2018 IEEE 21st International Symposium on Real-Time Distributed Computing

Real-Time ROS Extension on Transparent CPU/GPU Coordination Mechanism

Yuhei Suzuki*, Takuya Azumi†, Shinpei Kato‡, and Nobuhiko Nishio*

* Graduate School of Information Science and Engineering, Ritsumeikan University
† Graduate School of Science and Engineering, Saitama University
† College of Information Science and Engineering, Ritsumeikan University
‡ Graduate School of Information Science and Technology, The University of Tokyo

***Abstract*—Robot Operating System (ROS) promotes fault isolation, faster development, modularity, and core reusability and is therefore widely studied and used as the de facto standard for autonomous driving systems. Graphics processing units (GPUs) also facilitate high-performance computing and are therefore used for autonomous driving. As the requirements for real-time processing increase, methods for satisfying real-time constraints for ROS and GPUs are being developed. Unfortunately, scheduling algorithms specifying ROS's transportation (publish/subscribe) model, which can have execution order restrictions, are not being investigated, leading to the introduction of waiting time and degrading the responsiveness of the entire system. Furthermore, GPU tasks on ROS are also affected by the ROS transportation model, because central processing unit (CPU) time is occupied when GPU functions are launched.**

This paper proposes a loadable kernel module framework, called real-time ROS extension on transparent CPU/GPU coordination mechanism (ROSCH-G), for scheduling ROS in a heterogeneous environment without modifying the OS kernel and device drivers and then evaluates it experimentally. ROSCH-G provides a scheduling algorithm that considers ROS's execution order restrictions and a CPU/GPU coordination mechanism. Experimental results demonstrate that the proposed algorithm reduces the deadline miss rate and, compared with previous studies, makes effective use of the benefits of parallel processing. In addition, the results for the coordination mechanism demonstrate that ROSCH-G can schedule multiple GPU applications successfully.

***Index Terms*—Robot Operating System (ROS), GPU, Scheduling, Real-Time, DAG, Autonomous driving.**

I. Introduction

In recent years, real-time distributed embedded systems, such as autonomous driving systems [1], [2], have been attracting attention. As the development of autonomous driving systems advances, the systems become increasingly complicated and grow in scale. This trend is the reason why Robot Operating System (ROS), a widely studied framework in the investigation of multi- and many-core embedded systems, is used as the de facto standard for developing autonomous driving systems. In addition, ROS has rich libraries developed specifically for robotics, and it promotes fault isolation, faster development, modularity, and code reusability. At the same time, graphics processing units (GPUs) are used to accelerate computing. Obstacle detection in autonomous driving is an example of capacities that are expected to be accelerated by the parallel processing of GPUs.

As for real-time processing on ROS, autonomous driving systems must complete periodic tasks under various time constraints (hard real-time responses, such as obstacle detection, and soft real-time operations, such as navigation). However, ROS does not satisfy real-time constraints as a result of having a transportation mechanism that is implemented as middleware based on TCPROS and UDPROS, two transport layers for ROS Messages and Services, using TCP and UDP sockets. This critical problem has been gradually perceived and considered by many research communities, including ROS developers, and various solutions aimed at improving communication have been proposed [3], [4]. Satisfying real-time constraints for ROS also calls for a scheduling algorithm, because its publish/subscribe transportation mechanisms have a restriction on execution order to the effect that they can start their own execution when all the preceding tasks have been completed. Recent research [5], [6], [7], [8] on scheduling such a system concentrates primarily online scheduling (e.g., global earliest deadline first (GEDF) and deadline monotonic (DM)). However, these online scheduling methods can incur scheduling overheads. Such overhead can have serious effects on hard real-time systems. Thus, we are interested in offline scheduling, analyzing the execution order restriction. To the best of our knowledge, offline scheduling specifying ROS's transportation system does not yet exist.

Satisfying real-time constraints in autonomous driving also requires a scheduling algorithm that considers GPU tasks. GPU tasks, whose principal purpose is to accelerate specific computing blocks, are often best-effort oriented. Therefore, conventional GPU technologies are specifically designed for individual data-parallel and compute-intensive workloads. However, owing to the emergence of real-time systems using GPUs, having system software to support the real-time management of GPU resources is becoming a more significant requirement. In addition, GPU tasks on ROS are affected by ROS's execution order restriction, thereby central processing unit (CPU) scheduling is also important for GPU applications. Furthermore, CPU time can be occupied by busy spinning when waiting for the GPU. Practically, CUDA programming instructs CUDA to spin or yield its thread, and these instructions are changed based on heuristics by default. Releasing CPU while running GPU enables more efficient scheduling of tasks on the CPU. As a consequence, a

2375-5261/18/$31.00 ©2018 IEEE
DOI 10.1109/ISORC.2018.00035

cooperated CPU/GPU scheduling algorithm or mechanism is required for autonomous driving systems.

Contribution: In this paper, we propose offline CPU/GPU scheduling algorithms and transparent mechanisms for ROS called ROSCH-G[1]. ROSCH-G's implementation is based on RESCH [10], [11] which provides a loadable real-time scheduler suite for Linux, allowing a system to reconfigure scheduling algorithms easily and install their modules for real-time ROS and GPU applications. Offline scheduling is needed to consider multiple deadlines, whereas previous research of offline scheduling [12], [13], [14], [15] can satisfy only a single deadline constraint of the end node. Therefore, one of our contributions is the offline scheduling algorithm for such the restriction of ROS. In addition, the core set of innovations in scheduler design of ROSCH-G is a priority based GPU scheduler with exclusive control and a CPU release mechanism for offline scheduling. This is because that offline scheduling is also required that all tasks execution is predictable, whereas an execution order and an execution time of GPU task is unpredictable. To the best of our knowledge, this is the first study of offline scheduling for CPU/GPU tasks under the restrictions of the publish/subscribe model. From the experimental results, scheduling specified in the publish/scribe model demonstrates a performance improvement for real-time processing.

Organization: The remainder of this paper is organized as follows. Section II discusses problems with the real-time use of ROS and GPU, and Section III introduces related work. With specific emphasis on the constraints for ROS, Section IV presents the design and implementation of ROSCH-G. Sections V and VI present the results of an evaluation of the proposed ROSCH-G. Finally, Section VII concludes the paper and offers suggestions for future work.

II. Problem Definition

In this section, We discuss and clarify the scheduling requirements for a real-time response for ROS and GPUs. Note that the terms "task" and "*node*" are used interchangeably in this paper. A GPU task is defined as a process running on the CPU that launches a GPU kernel to the GPU. The GPU kernel is the process executed on the GPU.

A. Real-Time Requirement for ROS

ROS's communication among *nodes* is based on a publish/subscribe model. The publish/subscribe model can be paraphrased as an event-driven dataflow system in which a *node* is generally launched when the predecessor *nodes* are completed. This means that launching a *node* closely depends on the execution order restriction. However, such the restriction can introduce a waiting term that could jeopardize schedulability. For example, in Fig. 1, *node C* waits for processing *node E*, thereby pushing the processing of all succeeding tasks.

[1]ROSCH [9], our development on Github, stands for real-time extension modules for ROS.

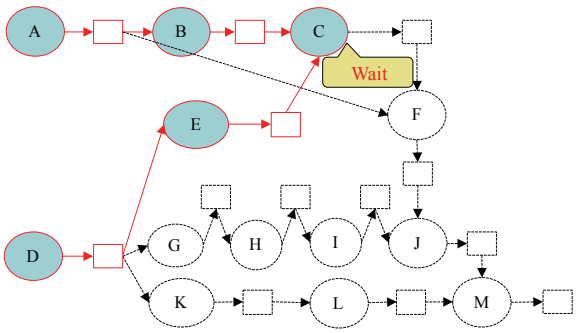


Fig. 1. Example of an execution order restriction

Meanwhile, GEDF and DM are well-known algorithms for priority-driven scheduling for real-time systems. Unfortunately, under such scheduling without considering the execution order restriction, even a delay of only one node affects the entire system. This means that a *node* prioritized based on the scheduling policy can degrade the responsiveness of the entire system in a system in which tasks depend on each other in regard to complexity. Thus, considering dependencies must be required for real-time ROS processing.

Scheduling considering dependencies is generally recognized to be NP-complete [16], [17]. This means that highly efficient on- or offline scheduling is required. In addition, an algorithm without scheduling overhead is also required in hard real-time systems. Online scheduling involves more or less scheduling overhead and affects systems with strict time constraints. Therefore, we are interested in offline scheduling to exclude scheduling overhead.

B. Real-Time Requirement for GPUs

Conventional GPU technologies are specifically designed for individual data-parallel and compute-intensive workloads. A latest NVIDIA's GPU architecture, Pascal, allows compute tasks to be preempted at instruction-level granularity, rather than thread block granularity as in prior Maxwell and Kepler GPU architectures, thereby preventing long-running applications from monopolizing the system (preventing other applications from running). Unfortunately, the preemption mechanism on GPUs does not ensure time constraints because the scheduling of GPU kernels is not based on priority. The preemption policy is that all GPU kernels are equally processed, so that the response time of a GPU kernel tends to be proportional to the number of simultaneously launched GPU kernels. Consequently, such the preemption mechanism can degrade a performance for real-time processing in general-purpose computing on GPUs (GPGPU) applications.

For instance, consider the example of the execution flow of GPU tasks illustrated in Fig. 2. Here three GPU tasks, τ_1, τ_2, *and* τ_3, in the many-core environment have decreasing priorities and share a single GPU device. This figure shows that GPU tasks with high priorities could not be processed because the scheduling of GPU kernels is not based on priority. Thus, scheduling mechanism for GPU kernel to complete a GPU task with high priority in the original processing time

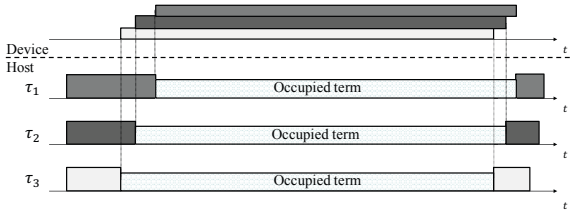


Fig. 2. Execution flow of GPU tasks on a latest NVIDIA's GPU architecture (Pascal).

regardless of the GPU type to satisfy time constraints is required, especially in a system such as ROS in which a delay of one node affects the entire system. Furthermore, a CPU/GPU coordination mechanism is also required, because a GPU task occupies CPUs while running its GPU kernel or waiting for the completion of high priority GPU tasks.

III. Related Work

In this section, we introduce related work in two fields: real-time scheduling for ROS, and GPUs. First, we introduce DAG scheduling algorithms and discuss the problems when assuming to use these algorithms in the ROS. Then, we introduce related work for real-time GPUs in terms of scheduling considering both CPU and GPU in addition to easily install.

A. Real-Time Scheduling for ROS

With the increasing number of available parallel programming models (e.g., Open MP and MapReduce), DAG scheduling considering an execution order restriction has attracted significant attention. In a system exhibiting the execution order restriction, offline DAG scheduling avoids preemption overhead and reduces the number of priority inversions that can be experienced under lazy scheduling, whereas the conventional scheduling method [18] for sequential tasks without considering the execution order may increase overall interference by blocking subsequent tasks. Here, we discuss previous research on on- and offline DAG scheduling algorithms. It should be noted that DAG scheduling is valid for ROS 2.0 [3], [19] which satisfies real-time requirements (e.g., CPU, memory, network bandwidth, threads, and cores) as well as ROS 1.

Recently, online DAG scheduling is also investigated. Research [5], [6], [7] on parallel tasks in general, and on DAG tasks in particular, has concentrated primarily on a DAG level scheduling that all tasks inherit the global parameters of their DAG. However, online DAG level scheduling can cause a deadline miss in the system including tasks with a different period because subtasks are scheduled by the same timing parameters. M. Qamhieh [8] proposed scheduling on a subtask level by GEDF that modifies timing parameters of each task based on their precedence constraints, thereby schedulability was improved rather than DAG level scheduling. However, the scheduling overhead of GEDF can have serious effects on hard real-time systems.

HEFT [12], which shortens the $makespan$[2], is one of the most well-known offline approaches and the basis for many DAG scheduling heuristics [12], [13], [14], [15].

These offline algorithms take into account the execution order restriction. However, the algorithm targets a fork-join structure (e.g., the OpenMP tasking model) and can satisfy only a single deadline constraint of the end $node$, whereas ROS can handle multiple deadlines.

HLBS [20], developed in our previous research, is an offline algorithm that can meet multiple deadline constraints and considers the execution order restriction. This algorithm recursively computes each task's priority by calculating its $laxity$ (or slack). However, the algorithm considers only cases involving the critical instant. This algorithm works similarly to nonpreemptive scheduling. Under nonpreemptive scheduling, it has been proved that analyzing only the critical instant is insufficient owing to the self-pushing phenomenon [21]. Consequently, the algorithm cannot guarantee schedulability when tasks are executed periodically.

B. Real-time for GPUs

The demand for real-time GPU resource management is increasing. Significant challenges for GPUs include system software support for bounded response times and guaranteed throughput. In recent years, GPU technologies have been applied to real-time systems by extending the OS modules to support real-time GPU resource management.

Chimera [22] employs two GPU-specific preemption techniques to achieve low throughput overhead and preemption latency. However, in GPGPU applications, it is necessary to consider the coordinated scheduling of CPU tasks because CPU time can be occupied by busy spinning. Gdev [23] provides an ecosystem of GPU resource management in the OS. It allows the user space as well as the OS itself to use GPUs as first-class computing resources. Gdev further provides a GPU scheduling scheme to virtualize a physical GPU into multiple logical GPUs, enhancing isolation among working sets of multitasking systems. Unfortunately, Gdev and Chimera do not consider CPU scheduling for the systems that tasks on CPU have the execution order. In addition, such the extensions makes it difficult to maintain the system with version updates, because the OS kernel and device drivers must be modified at the source-code level, thereby preventing continuous research and development of GPU technologies for real-time systems.

GPUSync [24] supports fixed-priority and EDF scheduling policies for CPU tasks, and GPU-SPARC [25] employs the SCHED_FIFO scheduling policy. Linux-RTXG [26] also does not require modifying the OS kernel and device drivers owing to the implementation by loadable kernel modules (LKMs). Linux-RTXG makes it possible to schedule the GPU kernel by embedding some functions in the application in addition to a CPU scheduler. However, these work does not properly schedule the GPU kernel when it is adapted to ROS, because the CPU and GPU schedulers are independent. To satisfy

[2] $makespan$ is the principal measure of the performance for task scheduling heuristics algorithms and derives the finish time of the system.

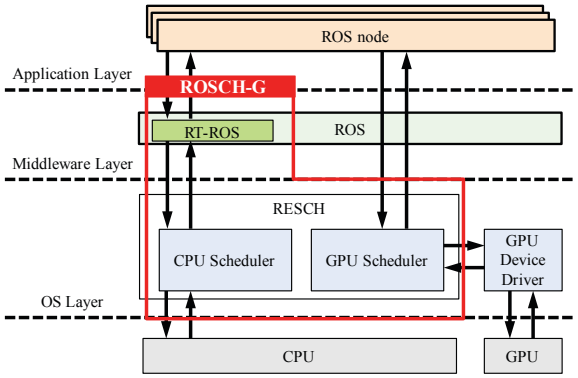


Fig. 3. ROSCH-G overview

the real-time processing of ROS on such a heterogeneous environment, the CPU and GPU schedulers must cooperate.

IV. Design and Implementation

In this section, the design and implementation of ROSCH-G, which provides a CPU/GPU coordination mechanism without modifications, are described. An architectural overview of ROSCH-G is given in Fig. 3. The ROSCH-G system architecture can be divided into two parts. As scheduling considering dependency is NP-hard, ROSCH-G first provides a CPU scheduler with an offline scheduling algorithm considering the execution order, called heterogeneous laxity-based ROS scheduling (HLBRS), improving on our previous work, HLBS. The CPU scheduler also realizes transparent scheduling through RT-ROS, which encapsulates real-time functions required for RESCH. The implementation of the CPU scheduler is provided in the kernel space by an LKM, enabling real-time mode to the $nodes$ from user space. These enable easy adaptation to a large-scale system based on ROS.

Second, a GPU scheduler based on RESCH realizes cooperative work with the CPU scheduler using APIs provided by ROSCH-G. The basic APIs for GPU scheduling are listed in Table I and an sample code with ROSCH-G APIs is shown in Fig. 4. Owing to the proprietary black box module and device driver provided by NVIDIA, embedding the function before and after launching GPU kernel function is the simplest solution. The proposed GPU scheduler provides a mechanism to preferentially process arbitrary GPU kernels and improves CPU utilization when multiple GPU kernels are launched simultaneously. Furthermore, combining the GPU scheduler with the CPU scheduler realizes more efficient ROS scheduling. In this work, it is assumed that GPU applications are written in CUDA; however, the concept of GPU resource management presented in this study is not limited by programming languages.

A. Offline DAG Scheduling Algorithm

Here, we introduce the graph attributes used to rank priorities. An application is represented by a DAG, $G = (V, E)$. In such a graph, V is the set of v tasks, E is the set of e edges between tasks, w_{ip} is the weight of task v_i on processor p, i.e.,

```
void gpu_task(){
 /* Initializations for launching a kernel function */
 ros_gsched_enqueue();
 /* Launch a GPU kernel */
 ros_gsched_dequeue();
 /* Finalization for a kernel function */
}
```

Fig. 4. Sample code with ROSCH-G APIs

the execution cost of task v_i, c_{ij} is the communication delay from task v_i to task v_j, D_i is the deadline attributed to an end node at the critical instant, and $\overline{w_i}$ and $\overline{c_{ij}}$ are the average of the w_i and c_{ij} dependent processors, respectively. It must be noted that the average is effective when the performance of the processors is different.

The proposed HLBRS algorithm schedules a $node$ by considering ROS as a DAG. HLBRS inherits the solution of different deadline requests that is characteristic of HLBS and solves for the period difference of entry nodes. The algorithm, which has ranking and assigning phases, computes the $laxity$ recursively to determine the assignment order of tasks, where the $laxity$ represents the rest time until the successor end $node's$ deadline. Here, the number of iterations is determined by the number of end nodes. After computing the $laxity$ of all the tasks, tasks are sorted in ascending order of $laxity$. Tasks with a smaller $laxity$ value are assigned higher priority. Finally, in the task allocation phase, prioritized tasks are assigned to processors based on the release time. The feasibility analysis is performed by repeating until all the deadlines in the above procedure become the harmonic period.

1) Task Ranking: The scheduler assigns a priority level to each task in a preprocessing step. Tasks are assigned the priority $laxity(v_i)$, which is defined recursively as follows:

- If v_i corresponds to the end node:

$$laxity(v_i) = D_i - \overline{w_{ip}} \tag{1}$$

- If v_i does not correspond to the end node:

$$laxity(v_i) = \min_{v_j \in succ(v_i)} (laxity(v_j) - \overline{c_{ij}}) - \overline{w_{ip}} \tag{2}$$

The $laxity$ is computed recursively by traversing the task graph from end $node$ to entry $node$. Tasks are assigned high priority in ascending order of laxity. This means that the task cannot afford the deadline that has been preferentially assigned by the processor. This ranking algorithm addresses Richard's Anomalies [27], which present a problem for fixed-priority methods in increasing $makespan$ when increasing the number of processors.

2) Task Allocation: In the task allocation phase, all prioritized tasks are allocated to processors in ascending order of laxity. I then add a restriction on the release time r_i to $earliest\ execution\ start\ time$ (EST) such that the proposed algorithm can schedule periodic tasks. The allocation of tasks to a processor is performed using EST and $earliest\ execution\ finish\ time$ (EFT).

TABLE I
BASIC SET OF APIS FOR GPU SCHEDULING

API	Description
ros_gsched_init()	Initializes exclusive control mechanism (definition is only once).
ros_gsched_enqueue()	Registers launching request. It must be called before the CUDA launch API (i.e., cuLaunchGrid()).
ros_gsched_dequeue()	Waits for completion of the GPU kernel.
ros_gsched_deinit()	Destroy exclusive control mechanism (definition is only once).

Algorithm 1 HLBRS algorithm

Input: Target application representing DAG $G(V, E)$.
Output: Scheduling of $G(V, E)$.

```
procedure HLBRS
    while Exist end nodes with no value of laxity do
        Compute laxity by traversing graph from end
        node to entry node.
        if The node already has a laxity then
            Assign smaller laxity to the node.
        end if
    end while
    Sort the tasks in ascending order of laxity.
    while All deadlines are not the harmonic period do
        while There are unscheduling nodes do
            Select the node has the smallest laxity.
            for Exist available processor do
                Compute EFT using the insertion-based
                scheduling policy.
            end for
            Assign the task to the processor indicating the
            minimum EFT.
        end while
    end while
end procedure
```

$$EST(v_i, H_p) = max(available(H_p, r_i) , \max_{v_j \in pred(v_i)} (EFT(v_j, host(T_j)) + c_{ji})) \quad (3)$$

$$EFT(v_i, H_p) = w_{ip} + EST(v_i, H_p) \quad (4)$$

The time $available(H_p, r_i)$ is the earliest time at which processor H_p is ready for task execution. The set of immediate predecessor tasks of task v_i is designated by $pred(v_i)$.

The pseudocode of the proposed HLBS is given in Algorithm 1. The time complexity of HLBRS regarding the number v of tasks, the number q of processors, and the number of loops until the hyperperiod h is $O(q \times v^2 \times h)$.

B. CPU/GPU Collaborative Scheduler

ROSCH-G provides a GPU scheduler, which addresses three problems for cooperative scheduling with the CPU scheduler. Furthermore, the implementation of ROSCH-G as loadable modules and the provided API's is on the internal procedure on ROS are expected to minimize the introduction cost of the real-time scheduler.

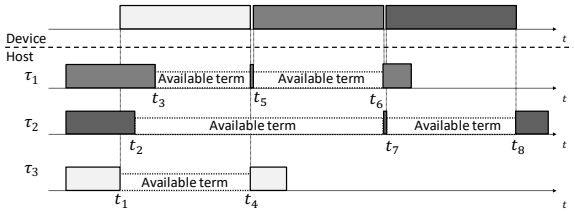


Fig. 5. Execution flow of GPU tasks on ROSCH-G (the priority of each task decreases in order from the top)

First, the GPU scheduler guarantees processing with the original execution time for a GPU kernel with high priority. To improve responsiveness for a high priority GPU task, the GPU scheduler exclusively controls the launch request of the GPU kernel at the host, because the response time of a GPU kernel is prolonged when multiple GPU tasks request the launching of GPU kernels to a single GPU device.

Second, it schedules the GPU kernel taking into account the dependency of ROS. The order of launching requests to be issued based on the priority given to the GPU task ($node$) by the algorithm is described in Section IV-A. Thus, the GPU kernel enables scheduling considering the entire system. This is important because even a single processing delay affects the entire system in ROS.

Finally, the GPU scheduler solves the CPU occupation problem of the GPU kernel. The GPU scheduler releases CPUs while running the GPU kernel, enables the CPU scheduler to collaborate in scheduling, and improves CPU utilization.

For instance, consider the example of the execution flow on ROSCH-G illustrated in Fig. 5. Here, three GPU tasks τ_1, τ_2, and τ_3 in the many-core environment have decreasing priorities while sharing a single GPU device. It should be noted that "Available term" in Fig. 5 indicates that its calling thread does not occupy the CPU by the CPU release mechanism of ROSCH-G. At the critical instant, if τ_3 is launching the GPU kernel and release CPU at time t_1, it might happen that a τ_2 request launching at time t_2 is blocked due to the exclusive control. Then, τ_2 is placed in a scheduling queue and suspended. At time t_3, τ_1 attempts to launch its GPU kernel, but it is also blocked because the GPUs is in use. Now, if τ_3 finishes its GPU kernel at time t_4, τ_1 with the highest priority in the scheduling queue launches the GPU kernel at time t_5. Finally, after finishing τ_1 at time t_6, τ_2 can launch its GPU kernel at time t_7 and finish at time t_8.

V. LOGICAL ANALYSIS

In this section, we firstly evaluated the proposed HLBRS algorithm using simulation and compared the proposed HLBRS

with HEFT [12] and HETS [14]. HETS is an algorithm that extends HEFT to minimize $makespan$. We introduce the metrics used for this performance evaluation in the following.

A. Performance Metrics

We use four metrics to analyze the performance. The metrics are based on metrics from the literature [12], [28]:

Scheduling Length Ratio

The scheduling length ratio (SLR) normalizes the scheduling length (i.e., $makespan$) to a lower bound. The task scheduling algorithm that yields the lowest SLR is considered to demonstrate the best performance. Here, SLR is defined as the ratio of $makespan$ to the sum of the computation times on the critical path (CP) and is calculated as follows:

$$SLR = \frac{makespan}{\sum_{v_i \in CP_{MIN}} min_{p_j \in Q}\{w_i\}}. \quad (5)$$

Note that the average SLR values over several task graphs were used in our experiments.

Speedup

$Speedup$ provides an index of how much faster a parallel execution time algorithm executes compared with a sequential execution time algorithm. Here, the sequential execution time is the cumulative computation costs of all tasks on a single processor that minimize $makespan$. $Speedup$ is derived as follows:

$$Speedup = \frac{min_{p_j \in Q}\{\sum_{v_i \in V} w_i\}}{makespan}. \quad (6)$$

Load Balancing

$Load\ balancing$ is an index of how well the scheduler takes advantage of multiple processors. LB is calculated using $makespan$ and the average (AVG) sum of processing times on each processor (i.e., AFT) and is obtained using Equations (7) and (9).

$$AVG = \frac{\sum_{p_j \in Q} AFT}{Np}. \quad (7)$$

$$LB = \frac{makespan}{AVG}. \quad (8)$$

Failure Ratio

We evaluate the performance regarding the $failure\ ratio$ (FR), which is defined as the ratio of the number of unschedulable task sets to the total number of task sets. FR is defined as follows:

$$\frac{the\ number\ of\ failure\ task\ sets}{the\ number\ of\ scheduled\ task\ sets}, \quad (9)$$

where the number of failure task sets is the set of tasks including any task that fails to satisfy the deadline constraints on the end nodes.

B. Simulation Setup

Task graphs for the experiments were generated using Task Graphs for Free (TGFF) [29] version 3.5. TGFF allows the generation of random DAGs according to parameters such as the number of tasks maximum indegree. Here, we set the parameters assuming an actual use environment as follows:

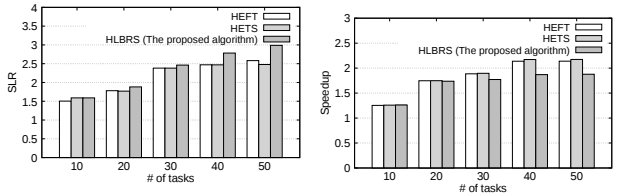
- Number of tasks in the graph (v). We set the number of tasks to $v = 10, 20, 30, 40, 50$.
- Indegree of a node $in_{degree} = 2$.
- Outdegree of a node $out_{degree} = 3$.
- Number of entry nodes $N_{entry} = 2$.
- Number of end nodes $N_{end} = 2$ (several end nodes that require different deadlines).
- Number of processors $N_p = 3$, and they have different capabilities.
- Communication to computation ratio (CCR) = 1.0. If CCR > 1, the larger the CCR, the larger the proportional communication time and the more obviously heterogeneous the communication. A high CCR is a marker of communication intensive processing. If CCR < 1, then computation dominates in the system. In these experiments, we set the communication to result in a computation ratio of CCR = 1.0.
- The deadline assigned to end node v_i is D_i. Deadlines are equal to the release periods of the entry nodes.

C. Comparison of performance metrics

We first present the experimental results for an average SLR of scheduling results by HEFT, HETS, and the proposed algorithm HLBRS. As shown in Fig. 6 (a), HEFT and HETS show better performance than HLBRS. This is because HEFT and HETS are specialized algorithms to shorten $makespan$, and these algorithms do not consider time constraints. For the same reason, the average $Speedup$ and LB values for HEFT and HETS were superior to those of HLBRS, as shown in Figs. 6 (b) and 6 (c). However, HLBRS's principal contribution is the preferential processing of $nodes$ with a strict time constraint and as shortened as possible. In real-time systems, it is also important to improve such performance metrics, but most important is to complete the processing of all $nodes$ within the time constraint. These evaluation results indicate that the proposed HLBRS does not significantly degrade performance on parallel processing.

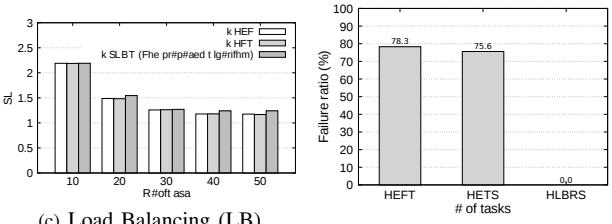
D. Comparison of FR

The failure ratio of scheduling was also demonstrated. This evaluation targets 5,000 task graphs with the number of tasks ranging from ten to 50, with each 100 times, in the above evaluation as shown in Fig. 6. This evaluation is also done under the condition that there is at least one algorithm that can produce a feasible schedule in three algorithms. As shown in Fig. 6 (d), the HLBRS algorithm outperformed the HEFT and HETS algorithms. This was an expected result, because the existing HETS algorithm was not specifically designed to schedule DAG tasks with time constraints. In HEFT and HETS, FR increased significantly, which was attributed to the



(a) Scheduling Length Ratio (SLR)

(b) Speedup



(c) Load Balancing (LB)

(d) Deadline Miss Ratio

Fig. 6. Comparison of the performance metrics

task order allocation. To shorten $makespan$, HEFT and HETS schedule all end $nodes$ to the last because the end $nodes$ do not affect subsequent tasks, whereas the other $nodes$ can cause blocking terms and affect the processing time of the entire system. In contrast, the proposed HLBRS algorithm, which considers time constraints in addition to shortening $makespan$, outperforms the existing HEFT and HETS. These evaluation results demonstrate that the proposed HLBRS can schedule DAG tasks with various time constraints.

VI. Experimental Evaluation

Real-time performance of ROSCH-G was evaluated. In the following, it is also demonstrated that the impact of the LKM-based real-time scheduler for GPU applications is acceptable. Finally, the performance of ROSCH-G with a real-world application was demonstrated.

A. Experimental Setup

Experiments were conducted using the Linux kernel 3.16.0, an NVIDIA GeForce GTX1080 GPU, a 3.40 GHz Intel Core i7 2600 (eight cores, including two hyperthreading cores), and an 8 GB main memory. The GPU application programs were written in CUDA and compiled using NVCC v6.0.1. The NVIDIA 331.62 driver with NVIDIA CUDA 8.0 was used.

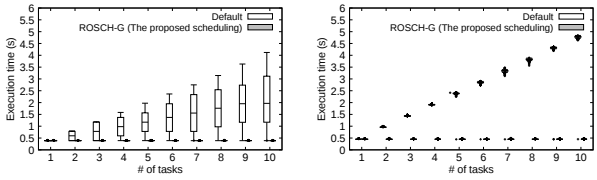
B. Variance of the GPU Kernel

The execution time of GPU kernels when multiple GPU kernels are launched to a single GPU simultaneously is also evaluated. This evaluation is conducted on GTX680 and GTX1080.

Fig. 7 indicates that the proposed GPU scheduler made the execution time of GPU kernels predictable, whereas the response time of a GPU kernel on conventional GPU architectures without any additional scheduling shown as Default tends to be proportional to the number of simultaneously launched GPU kernels. It is to be noted that GPU kernels on ROSCH-G could be completed within the original execution time regardless of the number of simultaneously launched GPU kernels and GPU types. This is quite important for schedulability analysis.

C. Impact of GPU scheduling

In this evaluation, an NVIDIA GeForce GTX680 is used for comparison with our previous work, which used Linux-RTXG.



(a) on Kepler architecture (GTX680)

(b) on Pascal architecture (GTX1080)

Fig. 7. Time variance of a GPU kernel

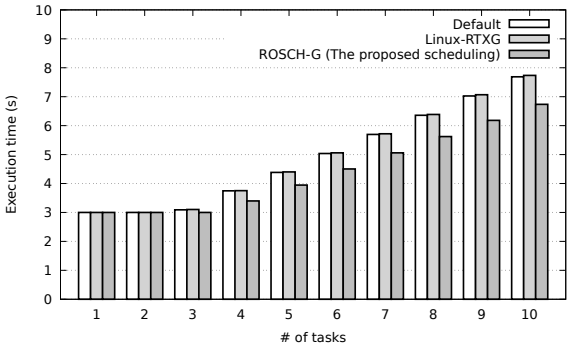


Fig. 8. Impact of GPU scheduling

Three schedulers, Default, Linux-RTXG, and ROSCH-G, were executed to measure overhead. Note that Default indicates that the scheduling provided a Linux kernel without any additional scheduler. The program, which provided the microbenchmark program provided by Linux-RTXG, generates multiple GPU tasks. With the CPU scheduling policy set to FIFO, all tasks release a job simultaneously, each of which includes data transfer between the CPU and GPU, followed by the execution of the GPU kernel.

We measured the execution time transition by increasing the number of GPU tasks. The impact of scheduling was as in Fig. 8. ROSCH-G demonstrated efficient scheduling compared with the other schedulers owing to having exclusive control and a releasing CPU mechanism. This figure indicates an improved response time in GPU-heavy systems, because the processing on the host while running the GPU kernel also increased.

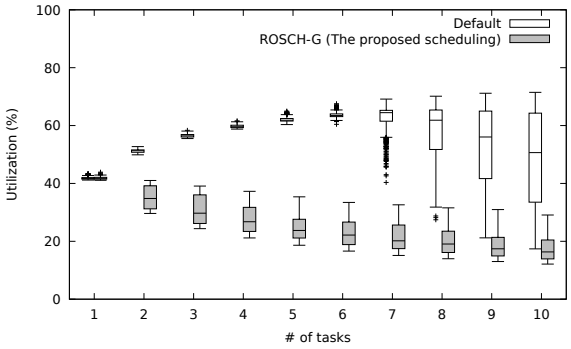
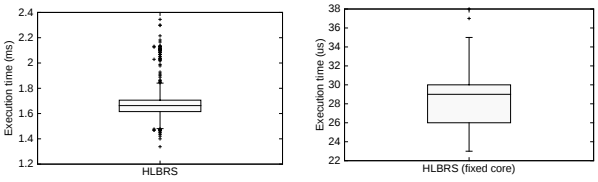


Fig. 9. Utilization of a GPU task



(a) Initialization overhead for HLBRS

(b) Scheduling overhead for HLBRS (fixed core)

Fig. 10. Overhead by HLBRS

D. Utilization test

In the same situation as for the evaluation of impacts of GPU scheduling, CPU utilization of the GPU task was also measured, as shown in Fig. 9. Default in the figure indicates the increment of CPU utilization and the variance in it. This is due to some GPU kernels are launched after finishing predecessor could finish processing within nearly its original execution time, because of the CPU occupation while running GPU kernel and the number of available CPUs.

On the other hand, the proposed ROSCH-G demonstrated a decrement of CPU utilization in addition to constant variance in it. The exclusive control proposed in ROSCH-G suppressed the variance of execution time, and the released CPU mechanism prevented an increment of CPU utilization.

E. Initialization and scheduling overhead for CPU scheduling

Overhead of the proposed HLBRS for CPU scheduling is evaluated. Then, we evaluated HLBRS under two allocation policies. HLBRS (fixed core) determines the affinity of the core and the priority of the node in advance using HLBRS, and HLBRS (Linux) determines the allocation core using the default scheduler of Linux. The proposed HLBRS (fixed core) and HLBRS (Linux) are initialized to give a priority. Fig. 10 (a) indicated significant overhead, but the application program is not significantly affected by this procedure, because it is called only once at the beginning. HLBRS (fixed core) also incurs the scheduling overhead (Fig. 10 (b)) for specifying cores in every cycle, whereas HLBRS (Linux) incurs only the initialization overhead. This evaluation, especially in HLBRS (Linux), indicated that the influence by the scheduling is considerably suppressed.

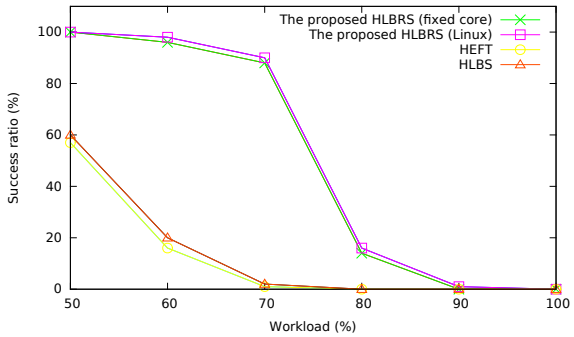


Fig. 11. Success ratio under scheduling algorithms for only CPU

F. Workload Test

Here, the performance of ROSCH-G with a real-world application is demonstrated. The demonstration was performed with a GPU-accelerated object detection program [2]. We tested the proposed HLBRS, HEFT [12], and HETS [14] for each sampling system load, gradually increased by 10 % beginning from 50 % and ending with 100 %. Each task set is scheduled for approximately ten min. As shown in Fig. 11, HEFT and HETS do not satisfy real-time constraints under high workload because conventional offline scheduling algorithms are not optimized for DAG with multiple deadlines. On the other hand, the proposed algorithms reached an approximately 89 % success rate until reaching a 70 % workload. Meanwhile, ROSCH-G (fixed core) requires memory to hold the allocation result of HLBRS. Furthermore, ROSCH-G (fixed core) can incur inefficient allocation, because the allocation analysis of HLBRS utilizes pessimistic assumptions using WCET. Consequently, ROSCH-G (Linux) can schedule more flexibly instead of guaranteeing schedulability.

Then, we added the proposed GPU scheduler and evaluated under the same conditions. Note that ROSCH-G (fixed core) and ROSCH-G (Linux) correspond to HLBRS (fixed core) and HLBRS (Linux), respectively. Default indicates the scheduling provided by the Linux kernel without any additional scheduler. Critical path method (CPM) preferentially processes tasks on the critical path. Rate monotonic (RM), without considering the execution order restrictions, assigns a priority according to an entry node' s cycle duration, and tasks depended on entry node with a shorter cycle duration are preferentially processed. As shown in Fig. 12, the proposed algorithms reached an approximately 97 % success rate until reaching a 70 % workload. In addition, ROSCH-G (Linux) without specifying core indicated that the GPU scheduler could be more effectively used.

VII. Conclusion

This paper proposed ROSCH-G, a real-time ROS extension on transparent CPU/GPU coordination mechanism. ROSCH-G can schedule a *node* on ROS without modifying the OS kernel, device driver, and user applications. In addition, the proposed GPU scheduler, which works cooperatively with a CPU scheduler, can schedule a GPU kernel and improve CPU utilization. An experimental evaluation of ROSCH-G

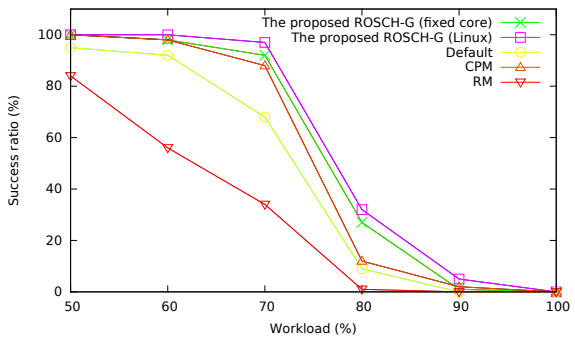


Fig. 12. Success ratio under CPU/GPU cooperative scheduling

indicates improvement for the real-time performance of ROS in a heterogeneous environment, including GPUs. To the best of my knowledge, this is the first study to schedule CPU/GPU tasks under the restrictions of the publish/subscribe model. We expected that the principal contribution of this paper would be a cornerstone to open up many problems relative to the real-time dataflow model, e.g., ROS. ROS has multiple inputs, outputs, and deadline constraints; thus, it differs from existing DAG structures.

Considering the trend of the latest technology, the proposed GPU scheduler leaves room for improvement. The GPU programming model moves from a synchronous model to an asynchronous model. In this paper, we have exclusive control of the GPU kernel, but this might not be suitable for an asynchronous programming model.

ACKNOWLEDGMENT

This work was partially supported JST PRESTO Grant Number JPMJPR1751.

REFERENCES

[1] "Waymo," https://waymo/, accessed: 2018-03-26.
[2] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems," *In Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2018.
[3] "Open Source Robotics Foundation (OSRF)," https://github.com/ros2, accessed: 2018-03-26.
[4] Y. Saito, T. Azumi, S. Kato, and N. Nishio, "Priority and Synchronization Support for ROS," *In Proceedings of the 4th IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pp. 77–82, 2016.
[5] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, *A generalized parallel task model for recurrent real-time processes*, 2012, pp. 63–72.
[6] J. Li, Z. Luo, D. Ferry, K. Agrawal, C. Gill, and C. Lu, "Global EDF Scheduling for Parallel Real-time Tasks," *Real-Time Syst.*, vol. 51, no. 4, pp. 395–439, Jul. 2015.
[7] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility Analysis in the Sporadic DAG Task Model," in *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.
[8] 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)*, 2014, pp. 13:13–13:22.
[9] "ROSCH: Real-time extension modules for ROS," https://github.com/CPFL/ROSCH, accessed: 2018-03-26.
[10] S. Kato, R. Rajkumar, and Y. Ishikawa, "A loadable real-time scheduler suite for multicore platforms," Tech. Rep., 2009.
[11] M. Asberg, T. Nolte, S. Kato, R. Rajkumar, and Y. Ishikawa, "ExSched: An External CPU Scheduler Framework for Real-Time Systems," Tech. Rep., 2009.
[12] H. Topcuouglu, S. Hariri, and M. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, 2002.
[13] K. C. Huang, Y. L. Tsai, and H. C. Liu, "Task Ranking and Allocation in List-based Workflow Scheduling on Parallel Computing Platform," *The Journal of Supercomputing*, vol. 71, no. 1, pp. 217–240, 2015.
[14] Anum Masood, Ehsan Ullah Munir, M. Mustafa Rafique, Samee Ullah Khan, "HETS: Heterogeneous Edge and Task Scheduling Algorithm for Heterogeneous Computing Systems," *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, vol. 13, no. 3, pp. 1865–1870, 2015.
[15] R.M, Pathan, P.Voudouris, and P.Stenstrom, "Scheduling Parallel Real-Time Recurrent Tasks on Multicore Platforms," in *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 4, 2017, pp. 915–928.
[16] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira, "DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm," in *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2010, pp. 27–34.
[17] M. Gary and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1979.
[18] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.
[19] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the Performance of ROS2," in *Proceedings of the 13th International Conference on Embedded Software (EMSOFT)*. ACM, 2016.
[20] Y.Suzuki, T. Azumi, N. Nishio, and S. Kato, "HLBS: Heterogeneous Laxity-Based Scheduling Algorithm for DAG-Based Real-Time Computing," in *Proceedings of the 4th IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, 2016, pp. 83–88.
[21] R. J. Bril, J. J. Lukkien, and W. F. Verhaegh, "Worst-case Response Time Analysis of Real-time Tasks Under Fixed-priority Scheduling with Deferred Preemption," *Real-Time Syst.*, vol. 42, no. 1-3, pp. 63–119, Aug. 2009.
[22] J.J.K. Park, Y. Park, and S. Mahlk, "Chimera: Collaborative Preemption for Multitasking on a Shared GPU," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2015, pp. 593–606.
[23] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class GPU Resource Management in the Operating System," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC)*, 2012, pp. 37–37.
[24] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A framework for real-time GPU management," in *Proceedings of the 34rd IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2013, pp. 33–44.
[25] W. Han, H. Bae, H. Kim, J. Lee, and I. Shin, "GPU-SPARC: Accelerating parallelism in multi-GPU real-time systems," Tech. Rep., 2014.
[26] Y. Suzuki, Y. Fujii, T. Azumi, N. Nishio, and S. Kato, "Real-Time GPU Resource Management with Loadable Kernel Modules," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 6, pp. 1715–1727, Jun. 2017.
[27] J.A. Stankovic, M. Spuri, M.D. Natale, G.C. Buttazzo, "Implications of Classical Scheduling Results for Real-Time Systems," *IEEE Computer*, pp. 16–25, 1995.
[28] R. Rajak, S. Gupta, G. K. Singh, and S. Jain, "A Novel Approach for Task Scheduling in Parallel Computing using Priority Attributes," *Smart CR*, vol. 5, no. 3, pp. 356–367, 2015.
[29] "TGFF Download Area," http://ziyang.eecs.umich.edu/dickrp/tgff/download.html, accessed: 2018-03-26.