i %= (J)((p * u) - min);
    i += min;
    v = i;
    return v / p;
}
```

2 Task System Generator

2.1 Uniform distribution

2.1.1 Random partitioning

We focused on producing uniformely distributed utilizations.

Suppose we want to generate a system $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ with

$$C_i \geq 1 \qquad \forall 1 \leq i \leq n \tag{2.1}$$

$$\sum_{i=1}^{n} U_i + \epsilon = U \tag{2.2}$$

$$\text{minimize } \epsilon \tag{2.3}$$

$$\epsilon \geq 0 \tag{2.4}$$

We choose $n - 1$ partition $p_j \in P =]0, U[$ with

$$|p_j - p_k| \geq \frac{1}{T_{min}} \qquad \forall p_j \in P, p_k \in P \cup \{0, U\}, p_j \neq p_k \tag{2.5}$$

To achieve Equation 2.5 we define the *floor_min_uniform* function (see Source Code 2.1).

An example of use can be seen in Source Code A.4.

(Link to the source code.)

2.1.2 Limits

The *lowest common multiple* is exponential in n. If we only use the primitive types provided by the C++ language we are bounded to a *max* value.

Sufficient condition for the feasability of lcm computation

$$T_{max}^n \leq 2^b - 1 \tag{2.6}$$

Where

$$\begin{aligned} T_{max} &= \text{the maximum value of the period distribution} \\ n &= \text{number of tasks in the system} \\ 2^b - 1 &= \text{the maximum value for an integer} \end{aligned}$$

Another requirement was to never overflow the U asked by the user.

Sufficient condition for a non-overflowing total usage

$$U_i \geq \frac{1}{T_i} \tag{2.7}$$

Where

$$\begin{aligned} U_i &= \text{usage of } \tau_i \\ T_i &= \text{period of } \tau_i \end{aligned}$$

For Equation 2.3 we can note that

$$\epsilon_{max} = \frac{n}{T_{max}} \tag{2.8}$$

We decided not to attach much importance to Equation 2.3 because of the spectrum of study considered (see section 3).

2.2 Specific Populations

In the previous section we exposed the way we chose to produce a uniform distribution of task systems.

This implementation will be used in section 3 but we could ask ourselves if there is no other option. It would be interesting to be able to study a specific population of task systems, this could be achieved with discrete non-uniform distributions of individual tasks.

3 Study

3.1 Spectrum

We studied the following ranges of values

- U [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
- Δ_r [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
- n [2, 3, 4]
- k 1000
- T_{min} 50
- T_{max} 100

The systems are generated with the technique explained in section 2.

Reading Equation 2.7 we have the warranty that $\epsilon_{max} = \frac{n}{T_{min}} \leq 0.08 < 0.1$. We can thus consider that results for U will be a biased mean of values in the interval $[U - 0.08, U]$.

We choose n and T_{max} small because of Equation 2.6.

A system is generated for each triple (U, n, k), this system is tested against the range of Δ_r values.

3.2 Results

The preemption rate is computed as $\frac{scheduler.preempted}{lcm}$ if the system is schedulable with the current configuration, $\frac{scheduler.preempted}{scheduler.i}$ otherwise (for schedulable systems $lcm = scheduler.i$).

Figure 2 shows an exponential decrease of the schedulability rate in u as well as a nearly linear increase of preemption rate with constant factor 0.1.

Figure 3 shows a nearly linear decrease of the schedulability rate in Δ_r with constant factor 0.02 as well as a negative exponential decrease of the preemption rate.

Figure 4 shows a nearly linear decrease of the schedulability rate in n as well as a linear increase of the preemption rate. This result is probably not exploitable since the range of n value is not wide enough.

Figure 5 shows the influence of u and Δ_r on the preemption rate $(n = 4)$.

Figure 6 shows the influence of u and Δ_r on the schedulability rate $(n = 4)$.

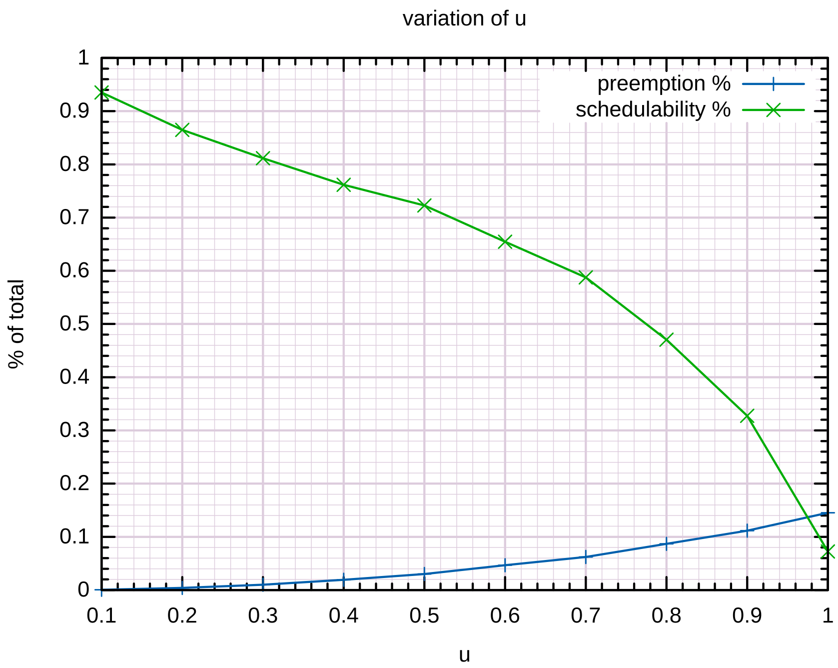

Figure 2: Schedulability and preemption rate depending on u

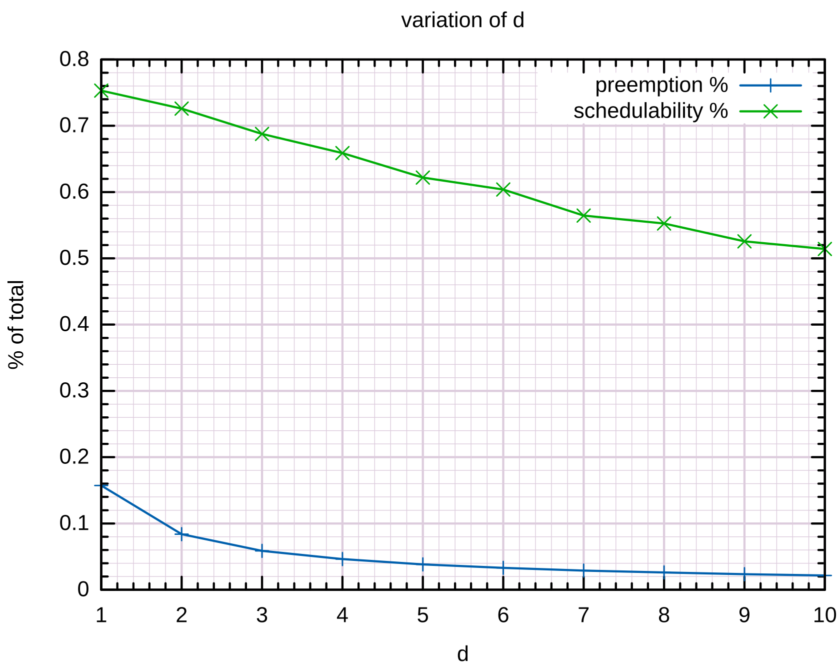

Figure 3: Schedulability and preemption rate depending on Δ_r

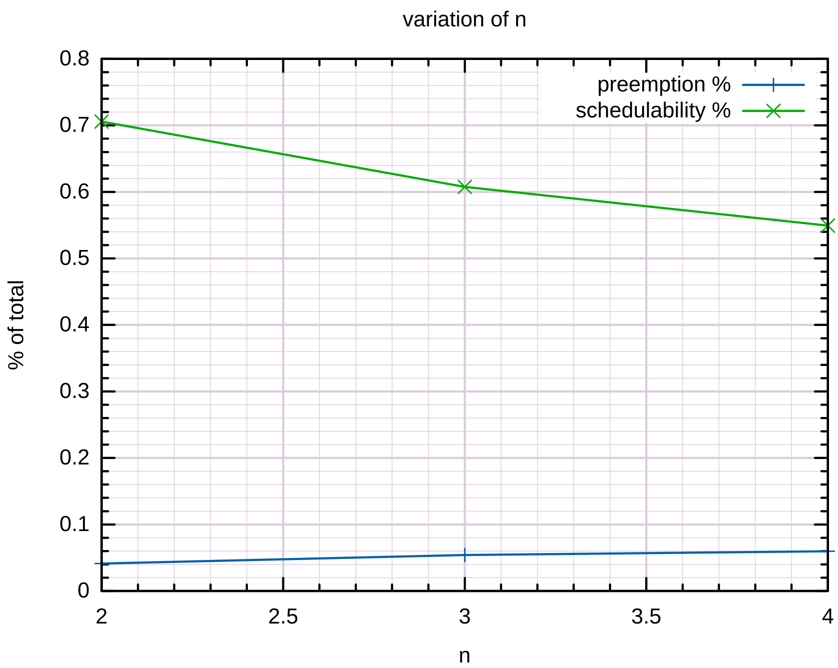

Figure 4: Schedulability and preemption rate depending on n

3.3 Interpretation

The results are not surprising.

S = schedulability rate
P = preemption rate

$$\uparrow U \Rightarrow \downarrow S \ \wedge \ \uparrow P \tag{3.1}$$

$$\uparrow \Delta_r \Rightarrow \downarrow S \ \wedge \ \downarrow P \tag{3.2}$$

$$\uparrow n \Rightarrow \downarrow S \ \wedge \ \uparrow P \tag{3.3}$$

The exponential decrease of the schedulability rate in u shown in Figure 2 suggests that to achieve a good use of the whole computation capabilities of the cpu some engineering to design the task system has to be done.

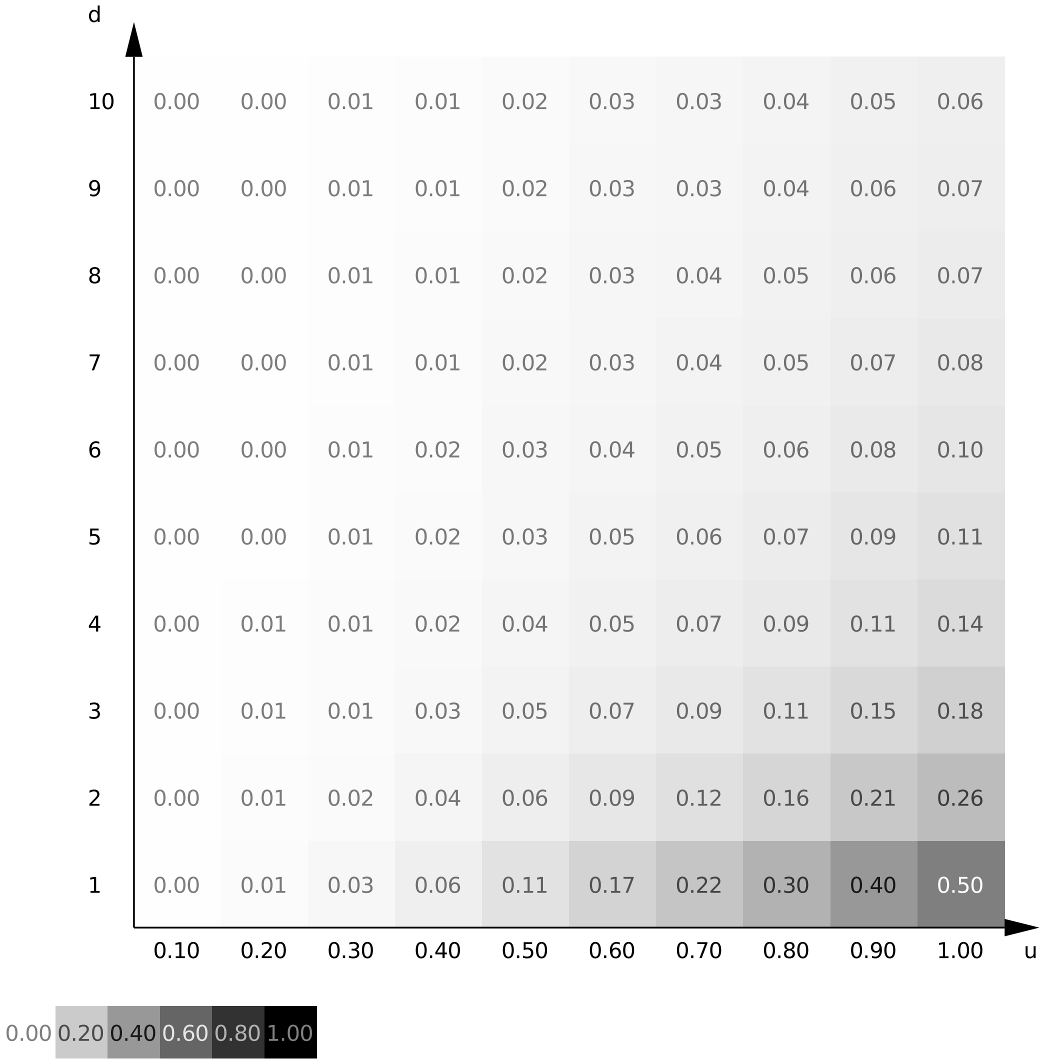

Figure 5: Preemption rate for $n = 4$

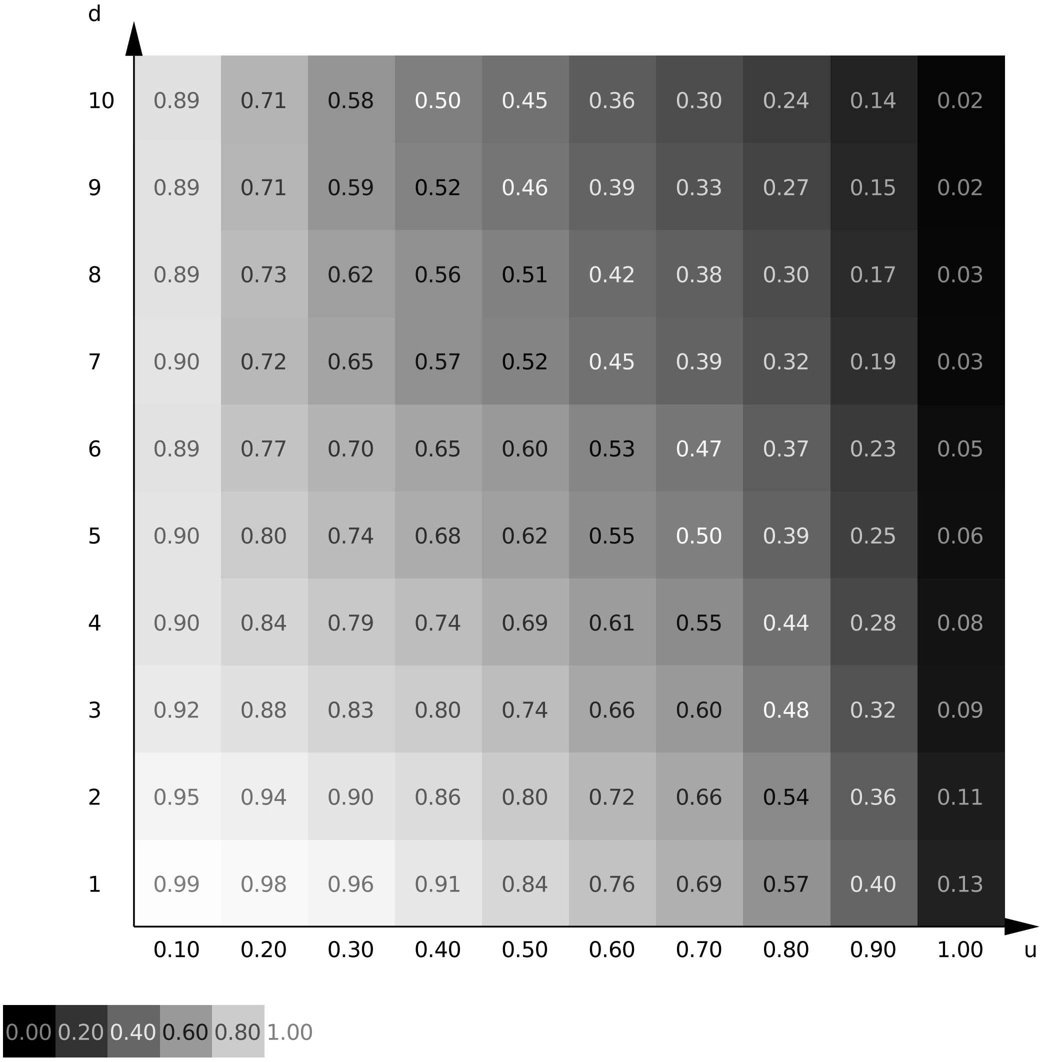

Figure 6: Schedulability rate for $n = 4$

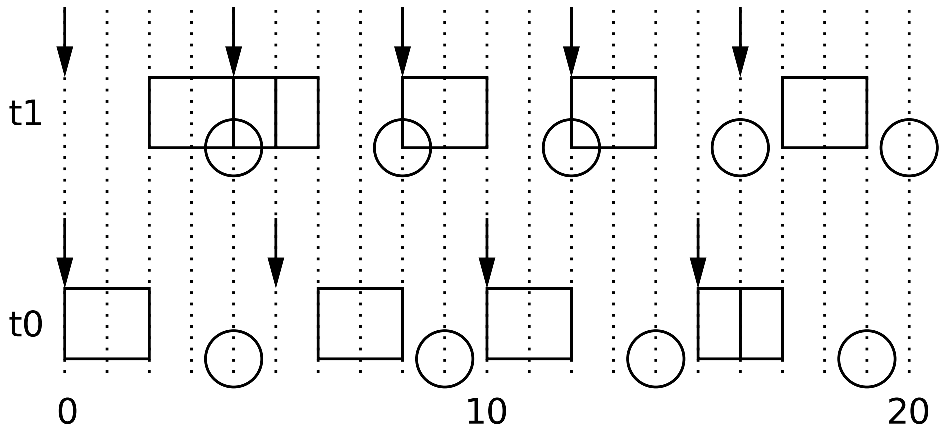

Figure 7: Output of the event pipe mode

4 Use Cases

4.1 Simulator

4.1.1 Regular mode

```
# ./run/simLLF 10 system/1
# cat system/1 | ./run/simLLF 10
```

4.1.2 Event pipe mode

```
# cat system/3 | ./run/simLLF 10 -p | ./run/plot_schedule shed-
ule/svg/1.svg -g 10
```

Figure 7 shows the output of this command.

4.2 Generator

```
# ./run/taskGenerator -u 0.7 -n 4
```

You can use the `--verbose` option to see more information.

You can use the `--seed` option to choose the seed of the generator.

4.3 Study

4.3.1 Command line used for section 3

```
# ./run/LLF_study -u 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 -d 1 2
3 4 5 6 7 8 9 10 -n 2 3 4 -k 1000 > benchmark/1
```

4.3.2 Splitting data for *gnuplot*

```
# cat benchmark/1 | ./run/gnuplot -o gnuplot/data/1 gnuplot/data/2
gnuplot/data/3
```

4.3.3 Plot data with *gnuplot*

```
# ./tools/gnuplot/render u gnuplot/data/1 gnuplot/svg/1.svg
# ./tools/gnuplot/render d gnuplot/data/2 gnuplot/svg/2.svg
# ./tools/gnuplot/render n gnuplot/data/3 gnuplot/svg/3.svg
```

4.3.4 Splitting data for custom color plot

```
# cat benchmark/1 | ./run/split_benchmark -o benchmark/2 bench-
mark/3 benchmark/4
# cat benchmark/2 | ./run/compute_mean -o mean/data/2p mean/data/2s
# cat benchmark/3 | ./run/compute_mean -o mean/data/3p mean/data/3s
# cat benchmark/4 | ./run/compute_mean -o mean/data/4p mean/data/4s
```

4.3.5 Plot data with custom color plot

```
# cat mean/data/2p | ./run/plot_mean -o mean/svg/2p –color 255 255
255 0 0 0
# cat mean/data/2s | ./run/plot_mean -o mean/svg/2s –color 0 0 0
255 255 255
# cat mean/data/3p | ./run/plot_mean -o mean/svg/3p –color 255 255
255 0 0 0
# cat mean/data/3s | ./run/plot_mean -o mean/svg/3s –color 0 0 0
255 255 255
```

```
# cat mean/data/4p | ./run/plot_mean -o mean/svg/4p –color 255 255
255 0 0 0
```

```
# cat mean/data/4s | ./run/plot_mean -o mean/svg/4s –color 0 0 0
255 255 255
```

Source Code A.1: Job progress (time based)

```
// if there is a current job
if(current != queue.end()){
    // deadline missed
    if(i > current->first){
        ++i;
        schedulable = false;
        callback(3, current->second.id, i, 0);
        break;
    }
    // there is still work to do
    else if(current->second.d - current->first > 1){
        queue_iterator  it  =  queue.insert(node_t(current-
   >first + 1, current->second));
        queue.erase(current);
        current = it;
        callback(2, current->second.id, i, i+1);
    }
    // current job done
    else{
        callback(2, current->second.id, i, i+1);
        queue.erase(current);
        current = queue.begin();
    }
}
else{
    ++idle;
}
```

A Code

Here are listed some interesting parts of the source code.

A.1 Differences of scheduler implementation

In Source Code A.1 and Source Code A.2 you can see that simulation steps computation differs.

A.2 Event loop

In Source Code A.3 you can see the event loop that allows much better performances in the event based implementation.

Source Code A.2: Job progress (event based)

```
// if there is a current job
if(current != queue.end()){
    // deadline missed
    if(i > current->first){
        ++i;
        schedulable = false;
        callback(3, current->second.id, i, 0);
        break;
    }
    // there is still work to do
    else if(current->second.d - current->first > (next - i)){
        queue_iterator it = queue.insert(node_t(current-
   >first + (next - i), current->second));
        queue.erase(current);
        current = it;
        callback(2, current->second.id, i, next);
        i = next;
    }
    // current job done
    else{
        callback(2, current->second.id, i, i + current-
   >second.d - current->first);
        i += current->second.d - current->first;
        queue.erase(current);
        current = queue.begin();
    }
}
else{
    idle += next - i;
    i = next;
}
```

Source Code A.3: Event loop

```
bool new_job = false;
bool check_priorities = false;
events_r range = events.equal_range(i);
const size_t count = events.count(i);
size_t j = 0;
typename events_t::iterator it = events.begin();
for(;j < count; ++j){
    switch(it->second.id){
        case NEW_JOB:{
            queue.insert(node_t(i + it->second.task-
    >deadline - it->second.task->wcet, J(it->second.task_id,
    i, it->second.task->wcet, i + it->second.task-
    >deadline)));
            callback(0, it->second.task_id, i, 0);
            callback(1, it->second.task_id, i + it-
    >second.task->deadline, 0);
            new_job = true;
            events.insert(event_p(i + it->second.task-
    >period, event_t(NEW_JOB, it->second.task_id, it-
    >second.task)));
            break;
        }

        case CHECK_PRIORITIES:{
            check_priorities = true;
            events.insert(event_p(i + delta,
    event_t(CHECK_PRIORITIES, -1, nullptr)));
            break;
        }

        case END_OF_INTERVAL:{
        }
    }
    typename events_t::iterator prev = it;
    ++it;
    events.erase(prev);
}
```

Source Code A.4: Usage of the *floor_min_uniform* function

```
while(sep.size() < n){
    sep.insert(floor_min_uniform(usage_distribution(generator),
   period_distribution.min(), 1u, u));
}
```

A.3 Random partitioning

Source Code A.4 shows the loop that fills the set of partitions.

List of Figures

List of Equations

List of Source Code

List of Algorithms