Artifact Evaluation for Dependency Graph Approach for Multiprocessor Real-Time Synchronization

Jian-Jia Chen, Georg von der Brüggen, Junjie Shi and Niklas Ueter
TU Dortmund University, Germany

August 24, 2018

Contents

1 Introduction and General Information

The evaluations presented in the paper *Dependency Graph Approach for Multiprocesser Real-Time Synchronization* are based on algorithmic experiments and scheduler implementations with the associated evaluations in LITMUS^{RT}. More precisely, the evaluations in that paper include the following four parts:

a) Comparisons of the resulting makespan for different scheduling algorithms in the setting of frame-based task sets.

b) Acceptance Ratio Experiments of different approaches in the setting of frame-based task sets.

c) Acceptance Ratio Experiments for different approaches in the setting of periodic task sets.

d) Implementation and evaluation of the Dependency Graph Approach (with respect to scheduling overheads) in LITMUS^{RT}.

In general, the source files are documented. In this document we intend to give an overview of the provided software, general dependencies and data structures.

Moreover, in order to simplify the evaluation process for the user, we prepared a *linux-image* (Ubuntu 16.04, 64 bits) that contains all relevant dependencies and python scripts for evaluating the experiments a)-c). [1]

```
https://drive.google.com/file/d/11NORrCj72If8RjB8XWZb0mB7GiSyCPgQ/
view?usp=sharing
```

The username and password for the virtual machine are both **dgam**. Please notice that our evaluations are computationally intensive and thus incur **long runtimes**. For the evaluations of the first three parts a)-c), we used a cache-coherent SMP consisting of one 64-bit Intel Processor i5-7200U running at 2.5 GHz and 16 GB of main memory. All statements about computation times made in this document (if not otherwise stated) refer to this setup. We provide input data sets for each figure shown in the paper *Dependency Graph Approach for Multiprocesser Real-Time Synchronization* and propose the users to use these inputs in order to reproduce the results fast. It is however possible and encouraged to generate or reproduce new input and output data sets if sufficient time is available.

Another repository for the patch of LITMUS^{RT} and the evaluations in LITMUS^{RT} can be accessed under:

```
https://github.com/Strange369/Dependency-Graph-Approaches-for-LITMUS-RT.git
```

Furthermore, we used the third-party software called **SET-MRTS** (Schedulability Experimental Tools for Multiprocessors Real Time Systems https://github.com/RTLAB-UESTC/SET-MRTS-public) to compute acceptance ratios for various state-of-the-art resource locking protocols that are shown in the evaluations.

2 Algorithmic Evaluations

In this section, we describe the experiments setup, i.e., what algorithms are evaluated, the structure of our software, important data structures, and how to run, modify, or create experiments. Due to the pre-existing codebase of some schedulability algorithms, tests, and use of external software to run schedulability tests, we were forced to work with different data strcutures that may have to be converted at different stages of the evaluation. We will refer to these three different formats in more detail whenever it is required for the users understanding.

2.1 Folder structure

The scripts are organized into the four folders *algorithms*, *experiments*, *generator*, and *SET-MRTS* that are described in their respective sections.

[1] Just in case, we provide alternative way for downloading this image. Due to the large size of the image, we split it into five sub-files under:
https://depot.tu-dortmund.de/z3ha3
https://depot.tu-dortmund.de/6496r
https://depot.tu-dortmund.de/hahds
https://depot.tu-dortmund.de/6ypa3
https://depot.tu-dortmund.de/7mvtn
After downloading all of these files, please execute *'cat DGAM-AE.vdi.tar.bz.* > DGAM-AE.vdi.tar.bz'* to compose them back, and *'tar -xvj DGAM-AE.vdi.tar.bz'* to extract it.

2.1.1 Algorithms

The algorithms folder contains the files *'construction.py'*, *'rop.py'*, *'list_sched.py'*, and *'semi_fed.py'*. The functions in *'construction.py'* provide the algorithms to construct dependency-graphs from given tasksets according to Potts Algorithm [5] as well as Extented Jackson's Rule [2], and to compute the makespan of a given schedule. Further, it contains methods to compute lower bounds of the longest path of a DAG. The file *'list_sched.py'* contains files to generate various variants of list scheduling namely, Semi-Partitioned Non-Preemptive-, Partitioned Non-Preemptive-, Semi-Partitioned Preemptive-, and Partitioned Preemptive Scheduling are provided. Moreover, the tied partitioned algorithm described by Jinghao Sun et al. [6] is implemented here. The file *'rop.py'* contains the Resource-Oriented Partitioning (with release-enforcement) algorithms [3, 7]. The file *'semi_fed.py'* contains the semi-federated scheduling algorithm $S[X+1]$ for parametric DAG tasks [4]. Further documentation for these methods are provided in the source files, but are not mandatory to be studied in order to carry out the experiments.

2.1.2 Generator

This folder contains python and shell scripts to generate task sets used for the makespan calculation and frame-based task sets for schedulability tests (acceptance ratios) according to user specific configurations.

- **tasksets_makespan_generate.sh:** Generates task sets with 100% utilization for each processor. These task sets are solely used for the makespan experiments.
- **tasksets_generate_frame.sh:** Generates task sets with different utilizations (30% to 100% in 5% steps). Each task's period is set to 1 in the schedulability test (acceptance ratio) experiments.

Due to the usage of third-party software, we have to maintain different formats of the input task sets. To keep the consistency, all the schedulability test experiments are based on the same task set format.

- **convert_to_frame_rop.sh:** Converts the frame based task sets to the format which can be executed by the ROP script.
- **convert_frame_to_periodic.sh:** Adds different periods to frame based tasks. All tasks that share the same resource are assigned the same period.
- **convert_to_periodic_rop.sh:** Converts the periodic task sets to the format which can be executed by the ROP script.
- **convert_to_set_mrts.sh:** Converts the periodic task sets to the format which can be recognized by the SET-MRTS tool.

All the generated/converted task sets are stored in the inputs folder by the user given identifier.

2.1.3 SET-MRTS: How To

Copy the converted inputs from the folder *'experiments/inputs/input_paper_periodic_set_mrts/'* in *csv* format to the folder *src*, and run:

```
./STJJC set_mrts_n80_m100_p8_r8_s0.4_l0.5.csv
```

Due to the computational complexity of *linear programming*, the evaluations are computationally intensive and may take **up to three days** to complete (in our hardware platform). After completing the experiments, the results are written to the *results* folder.

The results can be converted to the base format format by using *'csv_results_convertor.py'* and by copying the results back to the folder *'experments/outputs/output_paper_periodic_set_mrts'*.

2.1.4 Experiments

In the *experiments* folder, we have prepared several bash scripts that are used to run the evaluations on pre-specified input data sets in parallel. All experiments read data from the *inputs* folder and store their results in a corresponding folder within the *outputs* folder. All the figures used in our paper are stored in the folder *figures*.

Makespan

- **makespan_frame.sh** For the input (frame-based task sets), a dependency-graph is constructed for each resource according to either Potts or Extended Jackson's Rule. Then the makespan for these dependency-graphs is calculated for the different scheduling algorithms in *'list_sched.py'*. We have two different partitioned algorithms, one is from [6], another is a heuristic algorithm to balance the work load among processors.

For drawing the figures under the two different partitioned algorithms in Section 6, please use the associated python file *'draw_makespan_hs.py'* (resulting in Fig. 3 presented in the paper) or *'draw_makespan_sun.py'* (not presented in the paper since it performed worse).

Acceptance Ratio - Frame-based

- **sched_dag.sh** For the input (frame-based task sets), a dependency-graph is constructed for each resource according to either Potts or Extended Jackson's Rule. Then the acceptance ratios for these dependency-graphs are calculated using Semi-Partitioned Preemptive scheduling.
- **sched_rop.sh** For the input (frame-based task sets), the acceptance ratio is computed using the ROP Algorithms in *'rop.py'*.

For drawing the results, please use the associated python file *'draw_sched_frame.py'*. The resulting figures relate to Fig. 4 in the paper.

Acceptance Ratio - Periodic

- **SET-MRTS** For the input (periodic task sets), the acceptance ratio is computed using any of the chosen resource synchronization protocols, e.g., LP-PFP-DPCP, LP-GFP-FMLP, LP-GFP-PIP, and GS-MSRP (see paper).
- **sched_periodic_rop.sh** For the input (periodic task sets), the acceptance ratio is computed using the ROP Algorithms in *'rop.py'*.
- **sched_sfed.sh** For the input (periodic task sets), a dependency-graph is constructed for each resource according to either Potts or Extended Jackson's Rule. Based on these dependency-graphs the parametric DAG task model $(vol(G), len(G), T)$ is computed and evaluated using the semi-federated scheduling test in *'semi_fed.py'* for the acceptance ratio.

In order to draw the results, you may use the associated python file *'draw_sched_periodic.py'*. The resulting figures relate to Fig. 5 in the paper.

Own Experiments

If the users would like to create new experiments, they should copy one of above scripts and change the inputs as well as algorithms according to the documentation in the source files. It is also possible to regenerate and to redo all the aforementioned experiments, but the users should take care of the folders' and the files' names, and change the input formats accordingly.

3 LITMUS^{RT} Evaluations

For the evaluation of our DGA implementation in LITMUS^{RT}, we use a cache-coherent SMP consisting of two 64-bit Intel Xeon Processor E5-2650Lv4 running at 1.7 GHz with 35 MB cache and 64 GB of main memory. Since an accurate timing behaviour can not be provided in a virtual machine, we recommend the user to prepare a physical machine to verify and reproduce the stated results, with respect to scheduling overheads, presented in the paper.

We use Ubuntu 14.04.5 Server (64-bit) and LITMUS^{RT} (2016) as our operating system (Desktop version Ubuntu 14.04.5 works as well) to evaluate our DGA Scheduling algorithm. In the following, we provide all required steps to install LITMUS^{RT}and our implemented scheduling algorithms.

```
sudo su
cd $DIR
// Download linux kernel
wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.1.3.tar.gz
tar xzf linux-4.1.3.tar.gz
// Download litmus-rt patch
wget http://www.litmus-rt.org/releases/2016.1/litmus-rt-2016.1.patch
mv linux-4.1.3 litmus-rt
cd litmus-rt
// Apply litmus-rt patch
patch -p1 < ../litmus-rt-2016.1.patch
// Apply our patch here as well
patch -p1 < ../litmus-dga.patch
// Put configure_dga inside the folder litmus-rt
// And change the name
mv configure_dga .config
// Check the configuration
sudo apt-get install libncurses5-dev
make menuconfig
https://wiki.litmus-rt.org/litmus/InstallationInstructions
// Compile bzImage and modules
make
make modules_install
make install
sudo reboot
//go to GRUB
//choose the litmus-rt kernel (May be named as linux 4.1.3)

//install the lib and tools
cd $DIR
wget http://www.litmus-rt.org/releases/2016.1/liblitmus-2016.1.tgz
tar xzf liblitmus-2016.1.tgz
cd liblitmus
wget http://www.litmus-rt.org/releases/2016.1/liblitmus-config
mv liblitmus-config .config
// Apply our lib patch here
patch -p1 < ../lib-dga.patch
make
cd $DIR
wget http://www.litmus-rt.org/releases/2016.1/ft_tools-2016.1.tgz
tar xzf ft_tools-2016.1.tgz
cd ft_tools
wget http://www.litmus-rt.org/releases/2016.1/ft_tools-config
mv ft_tools-config .config
make
cd $DIR
// Change the path, so that you can use the command anywhere you want.
vim .bashrc
PATH=\$HOME/ft_tools:\$HOME/liblitmus:\$PATH
```

After the installation of LITMUS^{RT}go to our *litmusrt-eval* folder and execute the bash script *'overhead.sh'* with *root* permission as follows.

```
root@dgam:~/litmusrt-eval$ ./overhead.sh
```

Enter the plug-in name (here: P-FP Partitioned Fix Priority) and a trace name, which will start the tracing device. Afterwards, open another terminal and execute the bash script that is named like the protocol you intend to evaluate. That is if you want to trace the overhead of Partitioned Dependency Graph Approach (pdga) then you can execute the following:

```
root@dgam:~/litmusrt-eval$ ./pdga.sh
```

When the virtual tasks finished their execution go to the terminal in which you started the tracing devices and press *'Enter'* to stop the tracing. To analyze the traces, use the following steps[1].

```
// (1) Sort
ft-sort-traces overheads_*.bin 2>&1 | tee -a overhead-processing.log
// (2) Split
ft-extract-samples overheads_*.bin 2>&1 | tee -a overhead-processing.log
// (3) Combine
ft-combine-samples --std overheads_*.float32 2>&1 | tee -a overhead-processing.log
// (4) Count available samples
ft-count-samples  combined-overheads_*.float32 > counts.csv
// (5) Shuffle & truncate
ft-select-samples counts.csv combined-overheads_*.float32 2>&1
| tee -a overhead-processing.log
// (6) Compute statistics
ft-compute-stats combined-overheads_*.sf32 > stats.csv
```

Please notice, most of the results in stats.csv are recorded by cycles rater than seconds.

References

[1] M. G. Björn B. Brandenburg and M. Vanga. A Tour of LITMUS-RT. `http://www.litmus-rt.org/tutorial/manual.html#overheadtracing`.

[2] L. A. Hall and D. B. Shmoys. Jackson's rule for single-machine scheduling: Making a good heuristic better. *Math. Oper. Res.*, 17(1):22–35, 1992.

[3] W.-H. Huang, M. Yang, and J.-J. Chen. Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share? In *Real-Time Systems Symposium (RTSS)*, pages 111–122, 2016.

[4] X. Jiang, N. Guan, X. Long, and W. Yi. Semi-federated scheduling of parallel real-time tasks on multiprocessors. In *Proceedings of the 38nd IEEE Real-Time Systems Symposium, RTSS*, 2017.

[5] C. N. Potts. Analysis of a heuristic for one machine sequencing with release dates and delivery times. *Operations Research*, 28(6):1436–1441, 1980.

[6] J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi. Real-time scheduling and analysis of OpenMP task systems with tied tasks. In *IEEE Real-Time Systems Symposium, RTSS*, pages 92–103, 2017.

[7] G. von der Brüggen, J.-J. Chen, W.-H. Huang, and M. Yang. Release enforcement in resource-oriented partitioned scheduling for multiprocessor systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS*, pages 287–296, 2017.