Real-Time GPU Resource Management with Loadable Kernel Modules

Yuhei Suzuki, Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, and Shinpei Kato

Abstract—Graphics processing unit (GPU) programming environments have matured for general-purpose computing on GPUs. 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 operating system modules to support real-time GPU resource management. Unfortunately, such a system extension 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. A loadable kernel module (LKM) framework, called Linux Real-Time eXtention with GPUs (Linux-RTXG), for managing real-time GPU resources with Linux without modifying the OS kernel and device drivers is proposed and evaluated experimentally. Linux-RTXG provides mechanisms for interrupt interception and independent synchronization to achieve real-time scheduling and resource reservation capabilities for GPU applications on top of existing device drivers and runtime libraries. Experimental results demonstrate that the overhead incurred by introducing the proposed Linux-RTXG is comparable to that of introducing existing kernel-dependent approaches. In addition, the results demonstrate that multiple GPU applications can be scheduled successfully by Linux-RTXG to meet their priority and quality-of-service requirements in real time.

Index Terms—GPU, resource management, scheduling, real-time systems, operating systems

1 Introduction

Graphics processing units (GPU) are common platforms for various data-parallel and compute-intensive applications. Although GPUs are primarily used to accelerate high-performance computing applications, their performance advantage for real-time systems has been increasingly recognized. Examples of such GPU applications include route navigation for autonomous vehicles [1], object detection [2], plasma control for fusion reactors [3], windowing applications [4], and database operators [5]. Benchmarking suites for various workloads [6] are also well deployed.

GPU applications, whose main 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, system software to support real-time management of GPU resources is becoming a more significant requirement. The standard system software packages released by GPU vendors and communities, such as CUDA [7] and OpenCL [8], are not tailored to support real-time systems; thus, extending such system software (including the operating-system (OS) kernel and device drivers) is complex.

Previous studies have investigated GPU resource management for real-time systems. For example, TimeGraph [9] provides GPU scheduling and resource reservation capabilities at the device driver level, multiplexing GPU commands to support priorities, and quality of service (QoS) for GPU applications. Gdev [15] is a rich set of runtime libraries and device drivers that achieves first-class GPU resource management, where GPU contexts are fully managed in the OS space. The main drawback of these methods is that they require detailed information about the implementation of the GPU runtime libraries and device drivers, which is primarily obtained through reverse engineering.

Generally, a GPU software stack is encapsulated by an application programming interface (API) in runtime libraries and an application binary interface (ABI) in device drivers. This means that modifications to system software outside of the API and ABI layers allow the entire software stack to be loadable, thereby enabling more sustainable solutions. CPU scheduling is also important for GPU applications because CPU time is consumed when GPU functions are launched from the API and ABI layers. Therefore, to satisfy real-time requirements for GPU computing, the management of CPU and GPU resources must be coordinated.

GPUSync, proposed by Elliott et al., attempts to coordinate CPU and GPU resource management for real-time systems [11], [12]. GPUSync was intended to extend CPU scheduling for multiple GPU-aware contexts with budget enforcement. GPUSync was built on top of the API and ABI

- *Y. Suzuki is with the Graduate School of Information Science and Engineering, Ritsumeikan University, Kyoto 603-8577, Japan. E-mail: gakky@ubi.cs.ritsumei.ac.jp.*
- *Y. Fujii is with the NTT Software Innovation Center, NTT Corporation, Tokyo 100-0011, Japan. E-mail: fujii.yusuke@lab.ntt.co.jp.*
- *T. Azumi is with the Graduate School of Information Science and Engineering, Osaka University, Osaka 565-0871, Japan. E-mail: takuya@sys.es.osaka-u.ac.jp.*
- *N. Nishio is with the College of Information Science and Engineering, Ritsumeikan University, Kyoto 603-8577, Japan. E-mail: nishio@cs.ritsumei.ac.jp.*
- *S. Kato is with the Graduate School of Information Science and Technology, University of Tokyo, Tokyo 113-8654, Japan. E-mail: shinpei.kato@gmail.com.*

Manuscript received 27 Apr. 2016; revised 10 Nov. 2016; accepted 13 Nov. 2016. Date of publication 18 Nov. 2016; date of current version 17 May 2017. Recommended for acceptance by B. He.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2630697

layers of proprietary software to provide a configurable framework that can verify the combination of task allocation policies for multicore CPUs and GPU resource allocation policies for multiple GPUs.

GPUSync is implemented using $LITMUS^{RT}$ [13], which introduces a significant number of changes to the OS kernel. TimeGraph and Gdev also make some modifications to the device driver. Such built-in approaches to the OS kernel and device drivers require users to install patches, and developers must maintain patches to remain compatible with new versions. This so-called "porting" work is complex and not sustainable given that open-source software, such as Linux, is frequently updated with non-trivial code changes.

Linux supports the loadable kernel module (LKM), which can load and unload kernel modules to add and remove supplementary kernel functions. Existing Linux-based real-time OSs (RTOSs) use an LKM to extend the real-time capabilities of Linux. RESCH [14], [15], a loadable real-time scheduler suite for Linux, provides ExSched, a real-time scheduling framework that is completely independent of code modifications to the OS kernel and device drivers. Unfortunately, RESCH does not support GPU resource management. It has never been demonstrated that GPU resource management for real-time systems can be fully implemented using an LKM without modifying the OS kernel and device drivers.

Contribution: We proposed an LKM framework, which we refer to as Linux Real-Time eXtention with GPUs (Linux-RTXG), that provides LKM-based real-time GPU resource management. Linux-RTXG allows a system to reconfigure scheduling algorithms easily and install their modules at runtime for GPU applications. The most significant contribution of Linux-RTXG is that resource management modules for CPUs and GPUs can be added to Linux without modifying the OS kernel or device drivers. CPU mechanisms for scheduling and resource reservation are based on RESCH, and GPU mechanisms for scheduling and resource reservation are implemented using Gdev. In addition to integrating RESCH and Gdev, Linux-RTXG provides a new framework to coordinate CPU and GPU resource management, thereby freeing them from the built-in OS kernel and device drivers. Thus, Linux-RTXG represents a completely "patchless" approach.

Organization: The remainder of this paper is organized as follows. Section 2 describes the system model and basic approaches for Linux-RTXG. With specific emphasis on GPU scheduling, Section 3 presents the design and implementation of Linux-RTXG. Section 4 presents the results of an evaluation of the system overhead and reservation performance of Linux-RTXG. Section 5 discusses related work, and Section 6 concludes the paper.

2 System Model

A GPU programming and scheduling model is assumed and described as follows. To highlight an unsolved problem concerning GPU resource management, existing work is introduced first. In addition, factors that have prevented GPU resource management from patchless implementation are described. Note that, regarding the model, the system is assumed to comprise multiple GPUs and multicore CPUs.

2.1 GPU Programming Model

General-purpose computing on GPUs (GPGPU) is supported by special programming languages, such as CUDA and OpenCL. 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. Conceptually, the contribution of this work is applicable to OpenCL and other languages. A GPU task is defined as a process running on the CPU that launches a GPU kernel to the GPU. Here the cyclic execution unit is referred to as a "GPU job." The GPU kernel is the process executed on the GPU. Multitasking environments in which multiple GPU application tasks can coexist are also assumed; however, the current GPU programming model does not allow different application contexts to be executed on the GPU simultaneously.[1] In other words, different GPU contexts can be created and launched, but they must be executed exclusively on the GPU.

GPU programming requires a set of APIs provided by runtime libraries, such as the CUDA Driver API and the CUDA Runtime API. A typical approach to GPU programming follows several steps: (i) *cuCtxCreate* creates a GPU context, (ii) *cuMemAlloc* allocates memory space to device memory, (iii) *cuModuleLoad* and *cuMemcpyHtoD* copy the data, and the GPU kernel from the host memory to the memory space of the allocated device, (iv) *cuLaunchGrid* invokes the GPU kernel, (v) *cuCtxSynchronize* synchronizes a GPU task to wait for the completion of the GPU kernel, (vi) *cuMemcpyDtoH* transfers the data back to the host memory from the device memory, and (vii) *cuMemFree*, *cuModuleUnload*, and *cuCtxDestroy* release the allocated memory space and the GPU context.

2.2 GPU Scheduling

Problems related to resource management in many variants of RTOSs [16], [17], [18], [19] have been addressed. Linux-based RTOSs are of particular interest [13], [14], [20], [21], [22]. GPUs are also supported by Linux. In this study, it is assumed that the OS architecture is Linux.

To meet real-time requirements in multitasking environments, RTOSs should provide *scheduling* and *resource reservation* capabilities. Rate Monotonic (RM) and Earliest Deadline First (EDF) [23] are well-known algorithms for priority-driven scheduling for real-time systems. There are many variants of resource reservation algorithms, such as the Constant Bandwidth Server (CBS) [24] and Total Bandwidth Server (TBS) [25].

GPU computing must deal with data transfer bandwidth and compute cores as shared resources. Therefore, in real-time systems, scheduling and resource reservation for the GPU and CPU must be considered. TimeGraph and Gdev are involved in solving problems concerning GPU resource management; however, they do not adequately consider the CPU time required for a GPU task to drive APIs. To support GPUs in real-time systems, the OS scheduler must be able to manage GPU tasks in coordination with CPU tasks on the host side while monitoring GPU time for completion of GPU kernels. Therefore, the problem of CPU and GPU

1. Concurrent kernel execution within the same application context has been supported by recent GPU hardware architectures.

coordinated scheduling is considered in terms of resource reservation mechanisms.

The recently developed GPUSync framework employs CPU and GPU coordinated scheduling. With GPUSync, the device driver and runtime library for GPU computing are hidden in black-box modules released by GPU vendors. On the other hand, TimeGraph and Gdev use reverse-engineered open-source software to address this problem. Both approaches are limited to some extent. Black-box modules make it difficult to manage the system in a fine-grained manner. Open-source software tends to lack some functionality owing to incomplete reverse engineering. GPUSync incorporates black-box modules in scheduling and resource reservation by arbitrating interrupt handlers and runtime accesses. This approach is preferred in the sense that a reliable proprietary driver and library can be utilized and functions for scheduling and resource reservation can be provided.

GPU Synchronization: Given that a GPU is a coprocessor connected to a host CPU, synchronization between the GPU and the CPU must be considered to guarantee correct execution logic. Most GPU architectures support two synchronization mechanisms. The "FENCE" mechanism is based on memory-mapped registers. To use FENCE, special commands must be sent to the GPU when a GPU kernel is launched so that the GPU can write the specified value to this memory-mapped space when the GPU kernel is completed. On the host side, a GPU task polls this value via the mapped space. The other technique is interrupt-based synchronization called "NOTIFY." Similar to FENCE, to use NOTIFY, it is necessary to send special commands to the GPU. Rather than writing to memory-mapped registers, NOTIFY raises an interrupt from the GPU to the CPU and simultaneously writes associated values to the GPU I/O registers. On the host side, a GPU task is suspended and waits for the interrupt. NOTIFY allows other tasks to use CPU resources while the GPU waits for its GPU kernel to complete execution. However, this process incurs scheduling overhead. FENCE is easier to use and more responsive than NOTIFY, but it is implemented at the expense of CPU utilization. More details about GPU architectures can be found in previous work [9], [10], [26].

To synchronize the CPU and the GPU, Gdev supports both NOTIFY and FENCE. NOTIFY is primarily used to schedule GPU tasks, and FENCE is used in driver-level synchronization. With Gdev, as mentioned previously, implementation of GPU synchronization involves additional commands to the GPU. Therefore, the device driver must be modified to a certain extent. GPUSync indirectly utilizes the NOTIFY technique with tasklet interception [27] on top of proprietary black-box modules. Tasklet refers to Linux's softirq implementation. GPUSync identifies the interrupt that has invoked a GPU kernel using a callback pointer with a tasklet.

LKMs: The main concept of the proposed system is to enable both the CPU and GPU to be managed by the OS scheduler without changing the OS kernel and device drivers. CPU scheduling under this constraint has been demonstrated by RESCH [14], [15]. To schedule GPU tasks, it must be possible to hook scheduling points when the preceding GPU kernel is completed. Then, the active context is switched to the next GPU kernel. The scheduling points can be hooked by two methods. The API-driven method (RGEM [28]) explicitly activates the scheduler after GPU synchronization invoked by the API, such as $cuCtxSynchronize()$. On the other hand, the interrupt-driven method used by TimeGraph and Gdev employs interrupts that can be configured by NOTIFY. GPUSync is also based on this interrupt-driven method. With CUDA, the standard $cuCtxSynchronize()$ API synchronizes completion of all GPU kernels. Therefore, the API-driven method can be used if a GPU context issues no more than a single GPU kernel. In other words, if a GPU task invokes multiple GPU kernels, the interrupt-driven method is more appropriate to realize real-time capabilities.

The interrupt-driven method forces TimeGraph, Gdev, and GPUSync to modify the Linux kernel or device drivers. This modification is required because the interrupt service routine (ISR) must be managed to create scheduling points. Gdev employs independent-synchronization mechanisms on top of the proprietary software; however, for scheduling and resource reservation with Gdev, the Linux kernel still requires some modification. Consequently, the available release versions of the Linux kernel and device drivers for TimeGraph, Gdev, and GPUSync are very limited. A core challenge of this work is to develop independent-synchronization mechanisms that do not require modification to the OS kernel and device drivers so that a coordinated CPU and GPU resource-management scheme can be utilized with a wide range of Linux kernels and device drivers.

Scope and Limitation: GPU resource management has a "non-preemptive" nature. For example, the execution of GPU kernels is non-preemptive. Data transfer between the CPU and GPU is also non-preemptive. Previous work has addressed the problem of response time by preventing overrun that occurs when the kernel is divided [29], [30]. The most difficult problem is scheduling of self-suspending tasks because the GPU is treated as an I/O device. For example, GPU tasks are suspended until their GPU kernels are completed. This "self-suspending" problem was presented as an NP-hard problem in previous work [31], [32]. However, considerable work on the scheduling of self-suspending tasks is ongoing [33], [34].

Note that this GPU "schedulability" problem is beyond the scope of this work. We focus on the design and implementation of GPU scheduling and resource reservation with existing algorithms and limit our investigation to time resources rather than memory resources. In other words, GPU-scheduling problems rather than device memory allocation problems are considered in the following.

The proposed system supports efficient data transfer between a CPU and GPU using GPU microcontrollers [35]; however, that support is not considered a contribution of this work. Our prototype system is limited to a Linux system and the CUDA environment; however, the concept is applicable to other OSs and programming languages if they support the FENCE and NOTIFY primitives.

3 Design and Implementation

In this section, the design and implementation of LinuxRTXG, which provides a framework for CPU and GPU coordinated scheduling based on the LKM, are described. The approach taken for GPU scheduling and its integration to CPU scheduling are explained. Owing to space limitations,

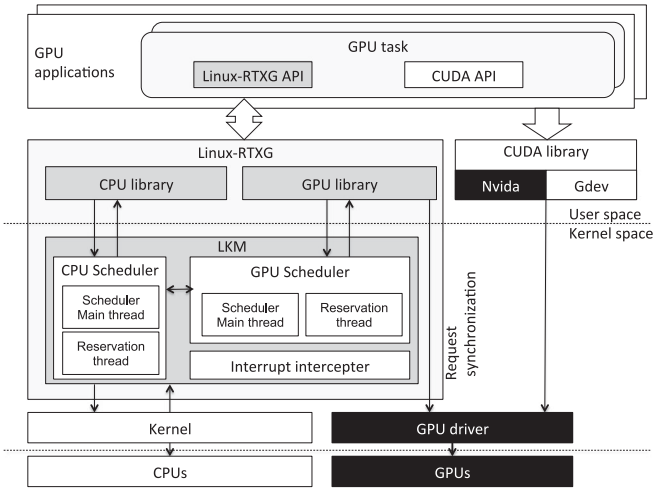


Fig. 1. Architectural overview of Linux-RTXG.

LKM-based CPU scheduling is not considered but can be found in literature related to the RESCH project [14], [15].

3.1 Linux-RTXG

An architectural overview of Linux-RTXG is shown in Fig. 1. The Linux-RTXG system architecture can be divided into two parts. First, the Linux-RTXG core contains a CPU scheduler and a GPU scheduler with a resource reservation mechanism. The implementation of the Linux-RTXG core is provided in the kernel space by an LKM. Thus, it can use exported Linux kernel functions, such as $schedule()$, $mod\_timer()$, $wake\_up\_process()$, and $set\_cpus\_allowed\_ptr()$. These functions can be called from the user space interface by the input/output control (ioctl) system call, which is a standard system call for device drivers. Second, the Linux-RTXG library provides an independent-synchronization method for coordinated management of CPU and GPU resources. The independent-synchronization method can be used on top of both proprietary [7] and open-source drivers [36]. Note that this method is required to manage interrupts for GPU scheduling without modifying the OS kernel and device drivers.

3.2 GPU Scheduling

Linux-RTXG is primarily based on the interrupt-driven method for GPU synchronization and is partly based on the API-driven method. The scheduler is invoked only when computation requests are submitted. The basic APIs supported by Linux-RTXG are listed in Table 1. Note that some APIs have arguments and others do not. Linux-RTXG is independent of GPU runtimes; therefore, the existing CUDA API does not need to be modified to cope with proprietary software. However, CUDA applications must add the Linux-RTXG APIs to use the functionality of Linux-RTXG. A sample code that includes the Linux-RTXG APIs is shown in Fig. 2. GPU tasks are provided with function calls to Linux-RTXG at strategic points.

```
void gpu_task(){
 /* variable initialization */
 /* calling RESCH API */
 dev_id = rtx_gpu_device_advice(dev_id);
 cuDeviceGet(&dev, dev_id);
 cuCtxCreate(&ctx, SYNC_FLAG, dev);
 rtx_gpu_open(&handle, vdev_id);
 /* Module load and set kernel function */
 /* Device memory allocation */
 /* Memory copy to device from host */
 rtx_gpu_launch(&handle);
 cuLaunchGrid(function, grid_x, grid_y);
 rtx_gpu_notify(&handle);
 rtx_gpu_sync(&handle);
 /* Memory copy to host from device */
 /* Release allocated memory */
}
```

Fig. 2. Sample code with Linux-RTXG APIs.

The GPU scheduler works cooperatively with CPU scheduler even though the GPU scheduler is implemented independently of the CPU scheduler. The execution flow of tasks in Linux-RTXG processing is as follows. First, the CPU scheduler schedules all tasks, including the GPU task, because the GPU kernel is invoked on the CPU. When a GPU task requests to launch the GPU kernel, the GPU scheduler manages GPU kernel launch times independent of the CPU scheduler. A GPU task that requests a GPU kernel launch occupies the CPU until the processing result of the GPU kernel is returned. In Linux-RTXG, to allow cooperative execution of other runnable tasks, the independent-synchronization method suspends GPU tasks, thereby allowing the GPU scheduler to allocate tasks.

The execution flow of GPU tasks managed by the Linux-RTXG APIs is shown in Fig. 3. In Fig. 3, GPU task G0 is running, and GPU task G1 requests to launch the GPU kernel.

TABLE 1
Basic Set of APIs for Linux-RTXG

API	Description
rtx_gpu_open()	Registers itself to Linux-RTXG and creates a scheduling entity. It must be called first.
rtx_gpu_device_advice()	Obtains recommendations for which GPU devices to use.
rtx_gpu_launch()	Controls GPU kernel launch time (i.e., a scheduling entry point). It must be called before the CUDA launch API (i.e., cuLaunchGrid()).
rtx_gpu_sync()	Waits for completion of the execution of the GPU kernel, i.e., it maintains a TASK UNINTERRUPTIBLE status.
rtx_gpu_notify()	Sends a NOTIFY/FENCE command to the GPU. Either FENCE or NOTIFY is selected by a flag set by an argument.
rtx_gpu_close()	Releases the scheduling entity.

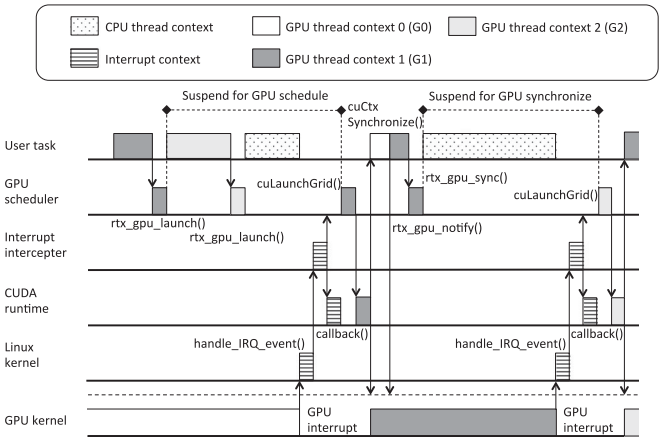


Fig. 3. Execution flow of GPU tasks.

GPU task G1 is suspended by rtx_gpu_launch() so that other user tasks can be executed. After this function call, the GPU scheduler controls the GPU kernel launch time. When the G0 GPU kernel is completed, an interrupt is launched by the GPU, and the corresponding interrupt handler is executed by the Linux kernel. Then, the interrupt interceptor activates some suspended tasks based on thier priority. The activated task G1 proceeds to launch the GPU kernel using a CUDA API, such as cuLaunchGrid(). After the kernel is launched, task G1 registers NOTIFY to set up an interrupt using rtx_gpu_notify(), and it reenters sleep mode until it receives the interrupt by rtx_gpu_sync(). The subsequent task is dispatched by the GPU scheduler, which is called by the interrupt from the GPU. Linux-RTXG manages the task execution order according to this flow.

A hierarchical scheduling mechanism that applies the concept of virtual GPUs to combine specified GPU tasks as a group is presented in the following. Virtual GPUs are created using a resource reservation mechanism, and GPU scheduling uses a priority mechanism. For resource reservation, virtual GPUs are activated by specifying the weights of GPU resources (e.g., memory bandwidth, compute bandwidth, and physical memory). Specifically, each invocation of the GPU kernel is associated with a scheduling entity, and Linux-RTXG allocates the scheduling entities to virtual GPUs. Note that virtual GPUs can belong to any physical GPUs. With Linux-RTXG, computing resources are distributed to virtual GPUs. Furthermore, a single virtual GPU can be allocated multiple GPU tasks. For example, if VGPU0, whose utilization of GPU resources is 30 percent, has three GPU tasks, these tasks are given a total of 30 percent utilization.

The virtual GPU scheduling pseudocode, which we refer to as BAND scheduling[2] [10] is shown in Fig. 4. This code assumes that rtx_gpu_open() has been called and associated with a virtual GPU. Initializing GPU resources for a virtual GPU is performed prior to launching the GPU kernel, and users can rewrite the utilization during scheduling using

2. BAND scheduling is based on Credit scheduling [37] for a virtual CPU and addresses the problem whereby the Credit scheduler fails to maintain the desired bandwidth for the virtual GPU. This is attributed to the fact that it presumes a preemptive constantly working CPU workload.

```
se: Scheduling entity
se->vgpu: Group that belongs to se
se->task: Task that is associated with se
vgpu->parent: Physical GPU identifier

/* in rtx_gpu_launch() */
void on_arrival(se) {
 check_permit_vgpu(se->vgpu);
 while(!check_permit_se(se)){
   enqueue(se->vgpu,se);
   sleep_task(se->task);
 }
}
 /* in rtx_gpu_sync() */
void on_completion(se) {
 reset_the_permit(se->vgpu, se);
 n_vgpu = pick_up_the_next_vgpu(se->vgpu->parent);
 se = pick_up_the_next_se(n_vgpu);
 if(se) {
   dequeue(se->vgpu,se);
   wake_up_task(se->task);
 }
 set_the_permit(se->vgpu, se);
}
```

Fig. 4. Virtual GPU scheduling pseudocode.

procfs provided by Linux. When a GPU task requests to launch the GPU kernel, on_arrival() is called by rtx_gpu_launch(). Check_permit_vgpu() checks whether the virtual GPU has permission to execute. Bandwidth is then assigned to the virtual GPU by the BAND scheduler. Check_permit_se() checks whether the corresponding thread has permission to execute. When execution permission is confirmed, the GPU task is placed in a wait queue and suspended, whereas on_completion() is called by rtx_gpu_sync() when the launched GPU kernel is completed. Pick_up_the_next_vgpu() calls the BAND scheduler and picks up the next virtual GPU after reset_the_permit() resets permission. After the next GPU thread has been activated, the BAND scheduler updates the budgets using set_the_permit().

3.3 GPU Synchronization

Here, the independent-synchronization and interrupt interception mechanisms are described. The independent-synchronization mechanism invokes NOTIFY and FENCE without using the GPU runtime API. The interrupt interception mechanism enables interrupt-driven invocation of the scheduler without modifying either the OS kernel or device drivers. As a results, Linux-RTXG does not need to modify the OS kernel and device drivers, but it can create the scheduling points for GPU tasks.

Independent-Synchronization Mechanism: The independent-synchronization mechanism using NOTIFY and FENCE is described as follows. This mechanism invokes an interrupt using NOTIFY and writes the fence value using the GPU microcontrollers and FENCE. NVIDIA's proprietary software uses the ioctl interface to communicate with the kernel space and user space. These ioctl interfaces provide driver functions, such as device memory allocation, obtaining

GPU states, and memory mapping. Gdev provides a runtime library that can control the GPU on top of NVIDIA's proprietary driver using these ioctl interfaces. The mechanism also uses an ioctl interface similar to Gdev to send commands to the GPU. Specifically, the mechanism is divided into initialization and notification processes.

The initialization process generates a dedicated GPU context. This process creates a virtual address space, allocates an indirect buffer object for commands, and creates a context object that must employ the FIFO engine, followed by allocation of a kernel memory object and mapping of the FIFO engine registers to host memory space through a memory-mapped I/O. The FIFO engine is a GPU microcontroller that decodes and dispatches the commands sent from the host CPU.

The notification process sends commands to a GPU compute engine or a GPU data copy engine using the $iowrite$ function associated with the mapped FIFO engine registers so that an interrupt will be raised from the GPU to the CPU. The compute engine and the data copy engine are GPU microcontrollers that control the states of GPU computation and data transfer. They are also used to switch GPU contexts for GPU computation and data transfer. Note that this independent-synchronization mechanism requires information about the ioctl interfaces. Therefore, it depends on the GPU architecture and the implementation of device drivers.

Interrupt Interception: Interrupts are handled by the ISR registered to the Linux kernel by the device driver. The scheduler function must receive the interrupts and identify them by reading the GPU status register. The GPU status register must be read by the OS scheduler before it is reset by the ISR. However, in interrupt interception, it is difficult to specify the GPU factors precisely because this is a proprietary driver. To build the interrupt generation factor, we reverse engineered the existing open-source driver Nouveau.

The Linux kernel has a structure called irq_desc that holds interrupt parameters for each interrupt number. This structure has an internal sub-structure called irq_action, which includes the ISR callback pointer. The irq_desc structure is allocated to global kernel memory space and is freely accessible from the kernel space. Therefore, the Linux kernel and the external LKMs can obtain information about irq_desc and the ISR callback pointer. The ISR callback pointer associated with the GPU device driver is obtained, and a new interrupt interception ISR is registered in the Linux kernel. Finally, interrupts from the GPU through the ISR can be intercepted, and the callback pointer can be retained. In addition, I/O registers are mapped from the PCIe base address registers (BAR) to the kernel memory space by the device driver [35], [38]. NVIDIA GPUs typically depend on six to seven BARs. BAR0 is the main GPU control space, through which all hardware engines are controlled. Therefore, Linux-RTXG remaps BAR0 to the allocated space using $ioremap()$ when the ISR is initialized. The interrupt interception mechanism can identify the source of every interrupt by reading this remapped space.

3.4 Scheduler Integration

The mainline Linux scheduler implements the following real-time scheduling policies.

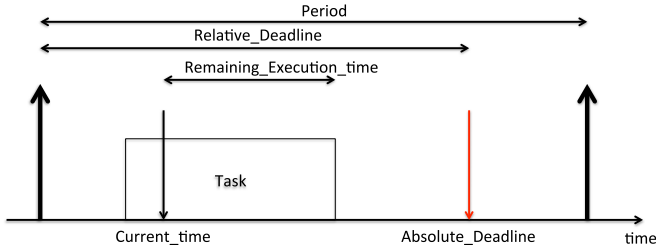


Fig. 5. Deadline relations.

- $SCHED\_DEADLINE$
- $SCHED\_FIFO$
- $SCHED\_RR$.

$SCHED\_DEADLINE$ is the implementation of CBS and EDF, which is the latest real-time scheduler for Linux introduced in version 3.14.0, and $SCHED\_FIFO$ and $SCHED\_RR$ represent fixed-priority scheduling. Unfortunately, synchronization does not work with the $SCHED\_DEADLINE$ scheduling policy for GPU tasks. Two problems concerning implementation must be solved. The first problem is attributed to the implementation of $sched\_yield()$. Note that $sched\_yield()$ uses $yield()$ in the kernel space. Releasing the CPU by $sched\_yield()$ while waiting for I/O in polling makes it possible to utilize CPU time more efficiently. However, $sched\_yield()$ will set the remaining execution time of the polling task to zero by treating it as a parameter of $SCHED\_DEADLINE$. As a result, the task cannot be executed until the runtime is replenished in the next period. This means that $sched\_yield()$ should not be called while polling in $SCHED\_DEADLINE$. However, $sched\_yield()$ is frequently used by device drivers and libraries. The real-time performance of GPU computing, even with CUDA, could be affected by this problem. This problem is addressed by limiting the GPU synchronization method to NOTIFY, i.e., eliminating the potential of FENCE, in the $SCHED\_DEADLINE$ scheduling policy.

The second problem is subject to the implementation of wake-up and sleep functions, particularly check Equation (1) when restoring a task from the sleep state. If Equation (1) holds, the runtime is replenished, and the absolute deadline is set to the next cycle deadline. $Absolute\_Deadline$ and $Relative\_Deadline$ are shown in Fig. 5. Note that $Absolute\_Deadline$ indicates the moment at which the response must be completed, and $Relative\_Deadline$ indicates the duration from the start of the task cycle to the $Absolute\_Deadline$

$$\frac{Absolute\_Deadline - Current\_Time}{Remaining\_Execution\_Time} > \frac{Relative\_Deadline}{Period}. \tag{1}$$

This check condition is revised so that the GPU execution time is subtracted from $Remaining\_Execution\_Time$ when a task is restored by the GPU kernel execution, except when a task is restored by the period.

4 Evaluation

The runtime overhead and real-time performance of Linux-RTXG was evaluated. The runtime overhead is classified into interrupt interception, independent synchronization,

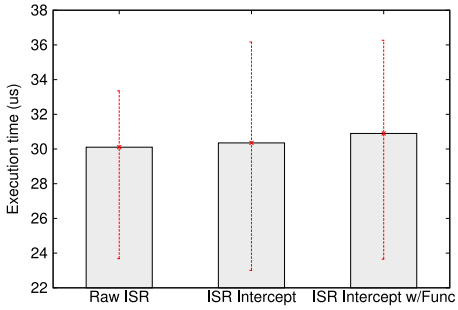


Fig. 6. Interrupt interception overhead.

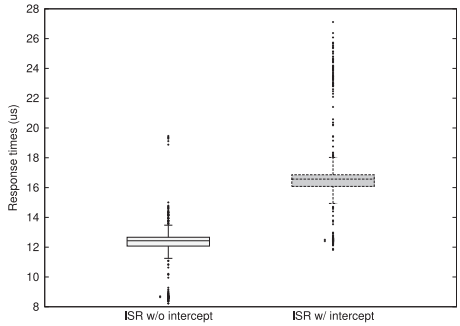


Fig. 7. Impact of interrupt interception.

and priority-driven scheduling in Linux. In the following, it is demonstrated that the overhead of the LKM-based real-time scheduler for GPU applications is acceptable. The real-time performance is verified in terms of QoS management and prioritization using both synthetic workloads and real-world applications on top of three device drivers, i.e., NVIDIA, Nouveau, and Gdev drivers. It is also demonstrated that multiple GPU applications co-scheduled by the LKM-based real-time scheduler are prioritized successfully and maintained at the desired frame rate even under high CPU load.

Given that CPU scheduling performance has been demonstrated in previous work [14], only GPU scheduling performance was evaluated experimentally in this work. Considering the results of this work and those of previous work, it is clarified that both emerging GPU applications and traditional CPU applications can be scheduled according to priorities and resource reserves using LKMs without modifying the Linux kernel and device drivers.

4.1 Experimental Setup

Experiments were conducted using the Linux kernel 3.16.0, an NVIDIA GeForce GTX680 GPU, a 3.40 GHz Intel Core i7 2600 (eight cores, including two hyper-threading cores), and 8 GB main memory. The GPU application programs were written in CUDA and compiled using NVCC v6.0.1. The NVIDIA 331.62 driver, the Nouveau Linux-3.16.0 driver with NVIDIA CUDA 6.0, and Gdev were used.

4.2 Interrupt Interception Overhead

The evaluations presented in Sections 4.2, 4.3, and 4.4 were conducted using the microbenchmark program provided by the Gdev project, which tests the GPU loop function. The overhead incurred by interrupt interception was measured using the Nouveau GPU driver to quantify the overhead of interception for varied interrupt types. The elapsed time from the beginning to the end of the ISR was adopted as a performance metric.

The overhead for interrupt interception is shown in Fig. 6. Here, "Raw ISR" represents the original ISR implementation. "ISR Intercept" represents the modified ISR with the interrupt interception mechanism, and "ISR Intercept w/Func" includes the overhead for identifying the ISR and calling the scheduler thread in addition to "ISR Intercept." The average execution time for 1,000 executions is presented with error bars.

Comparing Raw ISR and ISR Intercept indicates that the overhead for introducing the interrupt interception mechanism was 247 ns, which is only 0.8 percent of Raw ISR. Similarly, the overhead for introducing ISR Intercept w/Func was 790 ns, which is only 2.6 percent of Raw ISR. This indicates that the interrupt interception overhead is negligible relative to overall performance.

The response times of the ISR with and without interrupt interception are compared in Fig. 7. The response time is defined as the elapsed time from the beginning of interrupt processing to the point at which the interrupt type (e.g., timer, compute, FIFO, and GPIO) is identified. According to the results, interrupt interception makes response times 1.4 times longer.

Response times when interrupt interception is self-contained in the ISR (top-half) and when it is expanded to the tasklet (bottom-half) were also compared. This comparison differentiates Linux-RTXG from GPUSync in terms of performance because GPUSync intercepts the $tasklet\_schedule()$ on top of the proprietary driver, whereas Linux-RTXG realizes ISR-based interception. In this case, the start time of measuring the response time is set to the start time of the do_IRQ function call. The results of this comparison are shown in Fig. 8. As can be seen, the tasklet-based approach requires approximately five times longer response times than the IRQ-based approach because the tasklet is typically called after significant ISR processing steps.

4.3 Independent-Synchronization Overhead

The overhead of the independent-synchronization mechanism was also quantified. This mechanism must call $rtx\_gpu\_notify()$ at the time of the requested synchronization (e.g., after launching a GPU kernel). To use this mechanism, an initialization procedure is required. In the measurement of the initialization overhead, the overhead is defined by the execution time of corresponding APIs.

As shown in Fig. 9, the initialization overhead could reach 5,000 us, whereas the notification overhead is no more

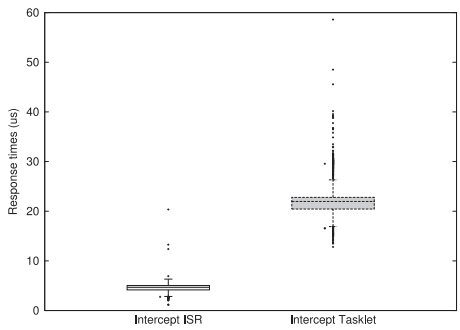


Fig. 8. Comparison of ISR and tasklet.

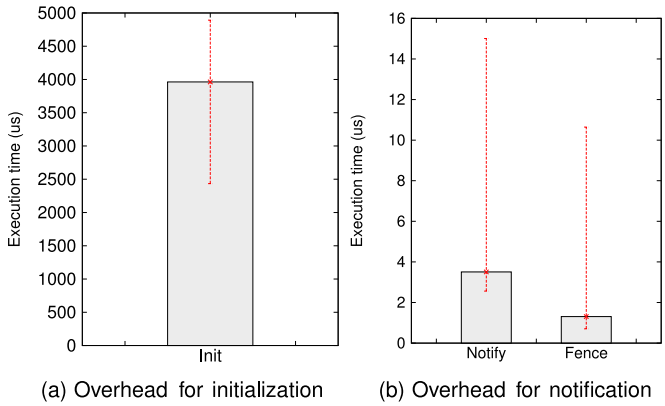


Fig. 9. Overhead for independent synchronization.

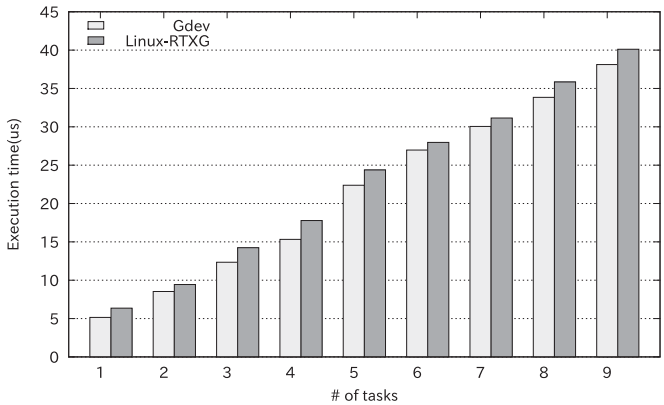


Fig. 11. Scheduling overhead.

than a few microseconds. The initialization procedure calls a Linux process to allocate indirect buffers and control registers for several GPU engines. Albeit significant overhead, the application program is not significantly affected by this procedure because it is called only once at the beginning. On the other hand, the notification overhead is not a major consideration for the application program; however, scattered notification overhead owing to an ioctl system call. When NOTIFY is used to set the notification, the overhead was 3.5 us, whereas it was reduced to 2 us when FENCE is used.

4.4 Scheduling Overhead

The scheduling overhead incurred by the proposed Linux-RTXG scheduler was evaluated. Four synthetic tasks, i.e., vanilla, gdev, mutex, and rtx, were executed to measure overhead. The program was modified the microbenchmark program provided by Gdev to generate multiple GPU tasks. Each task releases a job ten times periodically, each of which includes data transfer between the CPU and GPU, followed by the execution of the GPU kernel. Note that the rtx task was scheduled by Linux-RTXG, and the mutex task was limited to a single GPU kernel using an explicit mutual-exclusion control similar to that used for the rtx task. The gdev task was scheduled by Gdev. In this evaluation, to demonstrate the effect of scheduling, we based the benchmarking on Gdev using the runtime library in the OS to make the effect of memory copying comparable to our approach. The vanilla task was taken directly from the original microbenchmark program. As shown in Fig. 10, the vanilla task represents the execution time for GPU tasks without the use of a scheduler. The CPU scheduling policy was set to *SCHED_FIFO*, and the GPU scheduling policy was based on fixed priorities with GPU resource reservation. The resource reservation was performed automatically without configuration by the user. The independent-synchronization mechanism was employed with NOTIFY.

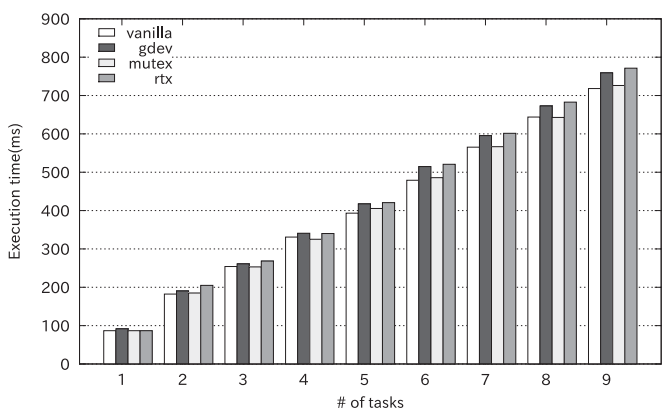


Fig. 10. Impact of scheduling.

We measured the average time of 100 GPU task executions (1,000 jobs). The impact of scheduling is shown in Fig. 10. The execution time increases proportionally to the number of tasks owing to the increased time caused by explicit mutual-exclusion control and scheduling the GPU kernel.

The pure scheduling overhead obtained by comparing the Gdev and Linux-RTXG schedulers is shown in Fig. 11. The scheduling overhead also increases because the time consumed by queueing GPU tasks increases. The maximum overhead corresponds to 11.7 percent of Gdev for the four tasks.

4.5 Prioritization and QoS Performance

Experiments were also performed to evaluate the prioritization and QoS performance for GPU applications provided by Linux-RTXG. The performance of Linux-RTXG was evaluated with both the proprietary NVIDIA driver and the open-source Nouveau driver compared to the built-in kernel approach (i.e., Gdev).

QoS management indicates if the GPU time of the corresponding task is guaranteed. QoS performance was evaluated by observing how well the tasks are isolated. First, GPU utilization was measured when running two GPU tasks. Each GPU task has a unique workload with a different GPU reserve size. One task was allocated to VGPU0 and given 40 percent GPU reserve. This task (with high priority) has 1.2 times the workload of the other task, which was allocated to VGPU1 with 60 percent GPU reserve. The VGPU1 task was scheduled to start approximately 5 s after the VGPU0 task.

The results obtained under the FIFO scheduling policy with the NVIDIA driver are shown in Fig. 12a, and those under the BAND scheduling policy are shown in Fig. 12b. The corresponding results obtained with the Nouveau driver are shown in Figs. 12c and 12d.

Since VGPU0 has a higher workload than VGPU1, the GPU tasks were performed according to their workload, as shown in Figs. 12a and 12c. On the other hand, the GPU resource reservation mechanism provided by Linux-RTXG successfully performed the GPU tasks according to their

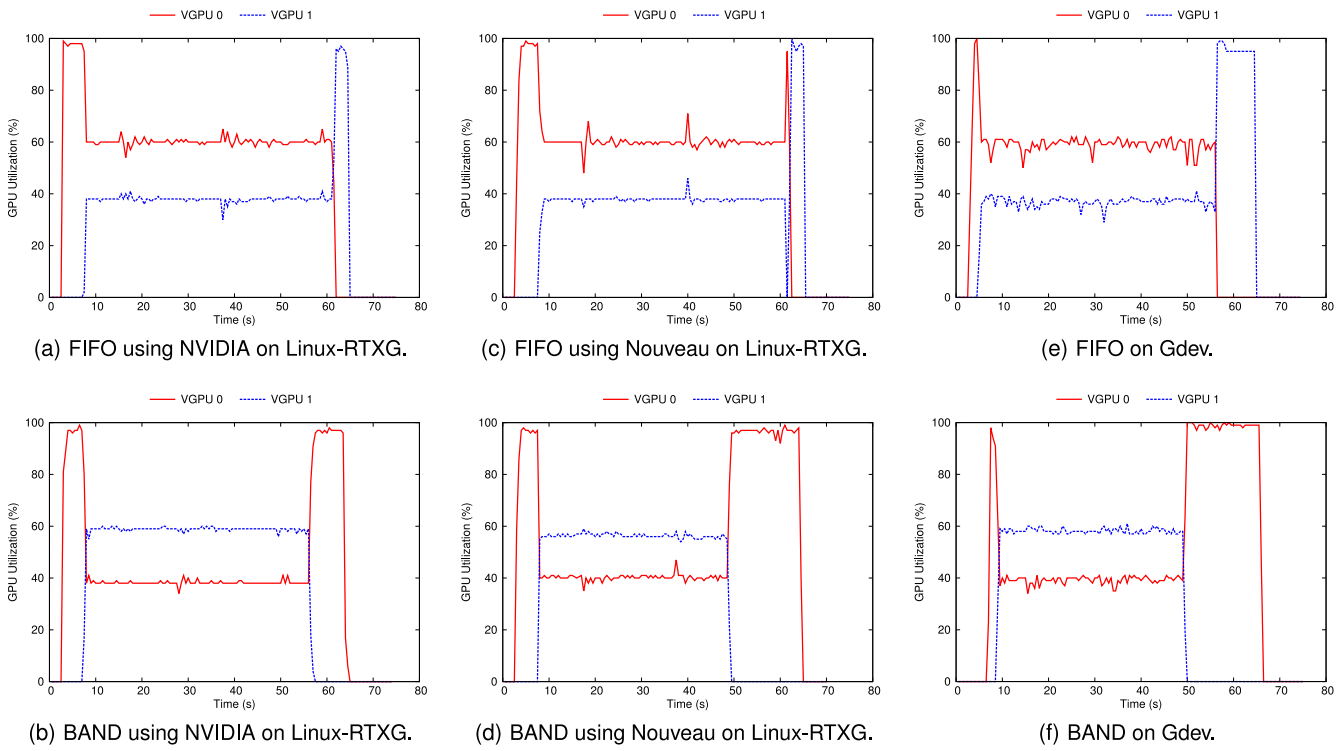


Fig. 12. GPU utilization in two tasks. Each task executes different types of GPU-intensive workloads with specified GPU reserves (VGPU0 = 40 percent; VGPU1 = 60 percent).

reserves, as shown in Figs. 12b and 12d. It is important to note that these prioritization and resource reservations for GPU applications can be achieved with Linux-RTXG without modifying the OS kernel and device drivers.

The maximum error of the VGPU1 task was approximately 3 percent under the BAND scheduling policy with the NVIDIA driver, and that of the VGPU0 task was approximately 5 percent. With the Nouveau driver, these values were 2 and 6 percent, respectively. These large increases occurred owing to GPU kernel overruns.

In addition to the performance of the proposed loadable kernel approach, performance was compared with that of previous work, i.e., Gdev. The Gdev scheduling results are shown in Figs. 12e and 12f. Compared to Gdev, no performance loss is imposed by Linux-RTXG. When the Gdev scheduler was used, the maximum error of the VGPU1 task was approximately 3 percent under the BAND scheduling policy, and that of the VGPU0 task was 5 percent. There was also a large variation on a time basis when the Gdev scheduler was used because the Gdev runtime functions must be called in the kernel space, whereas other system calls can easily block their operation.

GPU utilization running four GPU tasks was also measured. Each GPU task has the same workload and the same GPU reserve on a different VGPU. The isolation results for this scenario are shown in Fig. 13. The maximum error of VGPUs was 9 percent, which was incurred owing to the timing of budget replenishment and synchronization latency. In fact, this result nearly matches that reported in previous work with Gdev [10] using the built-in kernel approach. Consequently, the independent-synchronization mechanism employed in Linux-RTXG does not sacrifice scheduling performance when using existing drivers.

4.6 Real-World Application

Here, the performance of Linux-RTXG with a real-world application is demonstrated. The demonstration was performed with a GPU-accelerated object detection program [2] based on the well-known DPM and HOG algorithms. It was assumed that the monitoring system covered the four directions of the compass (east, west, south, and north).

The execution time per frame was measured in six different setups, as shown in Fig. 14, where (a) no scheduling was applied, (b) fixed priorities were applied with GPU scheduling, (c) GPU resource reservation was further applied with the BAND scheduling policy, and (d), (e), and (f) high CPU

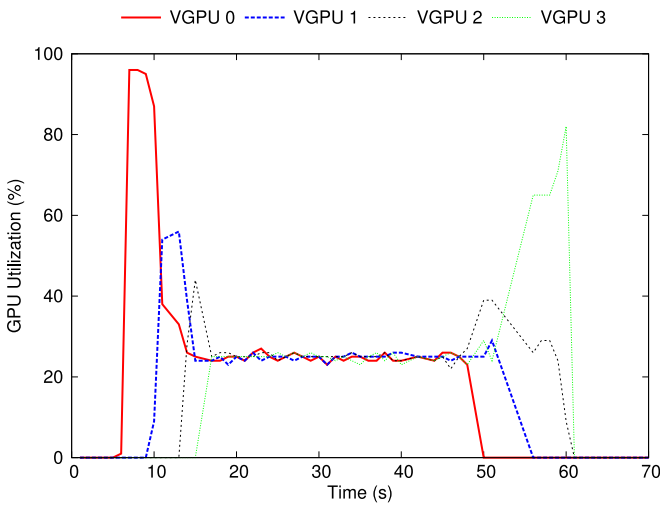


Fig. 13. Utilization of four tasks with the Linux-RTXG's BAND VGPU scheduling. Each task had an equal workload and an equal GPU resource allocation.

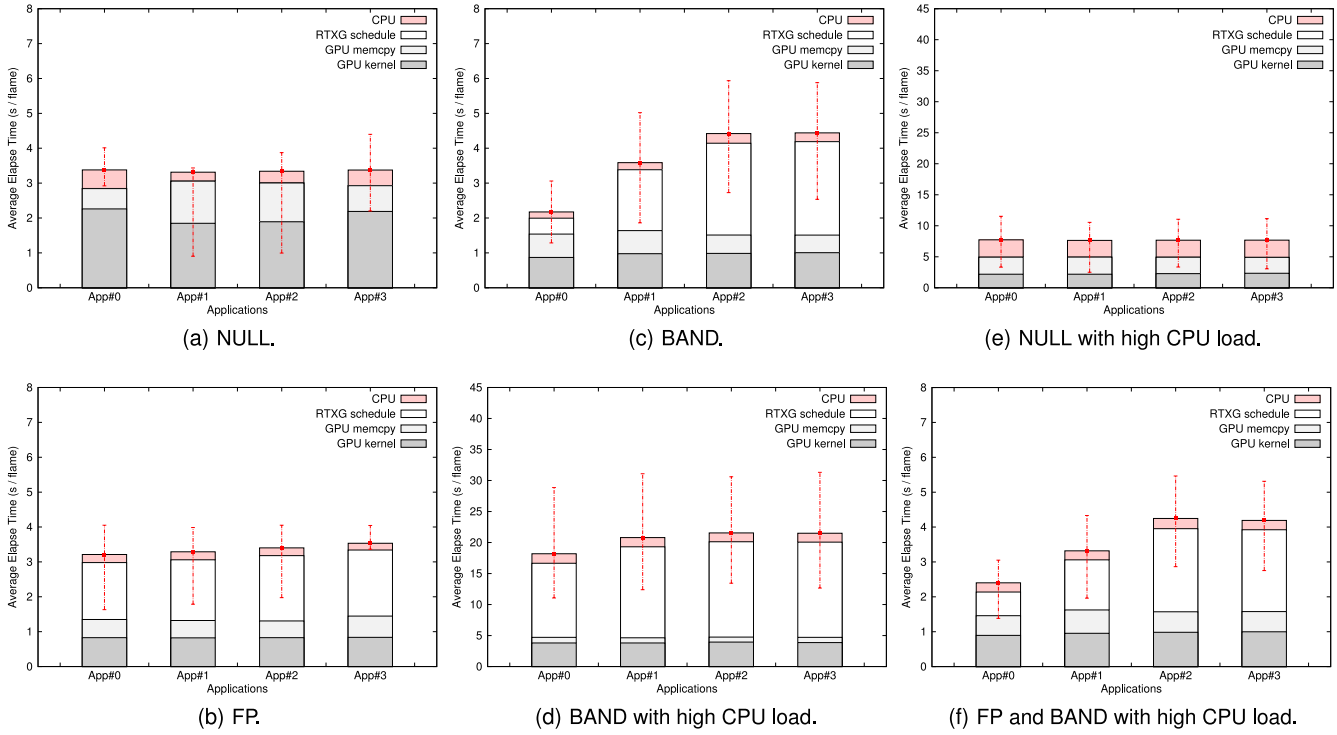


Fig. 14. Execution time of the GPU-accelerated object detection program.

load was contending with the *SCHED_OTHER* hackbench task. The GPU tasks were allocated 60, 20, 10, and 10 percent GPU reserves, respectively.

The results of this experiment indicate that if a high-CPU load task exists, the performance of GPU tasks could be seriously affected because GPU tasks cannot acquire the CPU time required to execute the GPU API. Linux-RTXG can provide both CPU scheduling and GPU scheduling with resource reservation capabilities to solve this problem, as shown in Fig. 14f. As can be seen, the target programs are not affected by the high-CPU-load task due to the *SCHED_FIFO* scheduling, and the GPU execution can be maintained at the desired frame rate owing to the GPU-scheduling and resource reservation capabilities. These results demonstrate that Linux-RTXG can supply prioritization and QoS performance to real-world applications.

4.7 Basic Performance

The Rodinia benchmark suite [39] can be used to realistically measure the performance of a wide range of heterogeneous computing devices, such as GPUs and multicore CPUs using CUDA and OpenMP. Heterogeneous computing using GPUs suffers from performance bottlenecks at numerous locations. The overhead when copying data from the host I/O to main memory is one example. The Rodinia benchmark can be used to identify such bottlenecks and evaluate possible solutions. Fig. 15 compares the standalone performance achieved by Linux-RTXG, NVIDIA's proprietary drivers, and Gdev. To clarify the location of bottlenecks, we compared two Gdev approaches, i.e., Gdev/OS, which integrates runtime support into the OS, and Gdev/User, which employs a runtime library in the user space. Table 2 shows the microbenchmarks and Rodinia benchmarks used in this evaluation. Note that the CPU and GPU scheduling policy was the same as that presented in Section 4.4.

In our previous work [10], we found that some "performance mode" provided by NVIDIA GPUs boost

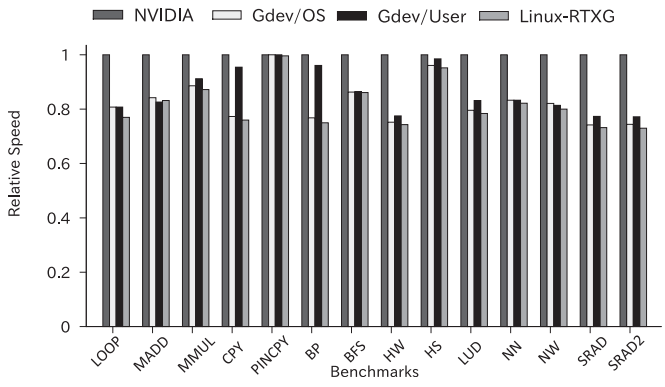


Fig. 15. Basic standalone performance.

TABLE 2
Benchmarks

Benchmark	Description
LOOP	Long-loop compute without data
MADD	1024 x 1024 matrix addition
MMUL	1024 x 1024 matrix multiplication
CPY	256 MB of HtoD and DtoH
PINCPY	CPY using pinned host I/O memory
BP	Back propagation (pattern recognition)
BFS	Breadth-first search (graph algorithm)
HW	Heart wall (medical imaging)
HS	Hotspot (physics simulation)
LUD	LU decomposition (linear algebra)
NN	K-nearest neighbors (data mining)
NW	Needleman-Wunsch (bioinformatics)
SRAD	Speckle reducing anisotropic diffusion (imaging)
SRAD2	SRAD with random pseudo-inputs (imaging)

TABLE 3
Linux-RTXG Compared to Previous Work

	CPU & GPU Scheduling	Multiple GPUs	Memory Allocation	Budget Enforcement	Data/Computing Overlap	Closed Source Compatible	Patchless	GPU Runtime independent
RGEM			x			x	x	
Gdev		x	x	x	x			
PTask		x		x	x	x		x
NEON					x	x	x	x
GPUSync	x	x	x	x	x	x		x
GPU-SPARC	x	x			x	x	x	
Linux-RTXG	x			x	x	x	x	x

hardware performance. The distinctiveness of NVIDIA's proprietary software lies in its special "performance mode." When meeting real-time constraints on the GPU, Linux-RTXG and Gdev demonstrated diminished performance compared to the proprietary software. In addition, Linux-RTXG was also inferior to Gdev/OS owing to the interrupt interception overhead. As observed in the BP benchmark results, the difference between our approach and Gdev/User demonstrates the effect of memcpy in the host memory because the BP benchmark is associated with very large datasets, thereby making the memcpy function of the Linux kernel a bottleneck. The results of the benchmark tests demonstrate that the performance of our approach is comparable to existing kernel-dependent approaches and that the bottleneck arises in memcpy's performance within host memory.

5 Related Work

RGEM [28] and GPU-SPARC [30] provide real-time GPU resource management without OS kernel and device driver modifications. However, their synchronization mechanism depends on proprietary software. TimeGraph [9], Gdev [10], PTask [40], and GPUSync [11] realize independent synchronization mechanisms but require modifying the OS kernel and device drivers. To the best of our knowledge, Linux-RTXG is the only solution that provides real-time GPU resource management with a synchronization mechanism without modifying the OS kernel and device drivers.

Table 3 compares Linux-RTXG and previous work. Here, CPU and GPU scheduling indicates whether the CPU and GPU schedulers are coordinated. GPUSync supports fixed-priority and EDF scheduling policies for CPU tasks, and GPU-SPARC employs the *SCHED_FIFO* scheduling policy. Multiple GPUs indicates whether the GPU scheduler considers optimization, such as migration cost for multiple GPUs. GPUSync, GPU-SPARC, PTask, and Gdev also focus on using multiple GPUs in a real-time system. When multiple GPUs are present, latency due to data migration becomes the primary performance determinant. Thus, previous work has considered the use of multiple GPUs and data locality. Linux-RTXG allows multiple GPUs but does not support such optimization. Memory allocation indicates whether the GPU scheduler considers memory operations, such as bus scheduling and memory bandwidth. RGEM, GPUSync, and Gdev tackle a problem by which memory transfers affect a real-time system using a GPGPU. Note that Data/Computing Overlap indicates whether data transfer and computing are performed simultaneously, and Closed Source Compatible indicates whether the GPU management system can use the NVIDIA CUDA. As shown in Table 3, Linux-RTXG includes important features that previous methods have contributed. In particular, the resource management modules of Linux-RTXG are all loadable and are freed from the detailed implementation of runtime libraries, device drivers, and the OS kernel.

More in-depth resource management would require detailed information about the execution mechanisms in the black-box GPU stack. Menychtas et al. presented interception-based OS-level GPU scheduling, called NEON, without modifying the software stack by using state machine inferencing [41], [42]. GPU resource management using GPU microcontrollers [26] and in-kernel runtime functions [10] has also been demonstrated to manage a GPU. Note that the Nouveau project was used as a baseline driver [36] for such open-source work.

There is also increasing interest in applying existing technology to GPUs. Kernelet [43] demonstrates preemptive scheduling by dividing the GPU kernel into sub-kernels. Chimera [44] employs two GPU-specific preemption techniques to achieve low throughput overhead preemption and preemption latency. However, in a real-time system, it is necessary to consider the coordinated scheduling of CPU tasks because these occupy more CPU resources when not processing by increasing the GPU kernels that have not completed processing. As the use of GPUs in systems with strict real-time constraints increases, preemptive scheduling is becoming necessary, and hardware support by NVIDIA is being investigated. If hardware support for preemptive scheduling is developed, our approach should improve the efficiency of the entire system by allowing the CPU scheduler to allocate tasks while the GPU kernel executes using the independent-synchronization mechanism. Pai et al. [45] proposed an effective technique for timeslicing kernel execution for a CUDA stream by running the GPU kernels concurrently in a pipeline manner. Although this work was not designed for use with real-time systems, the application of a CUDA stream can increase throughput, reduce turnaround times, and improve resource utilization.

6 Conclusion

In this paper, Linux-RTXG, which is a Linux extension with the LKM-based real-time GPU resource management module, has been proposed and evaluated. Linux-RTXG can manage GPU resources without modifying the OS kernel and device drivers. In addition, the proposed Linux-RTXG has been compared to CPU resource management proposed in previous work [14], [15]. A new scheme for GPU resource

management was also developed. By intercepting interrupts from the GPU in the top-half ISRs, the developed synchronization mechanism can wait for completion of the execution of specific GPU kernels without modifying the OS kernel and device drivers. An experimental evaluation of Linux-RTXG indicates that the chosen performance unit (i.e., task overhead) varied within approximately 10 percent for eight tasks and within approximately 4 percent for four tasks. Furthermore, the superior prioritization and QoS performance of Linux-RTXG were demonstrated with a real-world object detection application, where GPU programs can be protected from high CPU load while the desired frame rate on the GPU is maintained successfully. To the best of our knowledge, this is the first complete Linux extension that can manage GPU resources using the LKM.

REFERENCES

[1] M. McNaughton, C. Urmson, J. Dolan, and J.-W. Lee, "Motion planning for autonomous driving with a conformal spatiotemporal lattice," in *Proc. IEEE Int. Robot. Autom.*, 2011, pp. 4889–4895.

[2] M. Hirabayashi, S. Kato, M. Edahiro, K. Takeda, T. Kawano, and S. Mita, "GPU implementations of object detection using HOG features and deformable models," in *Proc. Int. Conf. CPSNA*, 2013, pp. 106–111.

[3] N. Rath, S. Kato, J. Levesque, M. Mauel, G. Navratil, and Q. Peng, "Fast, multi-channel real-time processing of signals with microsecond latency using graphics processing units," *Rev. Sci. Instruments*, vol. 85, no. 4, 2014, Art. no. 045114.

[4] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar, "Resource sharing in GPU-accelerated windowing systems," in *Proc. RTAS. IEEE*, 2011, pp. 191–200.

[5] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proc. Workshop GPGPU*, 2010, pp. 94–103.

[6] S. Che, et al., "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.

[7] CUDA Zone, [Online]. Available: https://developer.nvidia.com/category/zone/cuda-zone, Accessed no: Jan. 18, 2016.

[8] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Comput. Sci. Eng.*, vol. 12, no. 3, pp. 66–73, 2010.

[9] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *Proc. USENIC ATC*, 2011, p. 17.

[10] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt, "Gdev: Firstclass GPU resource management in the operating system," in *Proc. USENIX ATC*, 2012, pp. 401–412.

[11] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A framework for real-time GPU management," in *Proc. RTSS. IEEE*, 2013, pp. 33–44.

[12] G. A. Elliott and J. H. Anderson, "Exploring the multitude of real-time multi-GPU configurations," in *Proc. RTSS. IEEE*, 2014.

[13] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "*LITMUS*^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proc. 27th IEEE Int. Real-Time Syst. Symp.*, 2006, pp. 111–126.

[14] S. Kato, R. Rajkumar, and Y. Ishikawa, "A loadable real-time scheduler suite for multicore platforms," Univ. Tokushima, Tokushima, Japan, Tech. Rep. CMU-ECE-TR09–12, 2009.

[15] M. Asberg, T. Nolte, S. Kato, R. Rajkumar, and Y. Ishikawa, "Exsched: An external CPU scheduler framework for real-time systems," IEEE RTCSA, Washington, DC, USA, Tech. Rep. CMU-ECE-TR09–12, pp. 240–249, 2012.

[16] J. A. Stankovic and K. Ramamritham, "The spring kernel: A new paradigm for real-time systems," *IEEE Softw.*, vol. 8, no. 3, pp. 62–72, May 1991.

[17] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Redline: First class support for interactivity in commodity operating systems," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, vol. 8, pp. 73–86.

[18] H. Takada and K. Sakamura, "μITRON for small-scale embedded systems," *IEEE Micro*, vol. 15, no. 6, pp. 46–54, 1995.

[19] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time and multimedia systems," in *Proc. SPIE*, 1997, pp. 150–164.

[20] S. Oikawa and R. Rajkumar, "Portable RK: A portable resource kernel for guaranteed and enforced timing behavior," in *Proc. 5th IEEE Real-Time Technol. Appl.*, 1999, pp. 111–120.

[21] P. Mantegazza, E. Dozio, and S. Papacharalambous, "RTAI: Real time application interface," *Linux J.*, vol. 2000, no. 72es, 2000, Art. no. 10.

[22] V. Yodaiken, "The RTLinux manifesto," in *Proc. 5th Linux Expo*, pp. 187–197, 1999.

[23] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[24] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proc. IEEE Real-Time Syst. Symp.*, 1998, pp. 4–13.

[25] M. Spuri and G. Buttazzo, "Scheduling aperiodic tasks in dynamic priority systems," *Real-Time Syst.*, vol. 10, no. 2, pp. 179–210, 1996.

[26] Y. Fujii, T. Azumi, N. Nishio, and S. Kato, "Exploring microcontrollers in GPUs," in *Proc. APSys. ACM*, 2013, p. 6.

[27] G. A. Elliott and J. H. Anderson, "Robust real-time multiprocessor interrupt handling motivated by GPUs," in *Proc. ECRTS*, 2012, pp. 267–276.

[28] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "RGEM: A responsive GPGPU execution model for runtime engines," in *Proc. RTSS. IEEE*, 2011, pp. 57–66.

[29] C. Basaran and K.-D Kang, "Supporting preemptive task executions and memory copies in GPGPUs," in *Proc. ECRTS*, 2012, pp. 287–296.

[30] W. Han, H. Bae, H. Kim, J. Lee, and I. Shin, "GPU-SPARC: Accelerating parallelism in multi-GPU real-time systems," Tech. Rep., 2014.

[31] F. Ridouard, P. Richard, and F. Cottet, "Negative results for scheduling independent hard real-time tasks with self-suspensions," in *Proc. IEEE Real-Time Syst. Symp.*, 2004, pp. 47–56.

[32] K. Lakshmanan, S. Kato, and R. R. Rajkumar, "Open problems in scheduling self-suspending tasks," in *Proc. Real-Time Scheduling Open Problems Seminar*, pp. 12–13, 2010.

[33] B. Chattopadhyay and S. Baruah, "Limited-preemption scheduling on multiprocessors," in *Proc. Int. Conf. RTNS*, 2014, Art. no. 225.

[34] J. Kim, B. Andersson, D. D. Niz, and R. R. Rajkumar, "Segment-fixed priority scheduling for self-suspending real-time tasks," in *Proc. IEEE Real-Time Syst. Symp.*, 2013, pp. 246–257.

[35] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data transfer matters for GPU computing," in *Proc. ICPADS. IEEE*, 2013, pp. 275–282.

[36] Nouveau, [Online]. Available: http://nouveau.freedesktop.org/wiki/, Accessed on: Jan. 18, 2016.

[37] P. Barham, et al., "Xen and the art of virtualization," in *Proc. 9th ACM Symp. Operating Syst. Principles*, 2003, pp. 164–177.

[38] S. Kato, J. Aumiller, and S. Brandt, "Zero-copy I/O processing for low-latency GPU computing," in *Proc. ACM/IEEE 4th Int. Conf. Cyber-Physical Syst.*, 2013, pp. 170–178.

[39] S. Che, et al., "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. Int. Symp. Workload Characterization*, 2009, pp. 44–54.

[40] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating system abstractions to manage GPUs as compute devices," in *Proc. SOSP. ACM*, 2011, pp. 233–248.

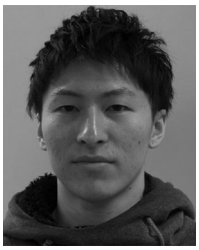
[41] K. Menychtas, K. Shen, and M. L. Scott, "Enabling OS research by inferring interactions in the black-box GPU stack," in *Proc. USENIX Conf. Annu. Techn. Conf.*, 2013, pp. 291–296.

[42] K. Menychtas, K. Shen, and M. L. Scott, "Disengaged scheduling for fair, protected access to fast computational accelerators," in *Proc. 19th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2014, pp. 301–316.

[43] J. Zhong and B. He, "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1522–1532, Jun. 2014.

[44] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," in *Proc. Archit. Support Program. Languages Operating Syst.*, 2015, pp. 593–606.

[45] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," in *Proc. Archit. Support Program. Languages Operating Syst.*, 2013, pp. 407–418.



Yuhei Suzuki received the BE degree from the Ritsumeikan University, in 2016. He is working toward the master's degree in the Graduate School of Information Science and Engineering, Ritsumeikan University. His research interests include parallel computing, task scheduling, and autonomous vehicles.

Yusuke Fujii received the BE and ME degree in information science and engineering from the Ritsumeikan University, Japan, in 2015. Currently, he works with NTT Software Innovation Center as a researcher and an engineer. His current research interests including high-performance heterogeneous distributed computing, cloud computing, operating systems, real-time systems.

Takuya Azumi received the PhD degree from the Graduate School of Information Science, Nagoya University, in 2009. He is an assistant professor in the Graduate School of Engineering Science, Osaka University. He was a visiting researcher with the University of California, Irvine, in 2011. His research interests include real-time systems, component based development, and cyber-physical systems.

Nobuhiko Nishio received the BE and ME degrees in the Department of Mathematical Engineering and Information Physics, University of Tokyo, Tokyo, Japan, in 1986 and 1988, respectively, and the PhD degree from Keio University, in 2000. From 1993 till 2003, he worked with Keio University SFC. From 2007 till 2008, he worked as a visiting scientist with the Google Inc. Currently he is a professor of the College of Information Science and Engineering, Ritsumeikan University, Shiga, Japan. His current research interests include ubiquitous computing, location based systems, long term human activity recognition, real-time and embedded computing.

Shinpei Kato received the BS, MS, and PhD degrees from Keio University, in 2004, 2006, and 2008, respectively. He is an associate professor in the School of Information Science and Technology, the University of Tokyo. He has also worked with Carnegie Mellon University, University of California, Santa Cruz, and Nagoya University, from 2009 to 2015. His research interests include operating systems, real-time systems, and parallel and distributed systems.