8th International Conference on Computer and Knowledge Engineering (ICCKE 2018), October 25-26 2018,
Ferdowsi University of Mashhad

Semi-Federated Scheduling of Multiple Periodic Real-Time DAGs of Non-Preemptable Tasks

Masoud Shariati, Mahmoud Naghibzadeh, Hamid Noori
Department of Computer Engineering
Ferdowsi University of Mashhad
Mashhad, Iran
{mshariati, naghibzadeh, hnoori}@um.ac.ir

***Abstract*—Research on real-time scheduling of tasks having different periods has a long history. Passing from uniprocessors to multiprocessors, and from independent sequential tasks to models like DAGs, each was an important turning point causing the need for a heavy mass of new research work. In this research, we consider the problem of scheduling multiple periodic real-time DAGs of non-preemptable tasks on multiprocessor platforms. The DAGs are independent and the deadline of each is until the arrival of the next request for the same DAG. In the context of multiple periodic real-time DAGs scheduling, federated scheduling has been a very successful approach and in this paper we try to extend this idea. The key feature of the federated approach is the dedication of cores to DAGs. An important disadvantage of federated scheduling is that for each DAG a significant portion of its dedicated processing capacity may be wasted. The novelty of the current paper is the proposing of a new method for semi-federated scheduling of multiple DAGs and we try to remedy the mentioned disadvantage to some extent. This is done by compacting the scheduling of each DAG and also relaxing the requirement for scheduling all tasks of a DAG on its dedicated cores. In the experiments section, we compared the proposed semi-federated method with a federated method and significant improvements in success ratio and the number of used cores is achieved, e.g. for a system with 64 cores and a workload of 80 percent, success ratio of the federated method was 0 percent and success ratio of the proposed method was 90 percent.**

Keywords*—*real-time multiprocessor scheduling; multiple periodic DAGs; semi-federated scheduling

I. INTRODUCTION

As using multiprocessor systems becomes more widespread, modeling real-time problems having parallel potentials in forms like DAGs gains special importance [1], [2]. It makes the definition of deadlines less than total execution time of all tasks of a DAG possible and also provides more relaxed deadlines and more fine grained tasks for a better real-time scheduling.

Research on real-time scheduling of periodic tasks has a history of about a half of century [3]. Passing from uniprocessors to multiprocessors, and from sequential tasks to models like DAGs, each was an important turning point causing the need for a heavy mass of new research work. By the term “sequential task” we mean a task that should not be running on more than one core at any given time. Some other useful definitions can be found later in this section.

An inevitable challenge in scheduling of real-time systems is guaranteeing that no deadline miss will occur. Having different periods makes this difficult. Having multiple cores [3], precedence relations between tasks [4], and the requirement to have limited or no task preemptions [5], [6] are other conditions that make the scheduling more complex. Thus, in this research, the real-time scheduling problem is studied at a high level of abstraction, and some conditions that usually apply to specific real-world applications such as issues related to reliability of embedded applications [7] or problems with shared caches in multi-core systems [8] is not considered. This is what the other related works do also.

There are three approaches for scheduling a set of independent sequential tasks on multiprocessor systems: global, partitioned, and semi-partitioned. In global scheduling, all requested tasks get stored in a global ready queue which is shared among all cores. Every core, as soon as ready, can take, indiscriminately, the task with the highest priority for execution. A challenge with the global approach is that its safe utilization is not high and hence not attractive for practical systems [9]. The other problem with this approach is its excessive task migrations for the case of preemptive scheduling [5].

Partitioned scheduling methods map each task to only one core and have no migration overheads. But, this results in significantly low core utilizations. Researches on semi-partitioned scheduling of sequential tasks have produced very good results in facing with the mentioned problem. This approach tries to schedule each task on a specific core for most tasks and only few tasks are allowed each to be scheduled on multiple cores, in order to increase system utilization [6].

In recent years, with the introduction of federated scheduling, the concept of partitioned scheduling is extended for scheduling a set of DAGs [10], [11]. The key feature of federated methods is the dedication of cores to DAGs. The idea behind the federated approach is overrun-freeness for the life of the system which is a great advantage in simplifying the safeness analysis. But, with this approach, a significant portion of the dedicated processing capacity devoted to each DAG (the federated cores) may be wasted.

Our goal is to propose a semi-federated [12] method that tries to make use of fewer number of cores by avoiding the

978-1-5386-9569-2/18/$31.00 ©2018 IEEE

dedication of low utilization cores to DAGs. In fact, the semi-federated idea is an effort to extend the successful idea of semi-partitioned scheduling to multiple DAGs rather than multiple sequential tasks. In analogy with the federated scheduling, the semi-federated scheduling is called so because all tasks of each DAG are not restricted to run on the dedicated cores and it is possible to have few common cores among multiple DAGs. According to evaluation results, in comparison with a federated method, we show that significant improvements in success ratio and the number of used cores are achieved. To the best of our knowledge, this is the first semi-federated method that does not require the tasks to be preemptable.

For the sake of accuracy and conciseness, some of the notions used in this paper are summarized below as some definitions:

Definition 1: We call a task that should not be running on more than one core at any given time a *sequential task*.

Definition 2: The request interval of a sequential task or a DAG is referred to as its *period*. This interval is the same for all requests of it.

Definition 3: The *period* for a task of a DAG, is the same as the DAG's period.

Definition 4: The *critical-path length* for a DAG is sum of execution times of the tasks on its critical-path [13] computed using task execution times. For each task, the Worst-Case Execution Time (WCET) of the task is assumed to be known.

Definition 5: The *utilization* of a periodic task is its execution time divided by its period.

Definition 6: The *utilization* of a periodic DAG is sum of its tasks' execution times divided by its period. In other words, the utilization of a periodic DAG is sum of its tasks' utilizations.

Definition 7: The *utilization* of a core that is scheduled for a period, is sum of its scheduled tasks' execution times divided by the period.

Definition 8: We call the average utilization of the cores of the system, *system utilization*.

Definition 9: The *Success ratio*, when scheduling multiple sets of DAGs, is the ratio of the number of successfully scheduled sets with respect to the total number of sets. The scheduling of a set of DAGs is successful when it is guaranteed that the deadline of none of the requests of its DAGs will be missed.

Definition 10: In a set of DAGs, *Light DAGs* are DAGs with utilization less than 1. The other DAGs are called *Heavy DAGs*.

In section II, the related literature will be investigated in more depth. In section III, we present the details of our proposed method. Section IV, is devoted to comparison and evaluation. Finally, the summary and future work is discussed in section V.

II. Related Work

Generally speaking, there are three approaches to scheduling multiple DAGs of tasks [14]:

1. Decomposing DAGs of tasks to constrained sequential tasks in which each task has a release time and a deadline. This approach tries to transform the problem of scheduling periodic DAGs to traditional problem of scheduling periodic sequential tasks and use the existing algorithmic ideas to safely schedule these tasks [15], [16].
2. Global scheduling without decomposition in which all ready tasks of all DAGs are placed in a common priority queue to be picked up for execution [4], [10].
3. Federated scheduling in which to each heavy DAG, dedicated cores are assigned and none of the cores are shared between such DAGs [10], [11].

In the first approach, the recurrence period of each decomposed task of a DAG is the same as that of the DAG itself. The release time of a task within the DAG under scheduling is the earliest possible time that the task is ready with respect to the request time of the DAG itself.

The method presented in [15] is the most noted research on decomposition-based scheduling of multiple real-time DAGs. Another transformation method based on stretching each DAG schedule to its deadline is presented in [16] and further studied in [17] and [18]. The evaluation results of the method shows that it outperforms the method of [15]. But, as the number of cores increase, the performance of both methods drastically drops. This is clearly a major weakness because number of cores of multicore processors is growing higher as the technology advances. The more recent decomposition-based method of [14] is evaluated to be very effective, with respect to schedulability test, in comparison to the method of [15] and even the federated method of [10]. Since the approach presented in this paper is not decomposition-based and neither is it a DAG transformation method, the details of these three methods are not discussed here.

The federated approach for scheduling multiple periodic DAGs is an extension to the successful approach of partitioning for a set of periodic sequential tasks [10]. The focus of this approach is on dedication of cores required for safe running of DAGs to them. In such a method, for light DAGs a different route is followed than heavy DAGs. The latter DAGs each require at least one dedicated core. Core reduction becomes meaningful when there is more than 1 cores federated to run all tasks of the DAG, but this idea is not considered in their method. Also, we believe that the estimation of the number of dedicated cores for each of DAGs in their method is for the worst case and hence in most cases it leads to low core utilization. The formula for the mentioned estimation in that method is given in Formula (1) which is actually a result of the work by Graham [3].

$$n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil \qquad (1)$$

In Formula (1), C_i is total execution time of all tasks of the DAG, L_i is the length of the critical-path of the DAG, and D_i is the deadline of the DAG. Recall that, one of the reasons for modeling real-time tasks in the form of a DAG is to run some of its tasks in parallel and pave the way for imposing short deadlines. Actually, when deadline is taken to be equal to the length of critical-path, the formula suggests that infinite number of cores is needed.

The semi-federated approach to scheduling DAGs of real-time tasks is studied in [12] and in comparison with federated approach [10] a noticeable improvement with respect to success ratio is reported. Although, its schedulability test is also based on Formula (1) and we believe for offline scheduling, where there is not a need for a schedulability test

for online admission of DAGs, a far better performance could be achieved. another limitation of this approach is that tasks must be preemptable.

In the experiments section, our method is compared with the method of [10]. The comparison is not by using the schedulability test based on the pessimistic estimate of Formula (1), and the actual performance of both methods is used. In contrast to federated scheduling, the semi-federated method in [12] and the decomposition technique in [14] are not applicable to non-preemptable tasks and are not included in the experiments.

In most reviewed research studies, the weakness is drastic reduction of success ratio when the number of cores increases and also the total load increases proportionally. Figure 6 of reference [12] clearly shows this phenomenon. Fortunately, with the presented method in this paper, the reverse of this phenomenon happens. The other feature of the presented method is that preemption is not a must have property of tasks.

III. Proposed Method

The overall approach is to assign some of the needed cores to each DAG and let it share other needed cores with other DAGs. In federated scheduling, each heavy DAG gets all its needed cores for its own use only [10], [11]. The fact is that, even if we know the minimum number of required dedicated cores for a DAG to meet its deadline, the DAG structure and the behavior of the used scheduling algorithm for the DAG may be such that waste a significant portion of its dedicated processing capacity. As stated earlier, complete dedication is a disadvantage of the federated scheduling. Therefore, by assigning some tasks of Heavy DAGs to common cores, we provide the basis for better utilizing all cores. Similar to federated scheduling, Light DAGs are converted to sequential tasks and they are also assigned to common cores, once again, to prevent having dedicated cores with low utilization.

The sketch of the designed mechanism to realize our idea is as follows. With the aim of trying to use fewer cores, we generate a primary schedule for each DAG. Then, we do some sort of schedule compaction on each primary schedule. After compaction, we put the cores with utilization less than a specified threshold (u), e.g. 0.6, aside and dedicate the other cores to the DAG. At last, for each task of the cores with utilization less than u, we determine a release time and a deadline to enable scheduling them with the other sequential tasks on common cores. These tasks are scheduled using one of the known schedulers suitable for independent (constrained) sequential real-time tasks. Algorithm 1, shows the method at a high level of abstraction and more details of some of its steps is described in the next algorithms.

Algorithm 1: Semi-Federated Scheduling of Multiple DAGs

1. m: The number of cores in the system
2. u: Core utilization threshold for dedication
3. **IF** (Sum of DAG utilizations) > m **THEN**
4. ***RETURN FAILURE***
5. **END-IF**
6. $Rem \leftarrow m$
7. **FOR EACH** DAG as DAG_i
8. **IF** Primary scheduling for DAG_i is successful **THEN**
9. $Sch \leftarrow$ Primary schedule of DAG_i
10. **ELSE**
11. ***RETURN FAILURE***
12. $Sch \leftarrow$ Compacted Sch
13. $h_i \leftarrow 0$
14. **IF** (Required number of cores for Sch) > 1 **THEN**
15. $h_i \leftarrow$ (The number of cores in Sch with utilization >= u)
16. $TS \leftarrow TS \cup$ (Detached tasks of cores in Sch with utilization < u)
17. **ELSE**
18. $TS \leftarrow TS \cup$ (Convert DAG_i to a single periodic task with offset 0 and deadline = DAG_i period)
19. **END-IF**
20. **IF** $h_i > Rem$ **THEN**
21. ***RETURN FAILURE***
22. **END-IF**
23. Dedicate h_i cores to DAG_i
24. $Rem \leftarrow Rem - h_i$
25. **END-FOR**
26. **IF** (Scheduling TS on at most Rem cores is successful) **THEN**
27. ***RETURN SUCCESS***
28. **END-IF**
29. ***RETURN FAILURE***

The compaction algorithm we use for step 9 of Algorithm 1 has some special features that are very helpful to the success of the semi-federated scheduling method. It will be discussed later in Algorithm 3. The value of the threshold u in steps 12 and 13 determines the minimum required utilization for the cores to be dedicated. In step 13, all tasks of the cores with utilizations less than u in the compacted schedule will be detached from the DAG to be scheduled later on the common cores. The details of the detachment algorithm used in the proposed method will be described in Algorithm 6.

In step 23, if the goal is balancing core loads and ensuring shorter runtime, we can use all the remaining cores (*Rem* cores) as common cores. Alternatively, we can target the goal of reducing the number of used common cores by running the algorithm in a loop, starting from the ceiling of the sum of task utilizations as the minimum number of required cores and going upward in steps of adding one core at a time until *TS* is safely scheduled. Of course, if the number of cores reaches *Rem* and for none of the values there exist a safe schedule the scheduling is unsuccessful. In the second option, the runtime of used algorithm becomes more important. In experiments, we used the second option to better show the performance of the proposed method in non-extreme system utilizations. The details of the algorithm used for scheduling the *TS* are presented in the experiments section.

Algorithm 2, sketches the details of realization of step 8 of Algorithm 1. Its goal is to achieve a safe schedule with as few cores as it can. Within its body, a classic heuristic scheduling algorithm called Graham's List Scheduling approach for multiprocessors is used [3]. This latter algorithm takes a

priority list of all the tasks to be scheduled, as its input. Here, we use the upward-rank topological sort for prioritizing the tasks. Reference [11] also uses an algorithm like Algorithm 2 for scheduling each DAG on its dedicated processors.

Algorithm 2: Primary Scheduling of a DAG

```
1. DAG: The DAG to be scheduled
2. m: The number of cores in the system
3. P ← List of tasks of DAG sorted by upward-rank
4. FOR i ← ⌈Utilization of DAG⌉ TO m
5.     IF Graham's List Scheduling of DAG with priorities P
         on i cores is successful THEN
6.         RETURN SUCCESS
7.     END-IF
8. END-FOR
9. RETURN FAILURE
```

The result of a successful run of Algorithm 2, is a preliminary schedule, and compaction and detachment operations will follow before the actual schedule is finalized.

The outline of the approach for compaction of the primary schedule of a DAG is to choose one of the used cores in the schedule and to insert its tasks into other cores that are used by the schedule for this DAG. In doing so, the number of tasks in the selected core are gradually decreased to the point where no more tasks can be removed by the algorithm or there are no more task left in that core. In the latter, the core is freed. In any case, the process continues by selecting another core until all cores are examined. Previously examined cores will not be the target of task insertions. This is a general idea that can be applied on the results of different types of schedulers and is not tailored for Algorithm 2.

Any effort to reduce the number of used cores – if implemented effectively and efficiently – can help improving the success ratio of a federated scheduling method. But, the merit in the compaction idea of this paper is that even in case the compactions do not reduce the number of used cores of individual DAG schedules, in our semi-federated method, compactions can result in a notable reduction in the total number of used cores of all DAGs. The reason is that the compaction idea lowers the total utilization of those cores in DAG schedules that their tasks should be detached, lowering the number of the required common cores to schedule detached tasks. The common cores have to generally satisfy a low safe utilization bound as they host sequential constrained tasks with different periods from different DAGs.

Algorithm 3, is a method that realizes the proposed compaction idea in which very simple heuristic decisions are used. For example, each time, (1) the core with lowest utilization is taken as source core, (2) the task with longest execution time is selected from source core, and (3) a core with the lowest utilization (other than source core itself) is chosen to be the candidate target core for the task. A side-effect of running Algorithm 3 on the primary schedule is the extension of the schedule toward the deadline as a result of task insertions. Therefore, tasks generated in step 13 of Algorithm 1 will have longer release time to deadline ranges.

Algorithm 3: DAG Schedule Compaction

```
1.  S ← The input schedule
2.  FOR p ← (Number of cores used in S) – 1 TO 1
3.      Sort cores in S by utilizations descending
4.      l ← Sorted list of core p tasks by workloads descending
5.      FOR i ← 0 TO (Number of tasks in l) - 1
6.          t_i ← Task with index i in l
7.          FOR k ← p – 1 TO 0
8.              IF Utilization_{P_k} + Utilization_{t_i} > 1 THEN
9.                  BREAK
10.             END-IF
11.             T ← S
12.             IF Insertion of t_i into core k is successful THEN
13.                 Sort cores in S by utilizations descending
14.                 BREAK
15.             END-IF
16.             S ← T
17.         END-FOR
18.     END-FOR
19. END-FOR
20. RETURN S
```

Step 12 in Algorithm 3 tries to remove the longest not yet tried task from the core with the lowest utilization and insert it in the schedule of another core. Algorithm 4 describes the details of this operation.

Algorithm 4: Insertion of task t into core k

```
1.  Remove t from its previous core
2.  i ← Latest finish time among parents of t in the schedule
3.  IF (Another task t' is scheduled to be running on core k at time i) THEN
4.      i ← Finish time of t'
5.  END-IF
6.  IF Pushing t into core k at time i is not successful THEN
7.      RETURN FAILURE
8.  END-IF
9.  FOR EACH task t_j that starts at time i or later on core k
10.     IF Schedule Correction for t_j is not successful THEN
11.         RETURN FAILURE
12.     END-IF
13. END-FOR
14. RETURN SUCCESS
```

In the condition expression in step 6 of Algorithm 4, the tasks scheduled at time i or later on the core k will be pushed forward, if needed, to make the required room for t to be scheduled at time i on the core. The operation is unsuccessful when pushing the tasks forward causes violation of the DAG deadline in this core. In step 10, the *Schedule Correction* algorithm is applied on all tasks whose start times are pushed forward by inserting t into the core k. This is done to guarantee that the insertion has not caused violation of the precedence relations of the tasks with the tasks on other cores. Algorithm 5 shows the details of this corrective operation. In step 4 of Algorithm 5, Algorithm 4 is called to, if possible, schedule a child task on its own core sometimes later, so that the violated parent-child precedence constraint is corrected.

Algorithm 5: Schedule Correction on a task t

1. $f \leftarrow$ finish time of t in the schedule
2. **FOR EACH** child of t as c
3. **IF** start time of c is less than f **THEN**
4. **IF** Inserting c into its own core is not successful **THEN**
5. *RETURN FAILURE*
6. **END-IF**
7. **END-IF**
8. **END-FOR**
9. *RETURN SUCCESS*

Algorithm 3, i.e. the schedule compaction algorithm applied on the primary schedule, is one of our important novelties in this paper and effectiveness of its operation plays a key role to the success of the semi-federated method. Here, to exemplify the effectiveness of the algorithm, an example on a DAG is presented. The DAG is shown in Figure 1. In this figure, the name of each task is shown in its left side and its execution time is shown inside the vertex. Complementary information about the DAG is given in Table 1.

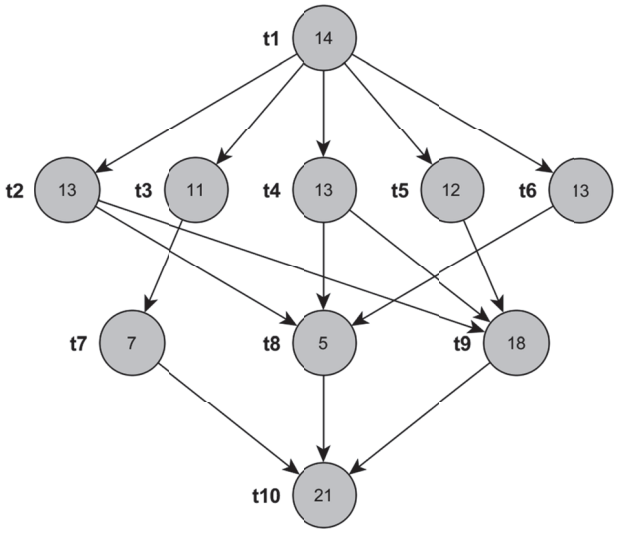


Figure 1: An example DAG to show the operation of the Schedule Compaction algorithm

Table 1: Complementary information about the DAG of Figure 1

Critical-path Length	Utilization	Period	Exec. Times Sum
66	1.6	80	127
Priority of tasks based on Upward-Rank, from left to right			
t1, t2, t4, t5, t3, t6, t9, t7, t8, t10			

In Figure 2, the result of running the primary scheduler on the DAG of Figure 1 is presented. The utilization of each core is shown in front of its schedule. The federated scheduling will dedicate 3 cores, with the average utilization of 53%, to run the DAG.

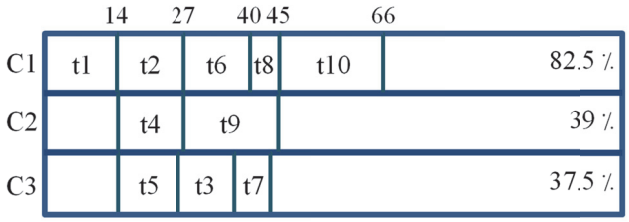


Figure 2: The result of running the primary scheduler on the DAG of Figure 1

In Figure 3, the result of running the schedule compactor on the schedule of Figure 2 is presented. In the semi-federated scheduling method presented in Algorithm 1, supposing a u less than 0.54, two cores with average utilization of 75% will be dedicated for the DAG. The task $t7$ with period 80, release time 38, and deadline 57 will be scheduled on common cores. It can be assumed that along with $t7$, a number of other detached tasks from one or more other DAGs will be re-scheduled on the core C3. This means that the utilization of the cores engaged in the scheduling is grown higher. So, because the number of used cores lowers through the scheduling process and more free cores remain available to the scheduler, the success ratio is deemed to increase.

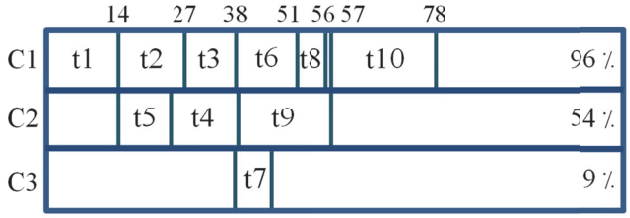


Figure 3: The result of running the schedule compactor on the schedule of Figure 2

Algorithm 6 presents more details about step 13 of Algorithm 1, i.e. the operation of detaching the tasks of a core. Having the schedule of a DAG, it is possible to consider any task apart from the DAG schedule and re-schedule it independent of the DAG, by determining a period, a release time, and a deadline for the task. For such a task, the period is the DAG's period, the release time is the finish time of its latest finish time parent, and the deadline is the start time of its earliest start time child. To detach more than one task, it is required to consider, beside the start times and finish times in the schedule, the release time and deadline of already detached tasks.

Algorithm 6: Detachment of tasks from *DAG*

/* Detachment of all the tasks scheduled on core c having the schedule *Sch* and the set of already Detached tasks *ADet* */

1. FOR EACH task t scheduled on core c
2. *O1, O2* $\leftarrow 0$
3. *D1, D2, Period(t)* $\leftarrow$ *Period(DAG)*
4. *O1* $\leftarrow$ *Latest finish time among parents of t in Sch (if any)*
5. *O2* $\leftarrow$ *Latest deadline among parents of t in ADet (if any)*
6. *D1* $\leftarrow$ *Earliest start time among children of t in Sch (if any)*
7. *D2* $\leftarrow$ *Earliest release time among children of t in ADet (if any)*
8. *Offset(t)* $\leftarrow$ *Maximum of O1 and O2*
9. *Deadline(t)* $\leftarrow$ *Minimum of D1 and D2*
10. *ADet* $\leftarrow$ *ADet* $\cup$ t
11. **END-FOR**

Algorithm 6 is somehow a greedy one. Meaning that, facing any task, it assumes the longest possible range of release time to deadline for the task. This way, the flexibility of the next tasks to be detached, from current core or next cores, gradually decreases.

One thing we have not discussed about the algorithms is their time complexity analysis. However, in the experiments section, we have shown the runtime of the presented semi-federated method in scheduling DAGs that each have at most 50 tasks on a system with 64 cores. To deal with bigger input sizes or increase the scheduling speed, one approach would be to put a limit on the depth of the recurrent calls of Algorithm 4 and when the limit is reached return failure. It would be the failure of step 12 in Algorithm 3, which is a good tradeoff of the success ratio and used processors count for runtime of the scheduling.

IV. Experiments

In the experiments, the number of cores in the system was selected to be 2, 4, ..., and 64. The maximum utilization of the system was chosen from 0.2, 0.3, …, and 0.9. The focus was mostly on success ratio, i.e. the ratio of the number of safely schedulable sets to the number of all selected sets for the experiment. The number of used cores and the time used to run the scheduler are also measured.

Primarily, 10000 DAGs were randomly generated using the DAGGen [19] generator with the number of vertices for each DAG chosen conforming uniform distribution from 3 to 50. In each DAG, the maximum number of level jumps for each edge was set to three. Other parameters were set to be the same as defaults. The whole set of 10000 generated DAGs are named the DAGs' Dataset, for consequent references.

In each experiment, 100 sets of DAGs are generated using the DAGs' dataset in such a way that satisfies the constraints on the specific experiment. The average of the results of all 100 sets of DAGs is the measure which is used in figures. To form a set of DAGs, DAGs are selected randomly one at a time from DAGs' dataset. For each selected DAG a repetition period is calculated with respect to its critical-path length, L. Reference [20] was the guideline of generating periods such that the hyper-period, i.e. Least Common Multiple (LCM) of all periods, of all selected DAGs is not very big. A Pareto distribution with α=1 and β=1 was utilized to generate the period of each DAG in such a way that it lies within $1L$ to $8L$. Then, the utilization of the selected DAG is calculated and, if adding it does not violate the maximum system utilization constraint for this experiment, it is added to the set. The process continues by selecting new DAGs until the maximum allowable system utilization for this experiment is reached.

Recall that after the static scheduling of DAGs, the whole system is composed of two types of tasks. Type 1 tasks are assigned to dedicated cores and their execution start time and completion time are calculated and the same timing is repeated for each period of the corresponding DAG. Type 2 tasks are not forced to be executed by a specific core and each of their instances could be executed by so called the most appropriate core. Although the whole scheduling method presented here is static, but Type 2 tasks are scheduled in a later phase of scheduling. Each Type 2 task has a release time and a deadline, and for their scheduling, tasks are selected based on Earliest Deadline First (EDF) and each picked task is assigned to a core based on the Next-Fit strategy.

On the other hand, cores are also in two types. A Type 1 core is dedicated to run all or some tasks of one DAG and Type 2 cores are those which only run Type 2 tasks. A Type 1 core may also be used for running Type 2 tasks when there is no conflict with its Type 1 tasks.

In the proposed method, cores with utilization equal or greater than u are considered dedicated. In our experiments, the best empirical value for u was around 0.5. Hence all evaluations are done for u = 0.5. Figure 4 shows the impact of increasing the threshold u on the success ratio of the method with respect to some different system utilizations. By increasing u from 0.1 to 0.5, the success ratio increases. For $u \geq 0.6$ the success ratio falls again, such cases are not shown in the figure.

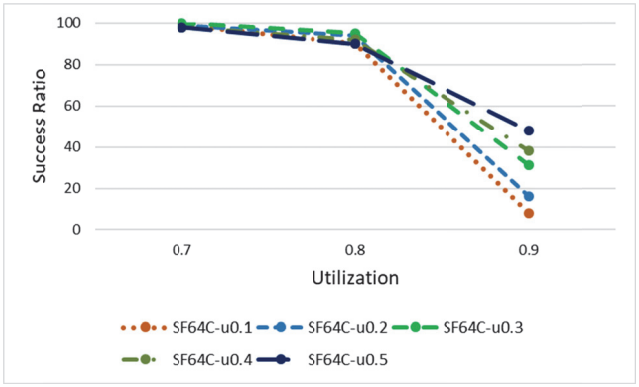
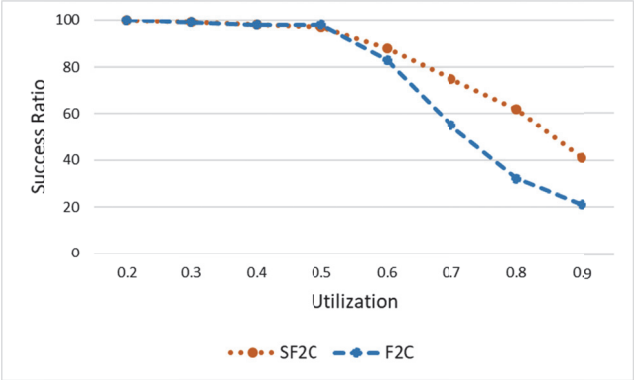
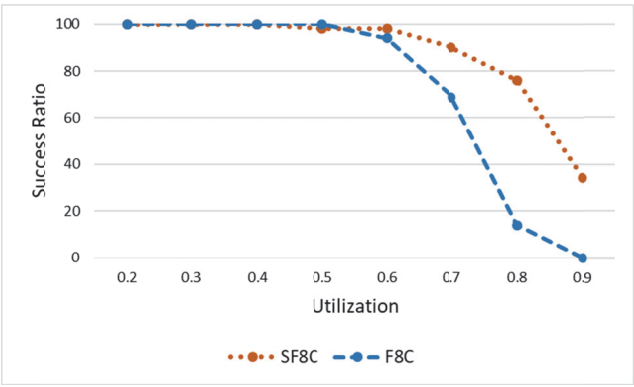


Figure 4. The effect of u on the success ratio measure

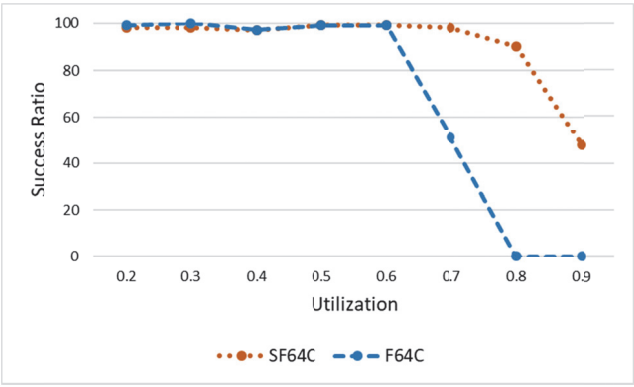
Similar to comparable researches, the most important performance measure for us is the success ratio of the proposed method [4], [14]. It shows the strength of the method with respect to safely running more DAGs with high total utilization. Figure 5 is the comparison of our method with its competing method, i.e. federated scheduling [10] on this measure for systems with 2, 8, and 64 cores.



(a) Two cores



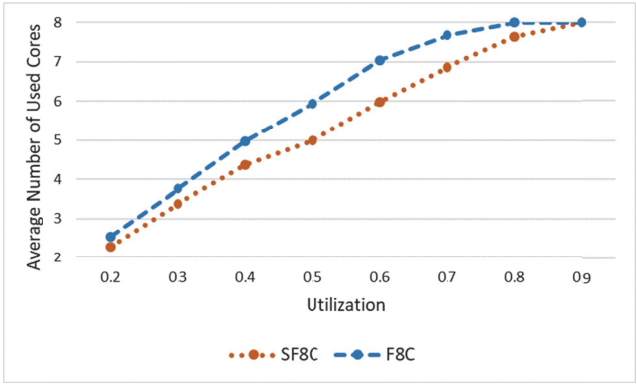
(b) Eight cores



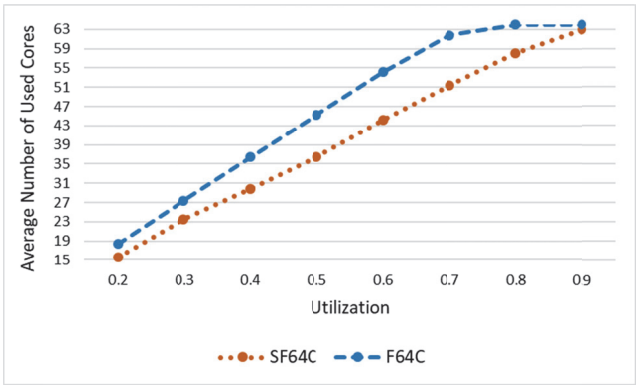
(c) Sixty four cores

Figure 5. Comparison of the Semi-Federated algorithm (SF) vs. Federated algorithm (F) with respect to success ratio for systems with 2, 8, 64 cores.

The number of used cores is another performance measure for comparing the federated and semi-federated algorithms. The results are shown in Figure 6. Two cases are considered, the first case considers the hardware system is composed of 8 cores and the second one considers 64 cores. In both cases, the maximum system utilization ranges from 0.2 to 0.9. The proposed algorithm's performance is superior in all cases.



(a) An eight-core system



(b) A sixty four-core system

Figure 6. The actual cores used by federated and semi-federated algorithms with different maximum system utilizations.

It is obvious that the run time of the proposed algorithm is higher than that of federated algorithm. To show how much more execution time is used by the semi-federated method the experiment is performed, once again, for 8 and 64 core systems. The results are shown in Figure 7 for different utilizations. The difference is noticeable but, we have to mention that (1) the scheduling is done off-line before the actual system deployment, (2) the time needed for the scheduling is in the scale of milliseconds, and (3) the scheduling is done for the whole hyper-period of DAGs.

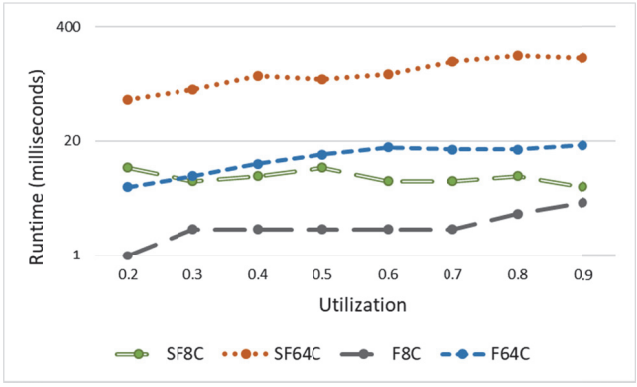


Figure 7. Runtime of federated and semi-federated algorithms for 8 and 64 core systems as the utilization increases.

V. Summary and Future Work

In this study, scheduling multiple periodic real-time DAGs for their safe execution in multicore systems is investigated. The approach is based on the concept of *federated scheduling* in which, for each DAG a separate set of cores is assigned. With this approach, practical cases reveal that a noticeable portion of each core's capacity is lost. We proposed a semi-federated scheduling in which the restriction of dedicating separate cores to each DAG is relaxed somehow, on one hand. On the other hand, a compaction procedure is applied to the results of each DAG's schedule, and then, cores with utilization less than a certain value are eliminated and their tasks are distributed to other cores in common between multiple DAGs.

The proposed method is implemented and compared with federated scheduling using 10000 DAGs which were generated using a previously developed method by other researchers. The number of cores of the hardware systems varied from 2 to 64 in the experiments. Success ratio of the proposed method was most of the time better than and very seldom the same as federated method. An extreme case was sets of DAGs to be scheduled on a 64-core processor with an imposed workload of 0.8, where the success ratio of the federated method was zero while that of semi-federated method was 90 percent.

Proposing a semi-federated method for DAGs with other types of recurrence such as sporadic DAGs, and also DAGs with other types of deadlines such as arbitrary deadlines seems very challenging. As processors with heterogeneous cores become common, federated and semi-federated methods can be extended to these kinds of processors.

References

[1] K. Yang, G. A. Elliott, and J. H. Anderson, "Analysis for Supporting Real-Time Computer Vision Workloads using OpenVX on Multicore plus GPU Platforms," Proc. 23Rd Int. Conf. Real-Time Networks Syst. 2015, pp. 77–86, 2015.

[2] K. Yang, M. Yang, and J. H. Anderson, "Reducing Response-Time Bounds for DAG-Based Task Systems on Heterogeneous Multicore Platforms," in Proceedings of the 24th International Conference on Real-Time Networks and Systems - RTNS '16, 2016, pp. 349–358.

[3] R. L. Graham, "Bounds on Multiprocessing Timing Anomalies," SIAM J. Appl. Math., vol. 17, no. 2, p. 14, 1969.

[4] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility Analysis in the Sporadic DAG Task Model," in 2013 25th Euromicro Conference on Real-Time Systems, 2013, pp. 225–233.

[5] E. Quinones, "Response-Time Analysis of DAG Tasks under Fixed Priority Scheduling with Limited Preemptions," in Design, Automation & Test in Europe Conference & Exhibition (2016), 2016, pp. 1066–1071.

[6] M. Naghibzadeh, P. Neamatollahi, R. Ramezani, A. Rezaeian, and T. Dehghani, "Efficient semi-partitioning and rate-monotonic scheduling hard real-time tasks on multi-core systems," in 2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES), 2013, no. Sies, pp. 85–88.

[7] R. Ramezani, Y. Sedaghat, M. Naghibzadeh, and J. A. Clemente, "Reliability and Makespan Optimization of Hardware Task Graphs in Partially Reconfigurable Platforms," IEEE Trans. Aerosp. Electron. Syst., vol. 53, no. 2, pp. 983–994, Apr. 2017.

[8] J. M. Calandrino and J. H. Anderson, "On the design and implementation of a cache-aware multicore real-time scheduler," in Proceedings - Euromicro Conference on Real-Time Systems, 2009, pp. 194–204.

[9] H. A. Hassan, S. A. Salem, A. M. Mostafa, and E. M. Saad, "Harmonic Segment-Based Semi-Partitioning Scheduling on Multi-Core Real-Time Systems," ACM Trans. Embed. Comput. Syst., vol. 15, no. 4, pp. 1–29, Aug. 2016.

[10] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, "Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks," in 2014 26th Euromicro Conference on Real-Time Systems, 2014, pp. 85–96.

[11] S. Baruah, "Federated Scheduling of Sporadic DAG Task Systems," in 2015 IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 179–186.

[12] X. Jiang, N. Guan, X. Long, and W. Yi, "Semi-Federated Scheduling of Parallel Real-Time Tasks on Multiprocessors," in 2017 IEEE Real-Time Systems Symposium (RTSS), 2017, pp. 80–91.

[13] S. Abrishami and M. Naghibzadeh, "Deadline-constrained workflow scheduling in software as a service Cloud," Sci. Iran., vol. 19, no. 3, pp. 680–689, Jun. 2012.

[14] X. Jiang, X. Long, N. Guan, and H. Wan, "On the Decomposition-Based Global EDF Scheduling of Parallel Real-Time Tasks," in 2016 IEEE Real-Time Systems Symposium (RTSS), 2016, pp. 237–246.

[15] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill, "Parallel real-time scheduling of DAGs," IEEE Trans. Parallel Distrib. Syst., vol. 25, no. 12, pp. 3242–3252, 2014.

[16] M. Qamhieh, L. George, and S. Midonnet, "A Stretching Algorithm for Parallel Real-time DAG Tasks on Multiprocessor Systems," in Proceedings of the 22nd International Conference on Real-Time Networks and Systems - RTNS '14, 2014, pp. 13–22.

[17] M. Qamhieh and S. Midonnet, "An experimental analysis of DAG scheduling methods in hard real-time multiprocessor systems," in Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems - RACS '14, 2014, vol. 14, no. 4, pp. 284–290.

[18] M. Qamhieh, L. George, and S. Midonnet, "Stretching algorithm for global scheduling of real-time DAG tasks," Real-Time Syst., pp. 1–31, Jun. 2018.

[19] F. Suter, "DAGGen." GitHub repository, https://github.com/frs69wq/daggen, 2013.

[20] J. Goossens and C. Macq, "Limitation of the Hyper-Period in Real-Time Periodic Task Set Generation," Proc. 9th Int. Conf. Real-Time Syst., pp. 133–148, 2001.