Scheduling of Real-Time Tasks with Multiple Critical Sections in Multiprocessor Systems

January 8, 2020

1 Implementation and Overheads

In this section, we present details on how we implemented the dependency graph approach in LITMUS^{RT} to support multiple critical sections per task. Afterwards, the implementation overheads are compared with the Flexible Multiprocessor Locking Protocol (FMLP) [1] provided by LITMUS^{RT} for both partitioned and global scheduling.

1.1 Implementation Details

When implementing our approach in LITMUS^{RT}, we can either apply the table-driven scheduling that LITMUS^{RT} provides, or implement a new binary semaphore which enforces the execution order of critical sections that access the same resource, since this order is defined in advance by the dependency graph. A static scheduling table can be generated over one hyper-period and be repeated periodically in a table-driven schedule. This table determines which sub-job is executed on which processor for each time point in the hyper-period. However, due to the large number of sub-jobs in one hyperperiod and possible migrations among processors, the resulting table can be very large. To avoid this problem, we decided to implement a new binary semaphore that supports all the properties of our new approach instead.

Since our approach is an extension of the DGA by Chen et al. [2], and Shi et al. [4], our implementation is based on the source code the authors provided online [3], i.e., it is implemented under the plug-in Partitioned EDF with synchronization support (PSN-EDF), called P-DGA-JS, and the plug-in Global EDF with synchronization support (GSN-EDF), denoted G-DGA-JS.

The EDF feature is guaranteed by the original design of these two plug-ins. Therefore, we only need to provide the relative deadlines for all the sub-jobs of each task, and LITMUS^{RT} will automatically update the absolute deadlines accordingly during runtime.

In order to enforce the sub-jobs to follow the execution order determined by the dependency graph, our implementation has to: 1) let the all the sub-jobs inside one job follow the predefined order; 2) force all the sub-jobs that access the same resource to follow the order determined by the graph.

The first order is ensured in LITMUS^{RT} by default. The task deploy tool `rtspin` provided by the user-space library *liblitmus* defines the task structure, e.g., the execution order of non-critical sections and critical sections within one task, the related execution times, and the resource ID that each critical section accesses. Moreover, the resource ID for each critical section is parsed by `rtspin`, so the critical section can find the correct semaphore to lock, and in our implementation we do not have to further consider addressing the corresponding resources. Afterwards, `rtspin` emulates the work load in a CPU according to the taskset. A sub-job can be released only when its predecessor (if any) has finished its execution. Please note that for sub-jobs related to critical sections the release time is not only defined by its predecessor's finish time inside the same job, but also related to another predecessor that accesses the same resource (if one exists).

A ticket system with a similar general concept to [3] is applied to enforce the execution order. However, due to different task structure which allows to support multiple critical sections, compared to [3], additional parameters had to be introduced and the structure of existed parameters had to be revised. To be precise, we extended LITMUS^{RT} data structure `rt_params` that describes tasks, e.g., priority, period, and execution time, by adding:

- `total_jobs`: an integer which defines the number of jobs of the related task in one hyper-period.
- `total_cs`: an integer that defines the number of critical sections in this task.

	total_jobs	total_cs	job_order	current_cs
τ_1	2	2	[1,3,6,8,9,9]	1
τ_2	2	2	[0,2,5,7,9,9]	1
τ_3	2	2	[1,3,6,8,9,9]	0
τ_4	2	2	[0,2,5,7,9,9]	0
τ_5	1	1	[4,4,9,9]	0

Table 1: An example of the data structure for tasks.

- `job_order`: an array which defines the total order of the sub-jobs related to critical sections that access the same resource over one hyper-period. In addition, the last Z elements record the total number of critical sections of the taskset for each shared resource. Thus, the length of the array is the number of critical sections in one hyper-period plus the number of total shared resources, i.e., `len(job_order)` = `total_jobs` $\times$ `total_cs` + Z.
- `current_cs`: an integer that defines the index of the current critical section of the task that is being executed.
- `relative_ddls`: an array which records the relative deadlines for all sub-jobs of one task.

Furthermore, we implemented a new binary semaphore, named as `mdga_semaphore`, to make sure the execution order of all the sub-jobs that access the same resource follows the order specified by the dependency graph.

A semaphore has the following common components:

- `litmus_lock` protects the semaphore structure,
- `semaphore_owner` defines the current holder of the semaphore, and
- `wait_queue` stores all jobs waiting for this semaphore.

A new parameter named `serving_ticket` is added to control the non-work conserving access pattern of the critical sections, i.e., a job can only lock the semaphore and start its critical section if it holds the ticket equals to the corresponding `serving_ticket`.

The pseudo code in Algo. 1 shows three main functions in our implementation: The function `get_cs_order` returns the position of the sub-job in the execution order for all the sub-jobs that access the same shared resource during the run-time. In LITMUS$^{\mathrm{RT}}$, `job_no` counts the number of jobs that one task releases. In order to find out the exact position of this job in one hyper-period, we apply a modulo operation on `job_no` and `total_jobs`. Since a job has multiple critical section and the `current_cs` represents the position of the critical section in a job, the index is calculated by counting the number of previous jobs' critical sections and the `current_cs` in this job. After that, the value of `cs_order` is searched from `job_order` based on the obtained index.

We provide an example with 5 tasks which share two resources. The four tasks τ_1, τ_2, τ_3, and τ_4 are identical to Fig. **??** and task τ_5 has a period $T_5 = 50$ and the same pattern as τ_4, i.e., it requests resource 2 in its second segment and request resource 1 in its forth segment. Hence, the hyper-period for this taskset is 50, τ_1, τ_2, τ_3, and τ_4 release two jobs in one hyper-period, and τ_5 releases one job in one hyper-period. The related data structure is shown in Table 1. Task τ_1 has the `job_order` = [1, 3, 6, 8, 9, 9]. The first two elements, i.e., [1, 3], represents that the two critical sections of J_1^1 have the execution order 1 and 3 accordingly, the following two elements, i.e., [6, 8] denotes the execution order for J_1^2's two critical sections in one hyper-period, and the last two elements, i.e., [9, 9] shows the number of jobs that request the related resources. For both resource 1 and resource 2, there are nine jobs which request the resource in one hyper-period. Assume that the `job_no` for τ_1 is 13. Line 1 in Algo. 1 returns the `current_jobno` which represents the corresponding relative position in one hyper-period, i.e., the 13^{th} job of τ_1 is the second job of τ_1 in the current hyper-period. Then line 2 finds the index of corresponding critical section, i.e., the second critical section of the second job of τ_1 has the index 3. In the end, the corresponding execution order can be found from `job_order` according to line 3 in Algo. 1. Therefore, the 13^{th} job of task τ_1 now has the execution order 8 to grant access to the corresponding resource.

Algorithm 1 DGA with multi-critical sections implementation

Input: New coming task τ_i\{job_no, total_jobs, total_cs, current_cs, relative_ddls\}, and Requested semaphore s_z\{semaphore_owner, serving_ticket, wait_queue\};

Function get_cs_order():
1: current_jobno $\leftarrow$ τ_i.job_no mod τ_i.total_jobs;
2: index $\leftarrow$ current_jobno $\times$ τ_i.total_cs + current_cs;
3: cs_order $\leftarrow$ τ_i.job_order[index];

Function mdga_lock():
4: **if** s_z.semaphore_owner is NULL and s_z.serving_ticket equals to τ_i.cs_order **then**
5: s_z.semaphore_owner $\leftarrow$ τ_i;
6: Update the deadline for τ_i;
7: τ_i starts the execution of its critical section;
8: **else**
9: Add τ_i to s_z.wait_queue;
10: **end if**

Function mdga_unlock():
11: τ_i releases the semaphore lock;
12: Update the deadline for τ_i;
13: τ_i.current_cs++;
14: **if** τ_i.current_cs = total_cs **then**
15: Set τ_i.current_cs $\leftarrow$ 0;
16: **end if**
17: s_z.serving_ticket++;
18: **if** s_z.serving_ticket = num_cs **then**
19: Set s_z.serving_ticket $\leftarrow$ 0;
20: **end if**
21: Next task τ_{next} $\leftarrow$ the head of the wait_queue (if exists);
22: **if** serving_ticket equals to τ_{next}.cs_order **then**
23: s_z.semaphore_owner $\leftarrow$ τ_{next};
24: τ_{next} starts the execution of its critical section;
25: **else**
26: s_z.semaphore_owner $\leftarrow$ NULL;
27: Add τ_{next} to s_z.wait_queue;
28: **end if**

The function `mdga_lock` is called in order to lock the semaphore and get access to the corresponding resource. After getting the correct position in the execution order in one hyper-period by applying function `get_cs_order()`, the semaphore's ownership will be checked. If the semaphore is occupied by another job at that moment, the new arriving job will be added to the `wait_queue` directly; otherwise, the semaphore's `current_serving_ticket` and the job's `cs_order` are compared. If they are equal, the semaphore's owner will be set to that job, and the job will start its critical section; otherwise, the job will be added to the `wait_queue` as well. In our setting the `wait_queue` is sorted by the jobs' `cs_order`, i.e., the job with the smallest `cs_order` is the head of the waiting queue. Hence, only the head of the `wait_queue` has to be checked when the current semaphore owner finishes its execution, rather than checking the whole unsorted `wait_queue`.

The function `mdga_unlock` is called once a job has finished its critical section and tries to unlock the semaphore. The task's `current_cs` is added by one to point to the next possible critical section in this job. If `current_cs` reaches to the `total_cs`, which means all the critical sections in this job have finished their execution, then the `current_cs` will be reset to zero. Next, the semaphore's `serving_ticket` is increased by 1, i.e., it is ready to be obtained by the successor in the dependency graph. If `serving_ticket` reaches the total number of critical sections related to this resource in one hyper-period, i.e., `num_cs`, the dependency graph is traversed completely, i.e., all sub-jobs that access the related resource finished their executions of the critical sections in the current hyper-period, the parameter `serving_ticket` is reset to 0 to start the next iteration. Please note, the `num_cs` can be found in the last Z elements of `job_order` according to the related resource id. After that, the first job (if any) in the `wait_queue`, named as τ_{next} is checked. If τ_{next} has the `cs_order` which equals to the semaphore's `serving_ticket`, the the semaphore's owner is set as τ_{next}, and τ_{next} can

Max. (Avg.) in μs	CXS	RELEASE	SCHED	SCHED2	SEND-RESCHED
P-FMLP	29.51 (0.98)	17.68 (0.96)	31.85 (1.31)	28.77 (0.18)	66.33 (2.86)
P-DGA-JS	30.65 (1.25)	18.63 (1.02)	31.09 (1.64)	29.43 (0.19)	59.09 (21.06)
G-FMLP	30.51 (1.05)	48.53 (3.75)	45.99 (1.51)	29.62 (0.16)	72.26 (2.50)
G-DGA-JS	26.87 (0.94)	30.01 (2.19)	30.25 (1.02)	19.26 (0.14)	72.53 (21.50)
P-LIST-EDF	18.76 (0.90)	18.98 (1.06)	48.50 (1.33)	29.25 (0.16)	38.3 (1.61)
G-LIST-EDF	30.87 (1.79)	61.63 (12.06)	59.05 (4.46)	27.17 (0.25)	72.09 (20.77)

Table 2: Overheads of protocols in LITMUS$^{\mathrm{RT}}$.

start the execution of its critical section. Otherwise, the semaphore owner is set as NULL, and the task τ_{next} is put back to the corresponding `wait_queue`.

Additionally, each sub-job has its own modified deadline accordingly, which means each job can have different deadlines when it is executing different segments. Therefore, we have to take care of the deadline update during the implementation. When we deploy a task using `rtspin` to the system, we deliver the relative deadline of its first sub-task as the relative deadline of the whole task. Since no two continuous non-critical sections are allowed in the task model, once a sub-job finishes its execution, either `mdga_lock` or `mdga_unlock` is called. If `mdga_lock` is called, the new critical section's deadline is updated by searching the `relative_deadline`; if `mdga_lock` is called, only the finished critical section can update related job's deadline for its successor (if any), since τ_{next}'s deadline has been updated when it tries to lock the semaphore already.

The implementations for the global and partitioned plug-ins are similar. However, due to the frequent preemption and/or interrupts in global scheduling, the preemption has to be disabled during the executions of semaphore related functions in order to protect the functionalities of aforementioned functions.

1.2 Overheads Evaluations

We evaluated the overheads of our implementation in the following platform: a cache-coherent SMP, consisting of two 64-bit Intel Xeon Processor E5-2650Lv4, with 35 MB cache and 64 GB main memory. The FMLP supported in LITMUS$^{\mathrm{RT}}$ was also evaluated for comparisons, including P-FMLP for partitioned scheduling and G-FMLP for global scheduling. These four protocols are evaluated using same task sets where each task has multiple critical sections.

The overheads that we tracked are:

- **CXS**: context-switch overhead.
- **RELEASE**: time spent to enqueue a newly released job into a ready queue.
- **SCHED**: time spent to make a scheduling decision, i.e., find the next job to be executed.
- **SCHED2**: time spent to perform post context switch and management activities.
- **SEND-RESCHED**: inter-processor interrupt latency, including migrations.

The overheads are reported in Table 2, which shows that the overheads of our approach and those of P-FMLP, G-FMLP are comparable. Furthermore, the implementations provided in [4], called P-LIST-EDF and G-LIST-EDF, were evaluated to examine the overhead and reported in Table 2. The direct comparison between P-LIST-EDF and P-DGA-JS (G-LIST-EDF and G-DGA-JS, respectively) is not possible because they are designed for different scenarios, depending on the number of critical sections per task. The reported overheads in Table 2 for our approach are for task sets with multiple critical sections per task, whilst the overheads for P-LIST-EDF and G-LIST-EDF were for task sets with one critical section per task. Regardless, they are in the same order of magnitude.

References

[1] A. Block, H. Leontyev, B.B. Brandenburg, and J.H. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA*, pages 47–56, 2007. http://dx.doi.org/10.1109/RTCSA.2007.8 doi:10.1109/RTCSA.2007.8.

[2] Jian-Jia Chen, Georg von der Bruggen, Junjie Shi, and Niklas Ueter. Dependency graph approach for multiprocessor real-time synchronization. In *IEEE Real-Time Systems Symposium, RTSS*, pages 434–446, 2018. URL: https://doi.org/10.1109/RTSS.2018.00057, http://dx.doi.org/10.1109/RTSS.2018.00057 doi:10.1109/RTSS.2018.00057.

[3] JunJie Shi. HDGA-LITMUS-RT. https://github.com/Strange369/Dependency-Graph-Approach-for-Periodic-Tasks, 2019. URL: https://github.com/Strange369/Dependency-Graph-Approach-for-Periodic-Tasks.

[4] Junjie Shi, Niklas Ueter, Georg von der Brüggen, and Jian-jia Chen. Multiprocessor synchronization of periodic real-time tasks using dependency graphs. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 279–292, 2019.