Tools for Simulating Type-Aware Federated Scheduling for Typed DAG Tasks on Heterogeneous Multicore Platforms

Junjie Shi and Jian-Jia Chen
Informatik Centrum Dortmund e.V. , Germany

June 15, 2023

Contents

1 Introduction and General Information

The **TypedDAG_Sim** is developed as a simulator for simulating the schedule of typed DAG tasks on heterogeneous multicore platforms, that is available in [1].

For a typed DAG task, besides the precedence constraints of DAG structure, each vertex has been allocated to a specified type of cores, i.e., the vertex has to be executed on that dedicated type of cores. Such an additional partitioning constraint makes common DAG task analysis invalid. To this end, several federated scheduling base approaches have been proposed and evaluated, where each DAG task is partitioned to a subset of available cores, and all the jobs that released by the DAG task has to be executed on these cores. In the traditional federated schedule approach, tasks are divided into two categories, i.e., *heavy* and *light*. For a *heavy* task, a subset of both types of processors are exclusively assigned, no other task can access these processors. For a *light* task, it shares one core from each type with other tasks. In the type-aware federated schedule in [7], an additional category is introduced, i.e., *semi-heavy*. For a *semi-heavy* task, it only exclusively requires one type of processors, but share another type of processors with other tasks. However, none of them are considering data accessing time for vertex execution, which has been taken into consideration in this simulator.

In general, to simulate the schedule of a given set of DAG tasks on a given heterogeneous multicore platform using federated schedule approach, a task-to-core mapping has to be generated according to an applied federated scheduling algorithm. Afterwards, the schedule detail of the task set is generated using discrete event driven simulation. Besides the execution time of each task, the developed *TypedDAG_Sim* simulator also considers the data accessing time by including a customized memory hierarchy.

The structure of the tool is shown in Figure 1. More precisely, the simulator includes the following parts:

1. *generators*: a) generate a unified configuration file according to these customized settings, i.e., `configuration.json`; b) generate the task sets with DAG structure, typed allocation, and data requests according to the configuration file.

2. *algorithms*: a) two schedulability test algorithms based on normal federated schedule approach [4, 5] and type-aware federated schedule approach [7] are applied to find the optimized task-to-core mapping with the objective of finding the minimal required number of cores; b) a raw type aware list scheduling algorithm, that list schedule is applied on each type of cores independently; c) memory hierarchy to simulate the memory accessing time for each vertex, where least recently used cache update policy is applied as well; d) the kernel of the simulator, to simulate the schedule of a given task set along with the task-to-core mapping and record the deadline misses.

3. *experiments*: a) the script to generate the task-to-core mapping of a given task set. If the worst case execution time (WCET) is not feasible, average case execution time (ACET) will be evaluated. In case the current available number of cores are not feasible, the least required number of cores are investigated; b) the script to simulate the detailed schedule is simulated by considering the real case execution time (RCET) and data accessing time; c) a draw function can be applied to draw the gantt chart of the simulated schedule.

[1] `https://github.com/JJShi92/TypedDAG_Simulator_public`

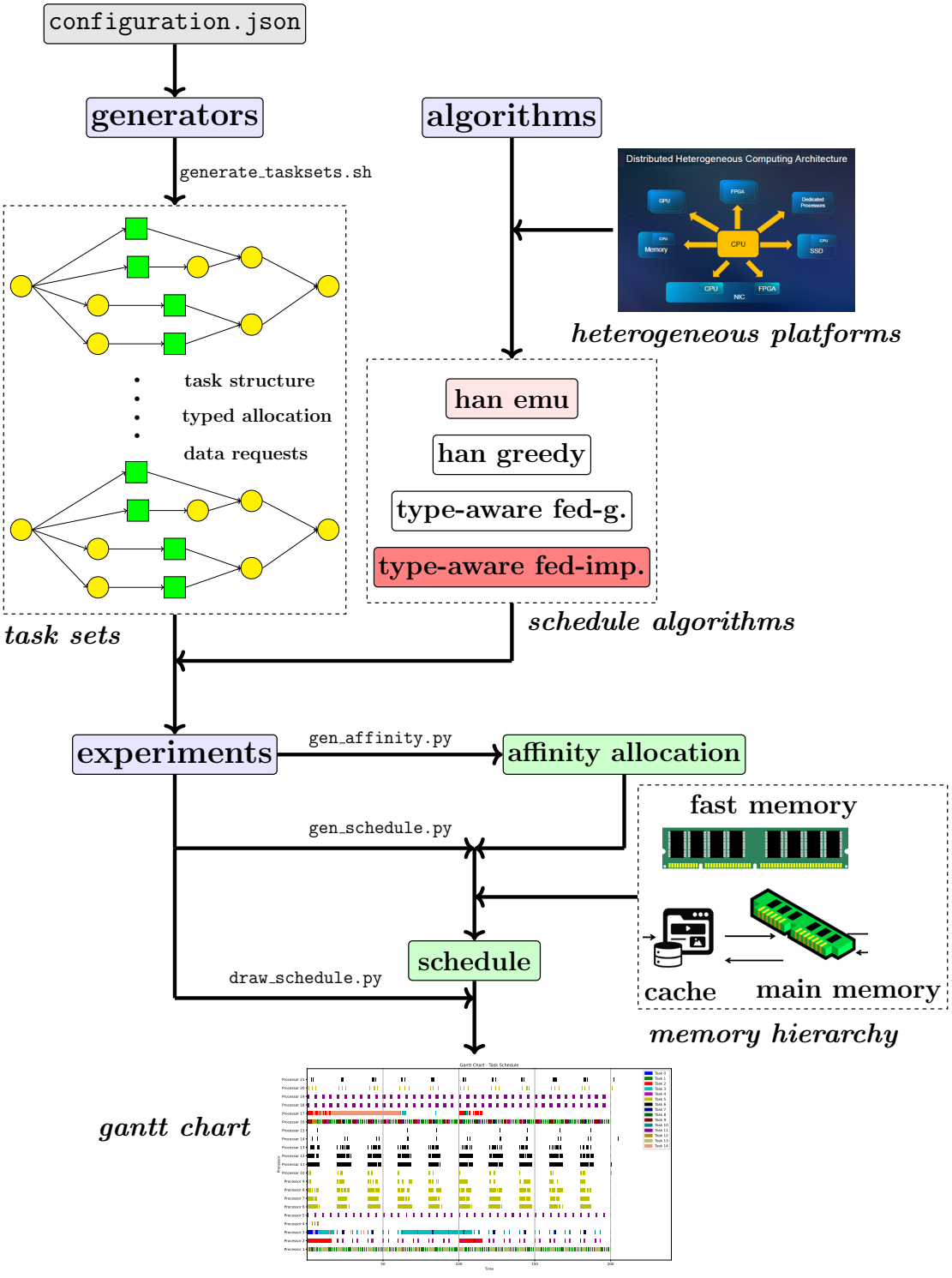

Figure 1: The structure of the proposed simulator.

2 Algorithmic Evaluations

In this section, we describe the experiments setup and folder structure in detail. The scripts are organized into the three folders *generator*, *algorithms*, and *experiments* that are described in their respective sections.

2.1 Pre-requests

Our tool requires *python* version 3.8 or higher and *numpy* version 1.24 or higher.

Before starting, the Dirichlet-Rescale (DRS) algorithm [2] is applied to generate utilizations of task sets randomly.

```
pip3 install drs
```

2.2 Generator

In generators folder contains `'configuration_generator.py'`, `'read_configuration.py'`, `'generator_pure_dict.py'`, `'typed_core_allocation.py'`, `'data_requests.py'`, `'tasksets_generator_pure.py'`, `'tasksets_generator_typed.py'`, `'tasksets_generator_data_requests.py'`, `'generate_tasksets.sh'`, and `tasksets_input_convertor.py`.

2.2.1 Configuration

Users can open `'configuration_generator.py'` to customize these supported configurations, all the available configurations are shown in Table 1. Once the configuration has finished, execute to following command, a unified configuration file, i.e., `configuration.json` will be created.

```
pythons configuration_generator.py
```

The `'read_configuration.py'` is applied to read the configurations from a given `configuration.json` file, which is called by other scripts.

2.2.2 DAG Task Generator

The `'generator_pure_dict.py'` is the algorithmic file for generating DAG task structure, e.g., vertices with weights, precedence constraints between vertices, period, deadline, and other useful information.

The `'tasksets_generator_typed.py'` is the script to read the configuration and generate detailed DAG tasks and store to `/experiments/inputs/tasks_pure/` by executing:

```
pythons tasksets_generator_pure.py
```

The default configuration is the `configuration.json` generated from Section 2.2.1. Users can also import an independent configuration file by:

```
pythons tasksets_generator_pure.py -i customized_conf_file.json
```

The designed total utilization for each task set is defined by the number of all cores from both types times the designed average utilization per core, i.e., $U_{tot} = (aprocessor + bprocessor) \times U_{avg}$. The period of each task is selected randomly from a set of semi-harmonic periods, i.e., $T_i \in \{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$, that is used in automotive systems [3, 6, 8–10]. In addition, the lower bound of execution time for each vertex is defined by the allowed preempt_times multiple with the maximum data accessing time, i.e., $U_{lb} = \frac{preempt\_times \times main\_mem\_time}{T_i}$. Afterwards, the utilization for each task and utilization for each vertex is uniformly and randomly generated by DRS algorithm. The $G(n, p)$ algorithm [1] is used to generate the edges between nodes, with the given probability, i.e., pc_prob. The feasibility of each DAG task is checked by calculating the length of the critical path, i.e, if the length of the critical path is longer than its period (cannot be schedulable by default), the DAG task will be regenerated.

2.2.3 Typed Core Allocation

The `'typed_core_allocation.py'` is the algorithmic file to generate typed allocation for each vertex. Please note, only two different types of cores are supported. The `'tasksets_generator_typed.py'` is the script for generating the detailed typed information and storing to `/experiments/inputs/tasks_typed/` by executing:

Parameter	Description
aprocessor	Number of processor A
bprocessor	Number of processor B
msets	Number of sets
ntasks	Number of tasks for each set. If $ntasks < 1$, the number of tasks is generated randomly due to the sparse parameter
sparse-0	Range: $[0.5 \times \max(aprocessor, bprocessor), 2 \times \max(aprocessor, bprocessor)]$
sparse-1	Range: $[(aprocessor + aprocessor), 2 \times (aprocessor + aprocessor)]$
sparse-2	Range: $[0.25 \times (aprocessor + aprocessor), (aprocessor + aprocessor)]$
pc_prob	Lower and upper bounds of the probability of two vertices having an edge. The real probability is in the range $[pc\_prob\_l, pc\_prob\_h]$
utilization	Average per core utilization for a set of tasks
scale	Scale to keep all the parameters as integers
skewness	Controls the skewness of the skewed tasks
per_heavy	Percentage of heavy^a or heavy^b tasks (e.g., 0%, 25%, 50%, 75%, and 100%)
one_type_only	Whether to allow a task to require only one type of processor: 0 (not allowed) or 1 (allowed)
num_data_all	Number of all available data addresses
num_freq_data	Number of frequently requested data addresses
percent_freq	Percentage of requesting the frequently requested data
allow_freq-0	Generate requested data randomly regardless of the frequently requested data
allow_freq-1	Allow to control the percentage of frequently requested data
main_mem_size	Size of the main memory. Assume a very large number can store all the requested data
main_mem_time	Time for accessing data from main memory
fast_mem_size	Size of the fast memory
fast_mem_time	Time for accessing data from fast memory
l1_cache_size	Size of the L1 cache
l1_cache_time	Time for accessing data from L1 cache
preempt_times	The allowed preempted time for each vertex, once the preempt_time is reached, the vertex becomes non-preemptive even by vertex has higher priority
try_avg_case	Boolean value to indicate that whether to try average case execution time (acet) when the WCET cannot pass the schedulability test
avg_ratio	Ratio of acet/wcet
min_ratio	Ratio of best case execution time/wcet
std_dev	Standard deviation for generating real case execution time when simulating the schedule
tolerate_pa	Upper bound of the tolerable number of type A processors when the current number of processor A is not enough
tolerate_pb	Upper bound of the tolerable number of type B processors when the current number of processor B is not enough
rho_greedy	Setting of rho for the greedy type-aware federated schedule algorithm. $0 < \rho \leq 0.5$ (default 0.5)
rho_imp_fed	Setting of rho for the improved type-aware federated schedule algorithm. $0 < \rho \leq 1/7.25$ (default $1/7.25$)

Table 1: All the supported configurations and the corresponding descriptions.

```
pythons tasksets_generator_typed.py
```

The script can also read a customized configuration file with the `-i` argument vector.

2.2.4 Data Requests

The `'data_requests.py'` is the algorithmic files for generating the required data address of each vertex respectively. The `'tasksets_generator_data_requests.py'` is the script file to generate detailed requested data according to the configuration file and store to `/experiments/inputs/tasks_data_request` by executing:

```
pythons tasksets_generator_data_requests.py
```

The script can also read a customized configuration file with the `-i` argument vector.

2.2.5 Batch Generation

The bash script, i.e., `'generate_tasksets.sh'` can be applied to generate all the aforementioned task set related information uniformly according to the configuration file. All these generated files are stored under the path `/experiments/inputs`, by executing:

```
./generate_tasksets.sh
```

In order to avoid file or processing chaos, an archive process is operated before task generation, i.e., the configuration file (`configuration.json`) `/experiments/inputs` and `/experiments/outputs` folders are moved to the **/archive** folder, that is named by the date and time.

2.3 Customized Inputs

We also provide one converter, named `tasksets_input_convertor.py`. It can convert a customized input task set in 'json' format to the 'numpy' format, that is used in this framework, by executing
(Please replace `input_task_set.json` as the real file name):

```
python tasksets_input_convertor.py -i 'input_task_set.json'
```

Besides the input json file, the configuration file also needs to be customized for further experiments. Please note, the current converter can only handle one task set and one utilization level at one time. The input json file requires each task has `'task_id'`, `'period'`, `'deadline'`, and `'vertices'`. For `'vertices'`, it contains several vertices, each of them has to define `'execution_time'`, `'successors'`, `'requested_data_address'`, `'core_type'`.

2.4 Algorithms

The algorithms folder contains the files `'affinity_han.py'`, `'affinity_improved_fed.py'`, `'affinity_raw.py'`, `'memory.py'`, `'misc.py'`, and `'sched_sim.py'`.

The simulator provides three different task-to-core mapping algorithms with different schedulability guarantee:

- The `'affinity_han.py'` is the normal federated schedule approach, where the schedulability test was presented in [4, 5]. It supports both *emu* and *greedy* approach for allocating heavy DAG tasks, i.e., a task's total utilization is higher than 1.
- The `'affinity_improved_fed.py'` is the new proposed type-aware federated scheduling algorithm in [7], where both *greedy* and *improved* task allocation approaches are supported.
- The `'affinity_raw.py'` is the *raw* affinity allocation approach, where the type aware listing schedule algorithm is applied. That is, each vertex can be executed on any cores from the defined type without any response time guarantee.

All the above three affinity allocation approaches output the task-to-core mapping in dictionary format.

The `'memory.py'` is the memory hierarchy file, which supports a two layers memory, i.e., cache and main memory along with independent fast memory, where the sizes and accessing times can be customized according to the configuration file. The least recently used (LRU) cache update policy is supported.

The `'misc.py'` contains some functions that are utilized widely in the simulator.

The 'sched_sim.py' is the schedule simulator kernel, which supports rate monotonic (RM) scheduling by discarding vertices that will potentially miss their deadlines. The simulator also supports the earliest deadline first (EDF) schedule as a demonstration for extensibility. Please note, only typed_dag_schedule_rm_discard_sim has the response time guarantee. The results contains the execution details, i.e., task id, vertex id, processor id, and duration, and deadline misses details, i.e., task id, vertex id, time slot when deadline missing.

2.5 Experiments

The experiments folder contains the files 'gen_affinity.py', 'gen_schedule.py', view_affinity.py, and 'draw_schedule.py'.

The 'gen_affinity.py' is trying to find the optimized affinity, i.e., task to core mapping, for a given task set. Both han's normal federated schedule approach and type-aware federated schedule approach are tried, the approach can arrange the task set on less number of cores are selected. Due to the performance advantage, we only apply *emu* allocation for han's approach and improved type-aware federated schedule approach in the simulator setting. The greedy allocation for han's approach and greedy type-aware federated schedule approach are also supported in the simulator. For greedy allocation for han's approac, users only need to replace the last argument of han.sched_han as 0. For greedy type-aware federated schedule approach, users can replace the imp_fed.improved_federated_p3 as imp_fed.greedy_federated_p(task_set, typed_org, rho_greedy, aprocessor, bprocessor).

The WCET of the given task set is evaluated at first, if the current available number of cores can be feasible, the results will be returned. Otherwise, average case execution time (ACET) will be evaluated. At the mean time, the task set with WCET will be evaluated by considering the maximum tolerable number of cores to find the minimal number of cores to satisfy the task set. If at least one of the aforementioned tries is feasible, the feasible results are returned and stored. If both tries fail, the task set with ACET will be deployed on the maximum tolerable number of cores. In the worst case, if it fails as well. The raw affinity will be returned, where the type aware listing schedule algorithm is applied. That is, each vertex can be executed on any cores from the defined type without any response time guarantee. The generated affinities are stored into the folder /experiments/outputs/affinity_allocation

The 'gen_schedule.py' is applied to generate the schedule for a given task set and affinity allocation by applying rate monotonic preemptive scheduling algorithm. If a task set can have the affinity allocation with its WCET, all tasks can finish their execution before deadlines. However, if the affinity allocation is generated by ACET, deadline miss can happen. In typed_dag_schedule_rm_discard_sim, a ready queue checker is deployed, that all the vertex that can potentially miss its deadline will be discarded before selecting for executing. The generated schedule is stored in the folder /experiments/outputs/schedule.

Please note, the generator allows to generate several task sets with the same configurations, both 'gen_affinity.py' and 'gen_schedule.py' are applied for all sets of the given task sets file.

Since all the generated results files are in numpy format, which may is difficult for reading directly, the view_affinity.py script is prepared. The script can print out the required number of two types of cores, the detailed task-to-core mapping, and store the affinity information in json format in the same folder with the same name, by executing:

```
python view_affinity.py -i 'outputs/affinity_allocation/aff.npy'
```

The required number of cores for both types are explicitly presented, i.e., Processor A: required type A cores, Processor B: required type B cores. The task-to-core mapping is in a format: "task id": id, "Porcessor A id": [type A core ids], "Porcessor B id": [type B core ids]. In this framework, the type A core is counted from 0 to $required\ type\ A\ cores\ -\ 1$, and the type B core is counted from $required\ type\ A\ cores$ to $required\ type\ A\ cores\ +\ required\ type\ B\ cores\ -\ 1$.

To intuitively observe the schedule results, the 'draw_schedule.py' can be applied to draw the Gantt chart of the generated schedule. Besides reading the configuration file, we provide several customized argument parameters as well:

- -s: the set id, several sets may be generated with the same configuration, the set id can help to draw the schedule results from different sets. The default set id is 0.
- -u: the designed average utilization per core. The utilization in the configuration file is a list, several utilization levels can be evaluation at the same time. The default utilization is the first utilization value in the configuration file.
- -f: the schedule file name that to be drawn without considering the configuration file.

- `-t`: the maximum time bound for drawing. When the schedule is simulated by a very long time interval, draw the complete schedule using python may not be possible. Therefore a time bound has to be defined to bound the time interval for drawing the schedule into a Gantt chart. The default time bound is 200ms.

References

[1] P. Erdös. On random graphs I. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

[2] D. Griffin, I. Bate, and R. I. Davis. Generating utilization vectors for the systematic evaluation of schedulability tests. In *41st IEEE Real-Time Systems Symposium, RTSS 2020, Houston, TX, USA, December 1-4, 2020*, pages 76–88. IEEE, 2020.

[3] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst. Communication centric design in complex automotive embedded systems. In *Euromicro Conference on Real-Time Systems, ECRTS*, pages 10:1–10:20, 2017.

[4] M. Han, N. Guan, J. Sun, Q. He, Q. Deng, and W. Liu. Response time bounds for typed DAG parallel tasks on heterogeneous multi-cores. *IEEE Trans. Parallel Distributed Syst.*, 30(11):2567–2581, 2019.

[5] M. Han, T. Zhang, Y. Lin, and Q. Deng. Federated scheduling for typed DAG tasks scheduling analysis on heterogeneous multi-cores. *J. Syst. Archit.*, 112:101870, 2021.

[6] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmark for free. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[7] C. Lin, J. Shi, N. Ueter, M. Günzel, J. Reineke, and J. Chen. Type-aware federated scheduling for typed DAG tasks on heterogeneous multicore platforms. *IEEE Trans. Computers*, 72(5):1286–1300, 2023.

[8] A. Sailer, S. Schmidhuber, M. Deubzer, M. Alfranseder, M. Mucha, and J. Mottok. Optimizing the task allocation step for multi-core processors within autosar. In *2013 International Conference on Applied Electronics*, pages 1–6, Sept 2013.

[9] S. Tobuschat, R. Ernst, A. Hamann, and D. Ziegenbein. System-level timing feasibility test for cyber-physical automotive systems. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, May 2016.

[10] G. von der Brüggen, N. Ueter, J. Chen, and M. Freier. Parametric utilization bounds for implicit-deadline periodic tasks in automotive systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS*, pages 108–117, 2017.