# Entropy-based scheduling performance in real-time multiprocessor systems

Carlos A. Rincon
*Computer Science Department*
*University of Houston*
Houston, Texas, USA
carincon@uh.edu

Daniel Rivas
*Computer Science Department*
*University of Houston*
Houston, Texas, USA
derivassanchez@uh.edu

Albert M. K. Cheng
*Computer Science Department*
*University of Houston*
Houston, Texas, USA
amcheng@uh.edu

2023 57th Annual Conference on Information Sciences and Systems (CISS) | 978-1-6654-5181-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/CISS56502.2023.10089704

***Abstract*—In this paper, we present the performance analysis of the entropy-based scheduling approach in real-time multiprocessor systems. We analyze the effect of using the entropy-based scheduling layer in deadline-based (global Earliest Deadline First (EDF)), laxity-based (Least Laxity First (LLF)), and PFair-based (PD2) scheduling algorithms by measuring the number of preemptions, the number of job migrations, and the number of task migrations. The performance comparison results between the selected scheduling algorithms with their entropy-enabled versions showed that the entropy layer reduces the number of task migrations for all studied algorithms and reduces the number of job migrations for LLF and PD2.**

***Index Terms*—Entropy-based scheduling, real-time systems, multiprocessors, performance analysis.**

## I. Introduction

In 2018, Rincon [1], using the information theory principles proposed by Shannon [2], presented an approach to reduce the number of task migrations on multiprocessor systems by adding an entropy-based scheduling layer to any multiprocessor scheduling algorithm. This approach aims to reduce the complexity of the scheduling problem by selecting the permutation that maps tasks to processors with the lowest entropy.

The results from [1] showed that for global EDF, the entropy-based scheduling layer is able to reduce the number of task migrations while generating the same amount of job migrations. However, the proposed solution was not tested in other types of real-time scheduling algorithms.

The purpose of this paper is to analyze the performance of the entropy-based scheduling layer on Deadline-based, Laxity-based, and PFair-based algorithms, using as dependent variables the number of preemptions, the number of job migrations, and the number of task migrations.

We also present Bench, a work-in-progress tool for helping with the performance analysis of real-time scheduling algorithms; more specifically, the generation of scenarios, automatic execution of scheduling algorithms against each one, and presentation of results in a developer-friendly format.

The rest of the paper is organized as follows. The next section describes the related work for the studied problem. In section 3, we present the design aspects of the studied entropy-based layer. In section 4, we present the design and implementation of Bench (a command-line-interface (CLI) created for benchmarking scheduling algorithms). Section 5 presents the performance comparison of the studied solution against Deadline-based, Laxity-based, and PFair-based algorithms using synthetic task sets generated by SimSo (Simulation of Multiprocessor Scheduling with Overheads [3]). Finally, we give our conclusions and future work in section 6.

## II. Related Work

Reducing the communication overhead between processors is a crucial element to consider in multiprocessor systems. In a real-time system, this overhead may lead a job to miss its deadline, which on a hard real-time system is catastrophic, while on a soft real-time system affects the quality of service.

Christodoulou [4] presented a heuristic to solve the resource-constrained project scheduling problem (RCPSP) based on maximizing the entropy of the project's resource histogram. The proposed RCPSP heuristic is based on the method presented by the same author which utilizes the general theory of entropy and two of its principal properties (subadditivity and maximality) to restate resource leveling as a process of maximizing the entropy in a project's resource histogram. This paper shows how entropy represents the uncertainty of a system and how it can be used to solve a particular type of scheduling problem.

Sharma and Nitin [5] presented a task migration technique based on entropy to select the processor to execute a task in a multiprocessor environment. They introduce entropy as a new governing parameter in real-time distributed systems to replace utilization. Using the maximum entropy principle, they present a scheduling solution for real-time systems in multiprocessor systems. Based on their results, the authors present the scaling factor and ability to compute the free space of a given processor as the key advantages of using entropy instead of utilization. This research shows that entropy can be used to represent the uncertainty of a system.

Rincon and Cheng in [6], [7], [8], and [9] presented the foundation for using information theory principles in real-time system scheduling as well as different scheduling solutions based on the proposed theory, for both uniprocessor and multiprocessor systems. For uniprocessor systems, the results showed that the overhead of using information theory is higher than the state-of-the-art scheduling solutions for real-time systems. For multiprocessor systems, a significant reduction in the number of job migrations and task migrations was achieved when comparing the scheduling solutions based on information theory principles with state-of-the-art multiprocessor scheduling algorithms for real-time systems.

## III. Entropy-based Scheduling Layer

From Shannon's work [2], we know that the entropy of a discrete random variable ($X$) is equal to $H(X) = \sum_{i=1}^{n}\left(p_i * log_b\left(\frac{1}{p_i}\right)\right)$, where $n$ is the number of possible values for the studied random variable, $b$ represents the information unit (2 = Shannon, $e$ = NAT, 10 = Hartley), and $log_b(1/p_i)$ represents the amount of information generated by each value of the studied discrete random variable.

Given $F(t)$=$\sum_{i=1}^{n} f_i(t)$, Rincon [1] defines the entropy of CPU $j$ at time $t$ ($H_{CPU_j}$) as $\sum_{i=1}^{n}(f_i(t)/F(t) * log_b(F(t)/f_i(t)))$, where $n$ is the number of tasks in the system, and $f_i(t)$ is the number of time instances of task $i$ executed in CPU $j$ at time $t$. He also defines the entropy of the system at time $t$ ($H_{SYS}(t)$) as the sum of the entropy of all the CPUs in the system at time $t$. For a system with $m$ processors, $H_{SYS}(t)$=$\sum_{j=1}^{m} H_{CPU\,j}(t)$.

For a system with $m$ processors and $n$ tasks, Rincon [1] implements a second scheduling layer based on the relationship between entropy and uncertainty, which maps the instance of the task selected by any real-time multiprocessor scheduling algorithm to the available processors, trying to reduce the uncertainty of the scheduling problem. At any scheduling time point $t$, the proposed layer performs the following steps: a) sorts $k$ jobs from the ready queue based on the criteria of the original scheduler (where $k$ is the number of available processors at time $t$); b) selects the first job from the sorted queue generated from step a (job to be scheduled); c) calculates the entropy of the system for all the mapping permutations between the selected jobs and the available processors; d) selects the mapping permutation which generates the least amount of entropy of the system; and e) assigns the job selected by the original scheduler to the available processor based on the selected mapping permutation.

Algorithm 1 presents the design of the entropy-based scheduling layer.

The activation of the entropy-based layer depends on the number of available processors. At any scheduling time $t$, the proposed solution is only executed if the number of available processors is greater than one (due to the number of permutations). A subset of the ready queue of size $k$, where $k$ is the number of available processors, is sorted based on the criteria of the original scheduler (*sortedJobs* = sort(*readyQ*, *schedulerCriteria*, *k*)). An array with all the permutations of the *sortedJobs* array is generated (*mappingP* = *permutations*(*sortedJobs*, *k*)). Finally, the total entropy of each permutation is calculated, choosing the processor where the selected job by the original scheduler is mapped based on the permutation with the lowest overall entropy (selectCPU(*availableCPUs*,min(*totalH*), *selectedJob*)).

**Algorithm 1** Entropy-based Scheduling Layer

```
Input:  readyQ, availableCPUs
Output: selectedCPU
 1: k= availableCPUs.size()
 2: if $k > 1$ then
 3:     sortedJobs = sort(readyQ, schedulerCriteria, k)
 4:     selectedJob = min(selectedJobs)
 5:     mappingP = permutations(sortedJobs, k)
 6:     for np=0; $np < mappingP$.size() do
 7:         for ncpu=0; $ncpu < k$ do
 8:             $H_{SYS}[np] = H_{SYS}[np] + H_{CPU}$(availableCPUs[ncpu],
                mappingP[np][ncpu])
 9:         end for
10:     end for
11:     selectedCPU=selectCPU(availableCPUs,min($H_{SYS}$),
                                    selectedJob)
12: end if
```

### *Reducing the Overhead of Using the Entropy-based Layer*

To reduce the overhead due to the vast amounts of data needed to calculate the entropy of the system, Rincon [1] defines the entropy of a processor $H(freq)$ as $log_b(NFreq)$ - (($log_b(currFreq)$ - $currH$) * ($currFreq$)-$currentTerm$+$newTerm$) / $NFreq$, where $freq$ is an array with the frequencies of the number of instances per task that have used the processor, $currFreq$ is the sum of the frequencies at the current scheduling point time, $NFreq$ = $currFreq$ + $extraFreq$, $currH$ is the entropy of the processor at the current scheduling point time, $currentTerm$ is equal to $freq[selectedtask]$ * $log_b(freq[selectedtask]$, and $newTerm$ is equal to ($freq[selectedtask]$ + $newIntances$) * $log_b(freq[selectedtask]$ + $newIntances$), where $newInstances$ are the number of instances of the selected task to be scheduled at the current scheduling time. Algorithm 2 presents the steps to compute the entropy of a processor using the proposed incremental approach:

**Algorithm 2** Incremental entropy per processor

```
Input:  freq, currFreq, currH, selectedTask, extraFreq
Output: H, NFreq

 1: NFreq = currFreq + extraFreq
 2: if NFreq == extraFreq then
 3:     H = 0
 4: else
 5:     if freq[selectedTask] == 0 then
 6:         currentTerm = 0
 7:     else
 8:         currentTerm = freq[seletedTask] *
                        $log_b$(freq[seletedTask])
 9:     end if
10:     newTerm = (freq[seletedTask] + extraFreq) *
                $log_b$(freq[seletedTask] + extraFreq)
11:     H = $log_b$(NFreq) - (($log_b$(currFreq) - currH) *
            (currFreq)-currentTerm+newTerm)/NFreq
12: end if
13: return H, NFreq
```

With this algorithm, Rincon [1] reduces the overhead to calculate the entropy of the system due to the vast amounts of data needed, by computing the entropy per processor incrementally.

## IV. Bench command-line-interface

Bench is a command-line interface (CLI) created for benchmarking scheduling algorithms. It is built on top of the SimSo library and offers a set of tools similar to those available in SimSo GUI while addressing the problem of batch generation of scenarios and execution of scheduling algorithms.

Bench is a simple self-contained Python module (bench.py) that offers the following features:

- Generates simulation scenarios using periods in a log-uniform distribution, including the configuration of CPU(s), utilization(s), etc.
- Lets you run the generated simulation scenarios against the specified schedulers (built-in or custom).
- Visualize the results in a web browser using Dash [10].
- Results are saved to sqlite [11] files so you can load and analyze them in any custom tool you prefer.

### A. Usage

The workflow with the bench tool usually looks as follows:

- Generate and save a simulation scenario.
- Run the simulation scenario from step 1 with one or more schedulers.
- Save the results.
- Visualize the results using the bench tool or any tool that accepts an SQLite database.

*1) Generating simulations:* The command for generating simulations looks as follows:

```
bench.py generate [-h|--help] <flags>
```

Available flags are:

- −−processors: List with the number of processors per simulation.
- −−utilizations: List of utilizations to be used per simulation.
- −−tasks: Total number of tasks per simulation
- −−experiments: Total number of simulations per experiment.

As an example, running the following command:

```
bench.py generate --experiments 10 --tasks 20 \
--processors 2,4,8 --utilizations 0.5,0.75,1.0
```

Will print:

```
writing to: bench-nearly-helped-dog-1608229536.sqlite
[SIM] procs: 2, utilization: 0.5, tasks: 20, experiments: 10
[SIM] procs: 2, utilization: 0.75, tasks: 20, experiments: 10
[SIM] procs: 2, utilization: 1.0, tasks: 20, experiments: 10
[SIM] procs: 4, utilization: 0.5, tasks: 20, experiments: 10
[SIM] procs: 4, utilization: 0.75, tasks: 20, experiments: 10
[SIM] procs: 4, utilization: 1.0, tasks: 20, experiments: 10
[SIM] procs: 8, utilization: 0.5, tasks: 20, experiments: 10
[SIM] procs: 8, utilization: 0.75, tasks: 20, experiments: 10
[SIM] procs: 8, utilization: 1.0, tasks: 20, experiments: 10
written to: bench-nearly-helped-dog-1608229536.sqlite
```

Bench will generate 9 simulations (since they are generated by taking the Cartesian product between processors and utilizations), each simulation with 20 tasks and 10 experiments.

The output file will automatically be saved to the current folder; however, it is possible to override the path using the −−output flag.

*2) Running simulations:* Running simulations is easier than generating them. The only important flags are the input file to use (the same file we generated in the previous step) and the duration of each simulation (defaults to 1000 milliseconds). To run simulations using one or more schedulers, we can execute the following:

```
bench.py run --input <SIMULATION-FILE> --duration 1000 \
simso.schedulers.<SCHEDULER> [MORE]
```

Custom schedulers (not part of the SimSo package) can also be specified as follows:

```
bench.py run --input <SIMULATION-FILE> --duration 1000 \
custom:<PATH-TO-SCHEDULER> [MORE]
```

The simulation will run a SimSo model for each combination of processors, utilizations, and tasks as specified in the generation step. Just like when generating simulations, the output file will be automatically saved to the current folder (can also be overridden with the −−-output flag).

*3) Visualizing results:* The results contain many metrics that can be visualized using the included parallel coordinates plot utility. To generate the plot, the following command must be executed:

```
bench.py chart --input <RUN-FILE>
```

The command above will create a small web server (powered by Dash [10]) that you can visit from your browser.

## V. Performance Analysis

The executed benchmarks are done by comparing the original scheduling algorithm implementations (vanilla) from the *SimSo* [3] Python library against their entropy-enabled implementation. The original algorithm generally remains unchanged, as the entropy layer is only executed when assigning CPU(s) to pending jobs.

We executed different scenarios to analyze the effect of the entropy-based scheduling layer, using as independent variables the number of CPU(s) and the utilization level, while the number of tasks remains fixed. The period for each task is randomly generated using a *loguniform* distribution from *SimSo*), and rounded to the nearest integer.

To get accurate results, we executed each scenario one hundred times (one hundred experiments), and assign the average of the experiments as the result for that specific scenario.

The number of CPU(s), the utilization percentage, and the number of tasks we used for the performance analysis are:

CPU: 2, 4, 6, 8

Utilization: 0.5 (50%), 0.75 (75%), 1.0 (100%)

Tasks: 20 independent tasks with implicit deadlines and no resource access.

Periods: log-uniform distribution

## *A. Results and discussion*

For each of the studied scheduling algorithms, we present a table showing the improvement for the entropy-enabled implementation compared to its original implementation. This improvement is calculated as follows:

Improvement = (OriginalResult - EntropyResult) / OriginalResult

For all studied algorithms, we notice that the probability of a migration (job or task) is inversely proportional to the utilization of the taskset and is directly proportional to the number of available processors (depending on the number of tasks per taskset).

*1) Deadline-based:* Global EDF Entropy vs. Global EDF:

For this experiment, we compared the performance of global EDF (Earliest deadline first) against global EDF with the entropy layer, with the latter being executed each time a job is scheduled (new or ready).

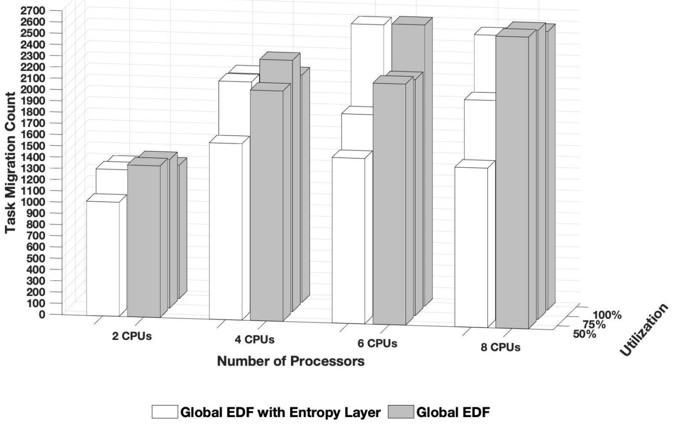

Fig. 1. Global EDF with Entropy Layer vs. Global EDF.

TABLE I
GLOBAL EDF ENTROPY VS. GLOBAL EDF

| CPU(s) | Utilization | Tasks | % Preemptions | % Job Migrations | % Task Migrations |
|---|---|---|---|---|---|
| 2 | 0.5 | 20 | -0.25 | 0.03 | 24.79 |
| 2 | 0.75 | 20 | -0.30 | -0.09 | 7.72 |
| 2 | 1.0 | 20 | 0.00 | 0.00 | 0.00 |
| 4 | 0.5 | 20 | 0.07 | 0.03 | 23.42 |
| 4 | 0.75 | 20 | 0.03 | 0.11 | 9.11 |
| 4 | 1.0 | 20 | 0.02 | -0.08 | 0.06 |
| 6 | 0.5 | 20 | -0.85 | 0.00 | 31.38 |
| 6 | 0.75 | 20 | -0.13 | -0.04 | 15.40 |
| 6 | 1.0 | 20 | 0.05 | 0.05 | 0.40 |
| 8 | 0.5 | 20 | 1.28 | 0.10 | 45.35 |
| 8 | 0.75 | 20 | -1.43 | -0.04 | 20.38 |
| 8 | 1.0 | 20 | 0.06 | -0.06 | 1.92 |

TABLE II
LLF ENTROPY VS. LLF

| CPU(s) | Utilization | Tasks | % Preemptions | % Job Migrations | % Task Migrations |
|---|---|---|---|---|---|
| 2 | 0.5 | 20 | 33.75 | 35.85 | 64.03 |
| 2 | 0.75 | 20 | 21.35 | 30.38 | 49.66 |
| 2 | 1.0 | 20 | 16.17 | 10.68 | 41.20 |
| 4 | 0.5 | 20 | 1.31 | 31.63 | 40.74 |
| 4 | 0.75 | 20 | 4.95 | 22.98 | 41.18 |
| 4 | 1.0 | 20 | 1.26 | 12.39 | 22.78 |
| 6 | 0.5 | 20 | -2.95 | 16.20 | 34.87 |
| 6 | 0.75 | 20 | -4.45 | 12.18 | 28.79 |
| 6 | 1.0 | 20 | -5.62 | 8.64 | 27.15 |
| 8 | 0.5 | 20 | 1.76 | 23.81 | 42.11 |
| 8 | 0.75 | 20 | 0.96 | 10.45 | 29.08 |
| 8 | 1.0 | 20 | -2.31 | 9.17 | 26.09 |

Table I shows that the *task migrations percentage* depends on the CPU utilization, with the former showing improvement the less utilization we have; this result makes sense since the entropy layer will have more freedom to schedule the best CPU from the available ones.

Results for the *Preemptions percentage* and the *job migrations percentage* can be explained by the fact that global EDF is a fixed-job-priority scheduling algorithm; therefore, no significant difference was observed for these parameters. These results show the same behavior as the ones presented by Rincon [1].

Based on the previous analysis, Figure 1 compares the performance of global EDF (Entropy vs. Normal) regarding task migrations.

*2) Laxity-based:* LLF Entropy vs. LLF:

For this experiment, we compared the performance of LLF (Least laxity first) against LLF with the entropy layer, with the latter being executed each time the scheduler is interrupted to calculate the laxities.

Table II shows the improvement of LLF Entropy in terms of the *Job migrations percentage* and the *Task migrations percentage* with respect to utilization. This performance improvement is due to the increase in the number of times the entropy layer gets called, given the effect of the laxity calculation in the scheduling algorithm.

Given that the performance improvement in terms of the *Job migrations percentage* is a consequence of both the entropy layer and the dynamic priority nature of LLF, Figure 2 compares LLF Entropy and LLF only based on the *Task migrations percentage*.

*3) PFair-based:* PD2 Entropy vs. PD2:

For this experiment, we compare regular PD2 (fair) against PD2 with the entropy layer, with the latter being executed each time the scheduler is interrupted (each tick) to recompute the virtual jobs.

Similar to the *Laxity-based* case, Table III shows the improvement in the *Job migrations percentage* and *Task migrations percentage* given the higher probability of a migration (job or task) due to the execution of the scheduling algorithm based on the value of the quantum.

The behavior of this algorithm for the *Preemption percentage* can be explained by the optimality condition of PD2 [12] for any number of processors, given the characteristics of the

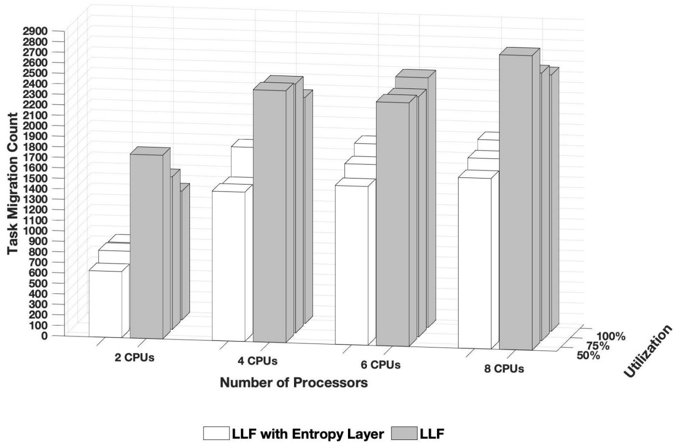

Fig. 2. LLF with Entropy Layer vs. LLF.

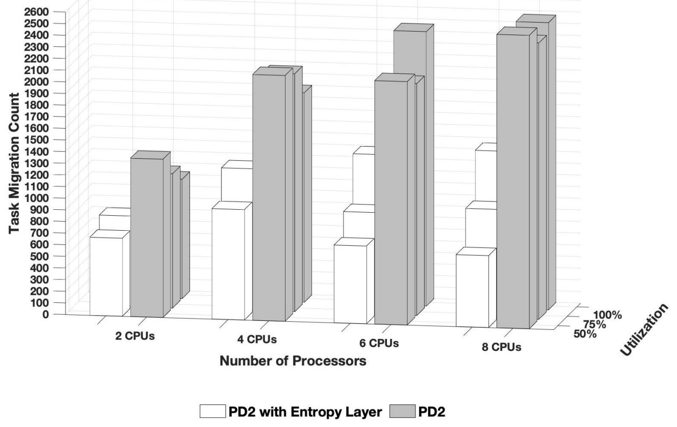

Fig. 3. PD2 with Entropy Layer vs. PD2.

TABLE III
PD2 Entropy vs. PD2

| CPU(s) | Utilization | Tasks | % Preemptions | % Job Migrations | % Task Migrations |
|---|---|---|---|---|---|
| 2 | 0.5 | 20 | -2.68 | 87.52 | 50.72 |
| 2 | 0.75 | 20 | -8.59 | 71.10 | 32.48 |
| 2 | 1.0 | 20 | -44.70 | 43.31 | 44.47 |
| 4 | 0.5 | 20 | 10.24 | 92.74 | 55.09 |
| 4 | 0.75 | 20 | -12.74 | 81.95 | 40.45 |
| 4 | 1.0 | 20 | -120.54 | 48.96 | 45.16 |
| 6 | 0.5 | 20 | 26.87 | 96.41 | 68.07 |
| 6 | 0.75 | 20 | -13.49 | 88.78 | 56.15 |
| 6 | 1.0 | 20 | -163.00 | 44.10 | 45.30 |
| 8 | 0.5 | 20 | 28.99 | 97.90 | 75.51 |
| 8 | 0.75 | 20 | -25.58 | 90.56 | 58.33 |
| 8 | 1.0 | 20 | -174.79 | 41.92 | 45.23 |

tasksets used for the experiments.

It is important to mention that the entropy-enabled PD2 implementation is the slowest out of all studied algorithms because the entropy layer executes on each scheduler call (which is significantly more frequent than EDF and LLF).

## VI. Conclusion and Future Work

This paper presents the performance analysis of the entropy-based scheduling approach in real-time multiprocessor systems.

We also introduce Bench, a work-in-progress tool for helping with the performance analysis of real-time scheduling algorithms; more specifically, the generation of scenarios, automatic execution of scheduling algorithms against each one, and presentation of results in a developer-friendly format.

In general, all the entropy-enabled implementations show improvement on the *Job migrations percentage* and *Task migrations percentage*; however, scheduling algorithms that interrupt more often are prone to performance degradation due to the overhead of the entropy layer.

Currently, the entropy layer selects the permutation between the selected jobs and available processors that generates the least amount of entropy. Future work will involve modifying the entropy layer by finding a sub-optimal solution that minimizes entropy without analyzing all permutations. Additionally, more complex benchmarks will be used to measure the performance of the proposed solution in non-synthetic scenarios.

## References

[1] C. A. Rincon, *Real-time System Scheduling and Information Theory: From Uniprocessors to Multiprocessors*. PhD thesis, University of Houston, 2018.
[2] C. E. Shannon, "A mathematical theory of communication," *Bell system technical journal*, vol. 27, 1948.
[3] M. Chéramy, P.-E. Hladik, and A.-M. Déplanche, "Simso: A simulation tool to evaluate real-time multiprocessor scheduling algorithms," in *Proc. of the 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, WATERS, 2014.
[4] S. E. Christodoulou, "Entropy-based heuristic for resource-constrained project scheduling," *Journal of Computing in Civil Engineering*, vol. 31, no. 3, p. 04016068, 2017.
[5] R. Sharma and Nitin, "Entropy, a new dynamics governing parameter in real time distributed system: a simulation study," *IJPEDS*, vol. 29, no. 6, pp. 562–586, 2014.
[6] C. A. Rincon and A. M. K. Cheng, "Using entropy as a parameter to schedule real-time tasks," in *2015 IEEE Real-Time Systems Symposium*, pp. 375–375, Dec 2015.
[7] C. A. Rincon and A. M. K. Cheng, "Poster abstract: Preliminary performance evaluation of hef scheduling algorithm," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 1–1, April 2016.
[8] C. A. Rincon and A. M. K. Cheng, "Using information theory principles to schedule real-time tasks," in *2017 51st Annual Conference on Information Sciences and Systems (CISS)*, pp. 1–6, March 2017.
[9] C. A. Rincon, X. Zou, and A. M. K. Cheng, "Real-time multiprocessor scheduling algorithm based on information theory principles," *IEEE Embedded Systems Letters*, vol. 9, pp. 93–96, Dec 2017.
[10] P. T. Inc., "Collaborative data science." url = https://plot.ly, 2015.
[11] R. D. Hipp, "SQLite." url=https://www.sqlite.org/index.html, 2020.
[12] A. Srinivasan and J. H. Anderson, *Efficient and Flexible Fair Scheduling of Real-Time Tasks on Multiprocessors*. PhD thesis, 2003. AAI3112080.