Evaluate Priority Assignment using Machine Learning Model (PAL) on a Large Task Set

Hong Tran (hongtran@wayne.edu)

Advisor: Prof. Nathan Fisher

Abstract

The real-time (RT) scheduling for multiprocessors has drawn considerable attention from the RT community in recent decades. However, there are still exist various challenging problems which are not completely solved. One of the hard problems is finding a priority assignment for global fixed task-priority preemptive (gFP) scheduling on a multiprocessor platform. This problem is difficult in the essence of intractability with a large number (n) of tasks in a task set (m, n) where m is a number of processors. Recent research [1] claimed that by integrating a pointer network from machine learning (ML) discipline into the RT domain, this gFP problem can be solved with a large number of tasks (n). From the research's evaluation, the proposed solution seems to perform very well with small (m, n) but its performance degrades gradually if either m or n are increased.

This project targets to re-implement the proposed model [1], conduct an extensive test, and provide an explanation for degradation of the model on task sets with large m and n.

1. Introduction

1.1 Problem Statement

To find a priority assignment of a large task set (m, n) for gFP problem, there are following potential approaches:

1. Exhaustive testing for all possible combination of the priority assignment. This method consumes a significant amount of time as much as $O\left(n! \cdot n^2 \cdot \underset{\tau_i \in \tau}{max} T_i\right)$. With a large of tasks (n), this approach is intractable.
2. Using low-performance OPA-compatible (optimal priority assignment) schedulability test in [2] and [3] then apply to Audsley's OPA [4]. This method has a drawback that not able to maximize all the m processors' performance.
3. Using high-performance OPA-non compatible stimulability test [5], [6], [7], [8] and apply them on heuristic priority assignments [9], [10], [11].
4. Using the proposed ML model in [1] to infer input task sets.

Each approach has its own pros and cons. The OPA-compatible scheduablitiy test [2], [3] with Audsley's OPA priority assignment can run as fast as $O(n^2)$ but it prevents use of slack values which are generated by relative priority of the higher -priority tasks and used to reduce the interference of the target task. Therefore, this approach exhibits low performance in term of scheduablitiy. For the OPA-non compatible tests [5], [6], [7], [8], it demonstrates higher performance by utilizing the slack values of higher-priority tasks; however, it cannot work with Audsley's OPA algorithm.

The proposed model in [1] demonstrates a method to incorporate ML and RT domain knowledge to solve the gFP scheduling on a multiprocessor platform with a large number of tasks (n). This PAL model is evaluated and compared with other models:

- **Exhaustive**: exhaustively testing all priority assignments (all permutations) of each task set with non-OPA compatible RTA-LC schedulability test.
- **DaBu**: OPA Limited Backtracking Priority Assignment with non-OPA compatible RTA-LC schedulability test [15].
- **ZLL**: Heuristic priority assignment with an improved RTA-LC schedulability test [8].

PAL model performs very well with $m \leq 4 \; and \; n \leq 20$ comparing to the existing well-known heuristic algorithms. However, with $m = 6$ and $n \geq 20$, it seems that the heuristic algorithms show a similar performance or even better than the proposed model [1]. This observation is contradicted with the goal of the proposed model. Therefore, the recent proposal in ML [1] exhibits a promising outcome but it is still lacking a concrete result and reasonable explanation for the degradation of the model when m and n are both increased.

1.2 Project Scope

The original goals of this project are as follows:

1. Implement the ML model proposed in [1].
2. Carry out an additional test for a large number of tasks (n) and processor (m).
3. Compare this test result with the heuristic approaches'.
4. Investigate why the model performance does not surpass the heuristics approaches with large n and m.

However, during various experiments, I found out that it is impossible to generate a large training task set ($m = 2, n \geq 8$) for the PAL model in the short timeframe of the project (see section 4.2). Therefore, instead of trying to evaluation and investigate with a large task set, I target to evaluate the model performance degradation on small task sets ($m = 2, n < 8$) and comparing the model with the exhaustive search on these task set.

1.3 Project Challenges

- The research [1] is relatively new (RTAS 2021) and hence there are not many references or discussions about this research.
- The model in the research [1] uses various schedulability tests, heuristic algorithms, and involving a pointer network model from machine learning discipline.

- There is no source code or data available from the paper's authors.
- From the author of the paper [1], the data preparation itself took 02 weeks running on a high-performance PC. Therefore, this project is challenging, given the fact that this project only has 05 weeks to complete which includes time to understand the paper, implement the model, prepare the data, perform extensive tests and analyze test results.

1.4 Project Report Structure

The reminder of this report is organized as follows.

- Section 2: **Overview on Multiprocessor scheduling**
- Section 2: **Structure of the PAL model**
- Section 3: **My implementation of PAL model and Experimental Results**
- Section 4: **My conclusions and future work.**

2. Overview on Multiprocessor scheduling

In the past, many technologies were focusing on improving processor performance by increasing its clock frequency. However, this approach has led to the problems with both high-power consumption and much heat dissipation. This is considered as a blocking point for a battery-powered devices (i.e., mobile devices). During the recent decades, many vendors include, from AMD, Intel, ARM, Freescale Semiconductor, NXP, etc. have move towards the multiprocessor systems. The integration of multiple processors into a single chip is one of the most important achievements in the modern embedded systems.

Scheduling for multiprocessor environments is divided into following main categories (Figure 1):

- **Global scheduling**: With the task migration allowed, the system uses a single queue to dynamically assign all the arrived task into an appropriate processor.

- **Partitioned scheduling**. In opposite with the global scheduling, this configuration does not allow task migration. Scheduling algorithm has a separated queue for each processor. Processor's affinity for each task is fixed.
- **Semi-partitioned scheduling** – This configuration is a mixed version of the two previous ones. Task or part of its can be migrated among the processors and their processor's allocation are fixed.

There are many of scheduling and schedulability analysis of single-processor systems available but most of the results do not extend to multiprocessor environments. For example, the dynamic task priority assignment EDF or fixed priority scheduling RM are optimal in uniprocessor system which is no longer holds on multiple processor system. In general, the algorithms for single processor scheduling may work but the schedulability analysis is not a trivial task.

In this paper, I want to focus on global fixed task-priority assignment (gFP) on a symmetric multiprocessor (SMP).

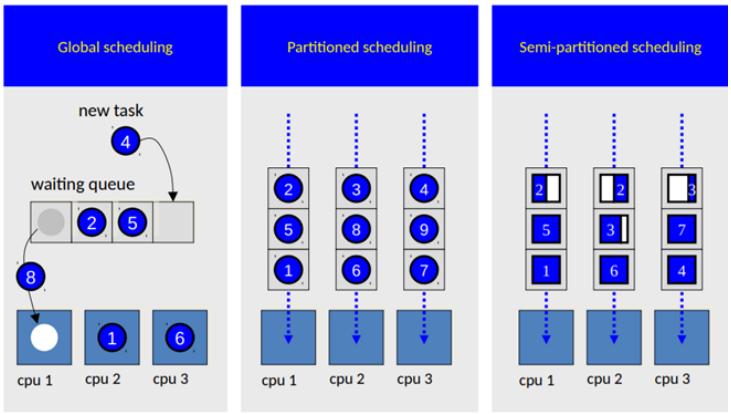

Figure 1 Scheduling in Multiprocessor Systems

3. Structure of the PAL model

The PAL model proposed in [1] integrates a pointer network from Machine Learning (ML) discipline into the Real-time (RT) domain to solve gFP problem with a large number of (n) task.

The training and test samples for PAL models contains an implicit-deadline sporadic taskset (m, n) and a priority assignment. The pointer network (Figure 1) with a unique feature that its trained model can infer beyond the maximum length of output they have trained on. For example, Consider the implicit-deadline sporadic task model, i.e., $\tau_1(T_1 = 5, C_1 = 1)$, $\tau_2(T_2 = 3, C_1 = 2)$ and $\tau_3(T_3 = 10, C_3 = 5)$, and the 1st, 2nd, 3rd priority tasks are τ_2, τ_1 and τ_3, respectively. By training the PAL model with a sufficiently large number of samples, the PAL model can infer a schedulable priority assignment from a given task set.

Input Test Sample `[[5,1][3,2][10,5]]`→ **PAL Model** → **Priority Assignment**

Output `[2,1,3]`

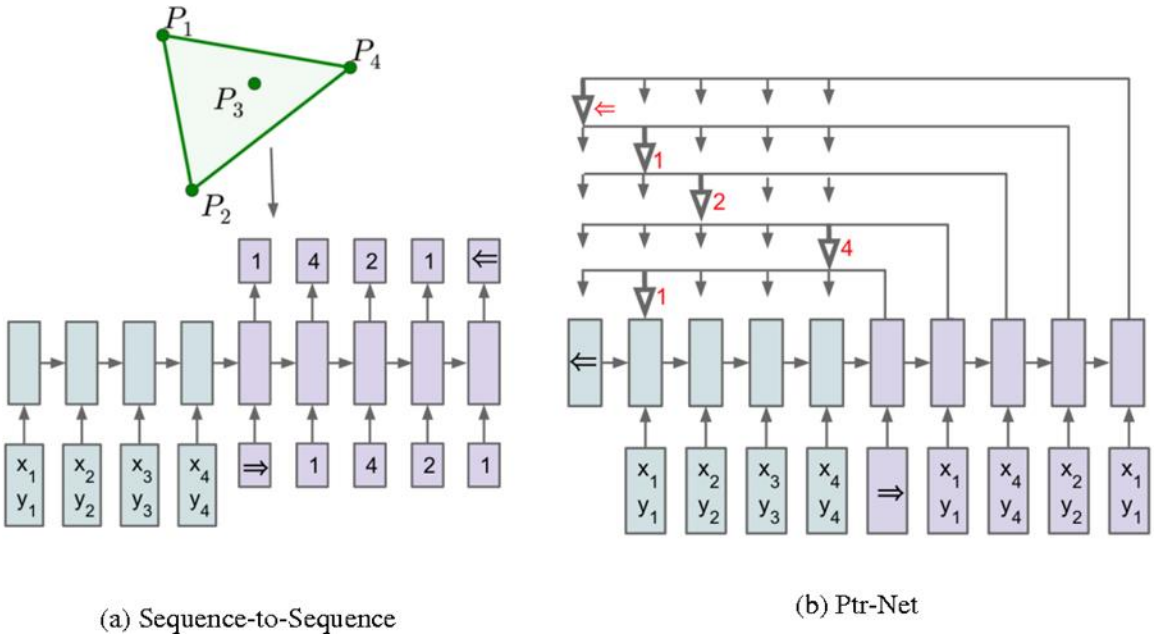

Figure 2 Pointer Network

The main ideas inside the PAL model are how to select the training samples and how to effectively generate training samples for a large taskset (m, n).

3.1 How to select the best training samples for PAL models:

PAL is designed with an assumption that a training sample with a best schedule preservability can help the PAL inferring effectively. Based on this idea, PAL adopts the system hazard concepts in [12] in the process to find good training samples. The system hazard (which is named as premier priority assignment) is defined as follows:

$$\textbf{System Hazard} = \min(\Theta(\tau, P, m)) = \min\left(\max_{\tau_i \in \tau} \frac{R_i}{T_i}\right),$$

$$R = response\ time,\ \ T = period,\ P = priority\ assignemnt,\ m = number\ of\ processor$$

System hazard [PeSh93]

- Maximum ratio between response time and the period for every task
- Task set with smaller system hazard is easier to preserve schedulability throughout changes of task parameters

Figure 3 System Hazard Concept

However, it turns out that finding the best system hazard of a taskset is intractable with a large number of tasks in a taskset (m,n) due to its complexity $O\left(n! \cdot n^2 \cdot \max_{\tau_i \in \tau} T_i\right)$ increasing proportional with $n!$.

We can consider the following example:

Train $\tau_1(T_1 = 5, C_1 = 2),\ \tau_2(T_2 = 5, C_1 = 2),\ \tau_3(T_3 = 10, C_3 = 1)$

τ = tasks, T = period, C = worst case execution time

where all cases(3!) are schedulable

Priority	System hazard
(τ_1, τ_2, τ_3)	$\rightarrow\ \Theta = 0.7$
$(\boldsymbol{\tau_1, \tau_3, \tau_2})$	$\rightarrow\ \boldsymbol{\Theta = 0.5}$
(τ_2, τ_1, τ_3)	$\rightarrow\ \Theta = 1.0$
(τ_2, τ_3, τ_1)	$\rightarrow\ \Theta = 0.7$
(τ_2, τ_3, τ_1)	$\rightarrow\ \Theta = 1.0$
(τ_2, τ_3, τ_1)	$\rightarrow\ \Theta = 0.6$

To find a best system hazard for a taskset (m=2, n=3), we need to compute on all 3! priority assignment of the taskset.

To tackle this problem, PAL decides to use the best system hazard for the priority assignment that the model can find in the reasonable time frame (this is denoted as pseudo-premier priority assignment). This can be done through following steps (Figure 4):

Priority		Selected by heuristics		System hazard
(τ_1, τ_2, τ_3)	→	(τ_1, τ_2, τ_3)	→	$\Theta = 0.7$
(τ_1, τ_3, τ_2)	→	X	→	$\Theta = 0.5$
(τ_2, τ_1, τ_3)	→	(τ_2, τ_1, τ_3)	→	$\Theta = 1.0$
(τ_2, τ_3, τ_1)	→	X	→	$\Theta = 0.7$
(τ_2, τ_3, τ_1)	→	X	→	$\Theta = 1.0$
(τ_2, τ_3, τ_1)	→	(τ_2, τ_3, τ_1)	→	$\Theta = 0.6$

Figure 4 Generate Training and Test Samples for PAL

1. Generate random tasksets.
2. Filter out all the schedulable taskset that can be easily found by heuristic priority assignments (e.g., DMPO, D-CMPO, DkC).
3. Choose a priority assignment that has a smaller system hazard than the one of all heuristic priority assignments.

3.2 How to effectively synthesize a large number of tasks in taskset (m,n)

The previous approach on finding pseudo-premier priority assignment significantly improves the performance of the taskset generating process. However, the with a large number of n (i.e., n>8), this process is still impractical in term of time consuming (for taskset (2,8), it took 2 hours and 7 hours to calculate the pseudo premier and premier for 100 tasksets, respectively. This does not include the time to generate and filter out random tasksets – step 1 and 2 in the section 3.1).

PAL proposed a better idea to inductively generate a large taskset based on smaller ones that have been generated already. In general, for generating the taskset (m,n), PAL will:

1. Synthesize small amount of biased (m,n) tasksets.
2. Re-use the previous generated taskset with smaller n.
3. Train PAL with tasksets from step (1) and (2), then it infers a new (m,n) taskset.
4. Add the inferred taskset (m,n) to the training set and repeat the step (3).

3.2.1 Generating training sample Process

A random taskset is generated with following steps:

1. Setup the bimodal distribution as documented in [13]
   p=[0.1, 0.3, 0.5, 0.7, 0.9]

For a given bimodal parameter p, a value for C_i/T_i is uniformly chosen in [0,0.5) with probability p, and in [0.5,1) with probability $1-p$.

2. Setup exponential distribution with $\mu = 1/\lambda$=[0.1, 0.3, 0.5, 0.7, 0.9] [13]

For a given exponential parameter $1/\lambda$, a value for C_i/T_i is chosen according to the exponential distribution whose probability density function is $\lambda \cdot \exp(-\lambda \cdot x)$.

3. Uniform-choose between above 10 distributions (bimodal and exponential distributions)
4. Generate n tasks:
   a. Generate T_i in range [10, 10000] with log-uniform distribution.

      Using following equation with T_g=1 [14]

$$r_i \sim U\left(\log T_{\min}, \log\left(T_{\max} + T_g\right)\right)$$

$$T_i = \left\lfloor \frac{\exp(r_i)}{T_g} \right\rfloor T_g$$

The uniform random values r_i produced are assumed to lie in the range $\left[\log T_{\min}, \log\left(T_{\max} + T_g\right)\right)$. $T_{\min}$ and $T_{\max}$ should be chosen as multiples of T_g.

   b. Generate C_i/T_i according to the chosen distribution,
   c. C_i as a multiplication between the generated T_i and C_i/T_i.

5. Discard a taskset (m, n) if:
    a. Condition C1: It is schedulable with DMPO + RTA-LC or DkC + RTA-LC or D-CMPO + RTA-LC
    b. Condition C1: It is schedulable with Audsley OPA + DA-LC
    c. Condition C2: It is not schedulable with necessary test C-RTA

The step 5 tries to find tasksets (red area in the Figure 5) which are not easily identify by well-known heuristics algorithms (DMPO, D-CMPO, DkC) and optimal priority assignment Audsley OPA but these tasksets are still likely to be schedulable (by the C-RTA test).

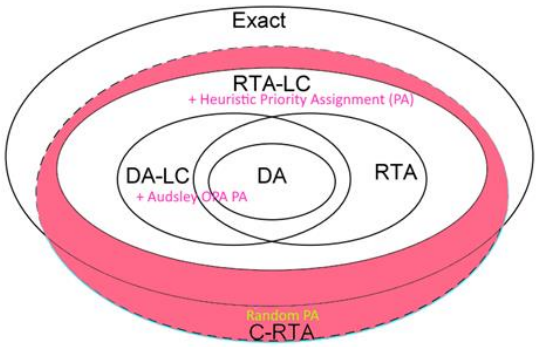

Figure 5 Random Taskset Selection

3.2.2 Finalizing training samples for PAL model:

Using the generated taskset from section 3.2.1 above, the training samples for PAL model can be formed in following sequences:

1. With the smaller number of task (i.e., $m = 2, 4 \leq n \leq 8$), PAL exhaustively search for each task's premier priority assignment (explained in section 3.1)
2. With large number of task (i.e., $m = 2, n \geq 9$), PAL uses its inductive properties to form the large number of training sample based on the taskset with smaller number of tasks (refer to the Figure 6)

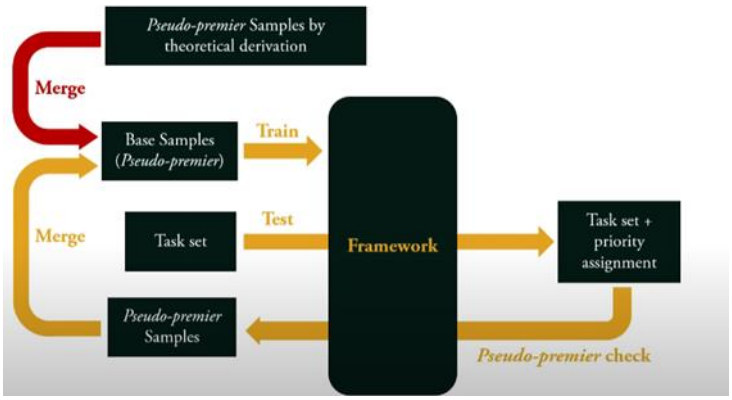

Figure 6 Generate Large n Tasks

For example, we want to generate the training sample for (m=2, n=9):

1. Using Nunbiased $=\frac{500k}{n-5}$ test samples from each $(2,5), (2,6), (2,7), (2,8)$ tasksets.
2. Apply data augmentation with $\rho = 5$ on all the Nunbiased samples.
3. Using Nbiased = 100 (by applying inductive property on 100 unbiased samples $(2, n-1)$).
4. Training PAL with $5 * N^{unbiased}$, and $5 * N^{biased}$ training samples.
5. Using PAL to infer a taskset $(2,9)$ to find a potential priority assignment.
6. Apply pseudo-primer check on the priority assignment.
7. Merge the found priority assignment to the training taskset.
8. Repeat from step 5 until we secure $\frac{100k}{n-4}$ training samples for $(2,9)$.
9. Apply data augmentation with $\rho = 5$ on all the generated $(2,9)$ samples. We will have total $5 * \frac{100k}{n-4}$ samples (2,9) ready for the training process.

4. My implementation of PAL model and Experimental Results

4.1 PAL Implementation

My implementation of PAL model runs on Python. This code reuses the pointer-network [16] and RTA-LC test in [17]. For the remaining parts of the project are developed from scratch including (premier/pseudo premier algorithm, C-RTA test, D-RTA-LC, Audsley OPA priority assignment, training samples generation process, etc.).

During my implementation, I observed that the process to generate sample task set running extremely slow. For (m=2, n=8), the generation rate is 0.06 taskset/10s. This does not include the exhaustive search for the best system hazard (which will take more O(n!))

After doing some run-time profiling, it turns out that there are two blocking points:

- Generating fix sum utilization.
- Filtering out taskset if it violates either (C1) non-schedulable taskset on DMPO with RTA-LC test, Audsley with DA-LC test and (C2) schedulable C-RTA test with some random priority assignment.

My original implementation for generating fix sum utilization was trying and testing random set of n tasks and make sure the total utilization $U \leq m$. This naive approach did not perform well as number of discarded tasksets are huge, especially when n becomes larger. I did try other algorithms including UUniFastDiscard and StaffordRandFixedSum. These algorithms run faster but the generation rate is reduced significantly. This could be due to a reduction in the distribution of generated U.

Applying multi-threading programming does not improve the performance of the generating process, even through the total CPU consumption is maintained constantly at 8%. The code seems to be bounded by other factors instead of computation only. Doing some research, the performance of the single-threaded process and the multi-threaded process will be the same in Python and this is because of Global Interpreter Lock (GIL) in Python.

Eventually, I found out that:

- By uniformly choosing the distribution for U_i in every tasks instead of every taskset (as mentioned in the paper [1])
- And adding more granularity level in the U_i's distributions (using bimodal distribution and exponential distribution in range [0.1 ~ 0.9] instead of [0.1, 0.3, 0.5, 0.7, 0.9]). This will reduce the generation time on large n.

This change will help the generating fix sum utilization not stuck in the try and discard sequences. My test shows that the time to generate random taskset is reduced by **x10** times.

4.2 Experimental Results

- Generating Random Taskset

	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)
# Taskset	104,184	244,608	81,134	35,538	20,866
# Tasks	416,736	1,223,035	486,804	248,766	166,928
Speed (taskset/10s)		26.44ts/10s			0.05ts/10s

Test environment: Intel(R) Core(TM) i7-10850H CPU @ 2.70GHz

- Exhaustive search for premier assignment

	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,8)*
Speed	14s/100ts	2min/100ts	5min/100ts	40min/100ts	6hrs/100ts (Not completed all the taskset yet)	2hrs/100ts (*) pseudo premier priority) (Not completed all the taskset yet)

- Training & Testing

Due to the limited time of this project, I cannot test on all the tasksets. I did implement the training and testing process and test with the taskset (2,4).

The pointer network is setups as follows: encoder/decoder built with LSTM [28] cells with 512 hidden units trained with learning rate of 0.001, batch size of 512, Adam optimizer, sparse categorical cross-entropy was used as a loss function. The number of training samples for each (m, 2) are 100,000 and test samples are 4,184.

5. My conclusions and future work.

With the limited time of this project, I almost completed the implementation of PAL model. The remaining parts are data augmentation and combing all the biased and unbiased taskset for the large n training samples.

For the project's question:

Why with big m and n, PAL seems to be decreased and cannot keep up with the ZLL DaBu?

The reasons for this issue could be:

- PAL is trained with exhaustive search for taskset with small m and n.
- For the taskset with large m and n, PAL relies on the smaller m and n taskset. However, these smaller tasksets are inferred by PAL as well. So, PAL may be lost its generality over the increasing of m and n.

Because all the premier priority assignment for large taskset (2,8) are not yet fully identified, therefore I do not have enough data to do additional tests for confirming my assumption.

There are several ideas for the future works:

a. Using the technique mention in section 3.2.2 to generate training samples for taskset (2,8)
b. Train and test the PAL model with taskset (2,8)
c. Exhaustive search for premier priority assignment of taskset (2,8)
d. Compare test result in step b) and c)
e. Inject some of tasksets in step c) into tasksets in step a) and re-train PAL model. Comparing test result of this trained model with the one in step b) to see if the test accuracy can be increased significantly. This can be used to answer the project's question.
f. Using an improved version of RTA-LC scheduability test [18]. Per my testing, this improved version provides a better estimation of the task's response time. By applying this scheduability test, a better taskset can be found but it might take more time during the taskset generation process.

References

[1] S. Lee, H. Baek, H. Woo, K. G. Shin and J. Lee, "ML for RT: Priority Assignment Using Machine Learning," 2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS), 2021, pp. 118-130, doi: 10.1109/RTAS52030.2021.00018.

[2] M. Bertogna, M. Cirinei, and G. Lipari, “Schedulability analysis of global scheduling algorithms on multiprocessor platforms,” IEEE Transactions on Parallel and Distributed Systems, vol. 20, pp. 553–566, 2009.

[3] R. Davis and A. Burns, “Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems,” Real-Time Systems, vol. 47, pp. 1–40, 2011.

[4] N. Audsley, “Optimal priority assignment and feasibility of static priority tasks with arbitrary start times,” Department of Computer Science, University of York, Tech. Rep. YCS164, 1991.

[5] M. Bertogna and M. Cirinei, “Response-time analysis for globally scheduled symmetric multiprocessor platforms,” in Proceedings of IEEE Real-Time Systems Symposium (RTSS), 2007, pp. 149–158.

[6] N. Guan, M. Sitgge, W. Yi, and G. Yu, “New response time bounds for fixed priority multiprocessor scheduling,” in Proceedings of IEEE Real-Time Systems Symposium (RTSS), 2009, pp. 387–397.

[7] N. Guan, M. Han, C. Gu, Q. Deng, and W. Yi, “Bounding carry-in interference to improve fixed-priority global multiprocessor scheduling analysis,” in Proceedings of IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2015, pp. 11–20.

[8] Q. Zhou, G. Li, and J. Li, “Improved carry-in workload estimation for global multiprocessor scheduling,” IEEE Transactions on Parallel Distributed Systems, vol. 28, no. 9, pp. 2527–2538, 2017.

[9] C. Liu and J. Layland, “Scheduling algorithms for multi-programming in a hard-real-time environment,” Journal of the ACM, vol. 20, no. 1, pp. 46–61, 1973.

[10] M. Bertogna, “Real-time scheduling for multiprocessor platforms,” Ph.D. dissertation, Scuola Superiore Sant’Anna, Pisa, 2007.

[11] B. Andersson and J. Jonsson, “Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition,” in Proceedings of IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2000, pp. 337–346.

[12] D.-T. Peng and K. G. Shin, “A new performance measure for scheduling independent real-time tasks,” Journal of Parallel and Distributed Computing, vol. 19, no. 1, pp. 11–26, 1993

[13] J. Lee, A. Easwaran, and I. Shin, “Maximizing contention-free executions in multiprocessor scheduling,” in Proceedings of IEEE Real-Time Technology and Applications Symposium (RTAS), 2011, pp. 235–244.

[14] P. Emberson, R. Stafford, and R. I. Davis, “Techniques for the synthesis of multiprocessor tasksets,” in International Workshop on Analysis Tools and Methogologies for Embedded and Real-Time Systems, 2010, pp. 6–11.

[15] R. Davis and A. Burns, On optimal priority assignment for response time analysis of global fixed priority pre-emptive scheduling in multiprocessor hard real- time systems,” Department of Computer Science, University of York, Tech. Rep. YCS-2009-451, 2010.

[16] Pointer Network source code https://github.com/keon/pointer-networks

[17] Schedcat Python library https://github.com/brandenburg/schedcat

[18] Q. Zhou, G. Li and J. Li, "Improved Carry-in Workload Estimation for Global Multiprocessor Scheduling," in IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 9, pp. 2527-2538, 1 Sept. 2017, doi: 10.1109/TPDS.2017.2679195.