Coordinated CPU and GPU Resource Management with Loadable Kernel Modules

Yusuke Fujii*, Takuya Azumi*, Nobuhiko Nishio*, and Shinpei Kato†

* College of Information Science and Engineering, Ritsumeikan University
† Graduate School of Information Science, Nagoya University

***Abstract*—Graphics processing units (GPUs) are well-designed for high-performance computing. Programming frameworks are also getting matured for general-purpose computing on GPUs (GPGPU). Originally, runtime environments of GPU computing are tailored to accelerate particular best-effort applications. In recent years, however, GPU computing has been applied for real-time applications, improving resource management mechanisms of the operating system (OS) kernel. Unfortunately, such a system extension is often a complex undertaking for research and development, and the modifications to kernel source code often narrow down consistency of OS kernel and device driver. In this paper, we present Linux-RTXG, which is a loadable kernel module approach to coordinated CPU and GPU resource management. Linux-RTXG does not require the OS kernel to be modified at the source code level, instead adding resource management mechanisms by loadable kernel modules with user-space library support. To achieve coordinated CPU and GPU resource management in Linux-RTXG, such as real-time scheduling and resource reservation, we design and implement novel interrupt intercept mechanisms and independent interrupt mechanisms. Experimental results demonstrate that the presented system achieves real-time scheduling and resource reservation capabilities for GPU-accelerated systems, maintaining the system overhead comparable to existing kernel-dependent approaches.**

***Keywords*—GPU, resource management, scheduling, real-time system, operating system**

I. Introduction

Graphics processing units (GPUs) are becoming mature computing platforms for various data-parallel and compute-intensive applications. Although the primary use of GPUs is to accelerate high-performance computing (HPC) applications, the performance advantage of GPUs is more and more recognized in real-time applications. Examples 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 are also well prepared for other various pieces of workload [6].

GPU applications were often best-effort oriented, whose main purpose is to accelerate particular computing blocks. Conventional system software for GPU computing, therefore, is particularly designed for individual data-parallel and compute-intensive workload. However, due to emergence of cyber-physical systems or streaming applications in recent years, system software support for real-time and multi-tasking environments is becoming a more significant challenge for GPU computing. Since the standard system software provided by GPU vendors and communities, such as CUDA [7] and OpenCL [8], is not tailored to support real-time multi-tasking environments, a complex undertaking is functional extension of system software including the operating system (OS), device drivers, and runtime libraries is required even in state-of-the-art research and development to use GPUs in such environments.

In previous work, GPU resource management has been presented in terms of real-time and/or multi-tasking environments. TimeGraph [9] provides GPU scheduling and resource reservation mechanisms at the device driver level to multiplex GPU commands, supporting real-time multi-tasking GPU applications. Gdev [10] is a rich set of runtime libraries and device drivers to multiplex GPU computations and CPU-GPU data transfers, achieving first-class GPU resource management. The main drawback of these approaches is that they require non-trivial detailed information on internal implementation of GPU runtime libraries and device drivers, mostly obtained based on reverse engineering.

Traditionally, GPU software stack is encapsulated by an application programming interfance (API) in user-space libraries and an application binary interface (ABI) in device drivers. This means that modifications to system software on top of the API and ABI layers allow the entire software stack to be loadable, resulting in more flexible solutions. CPU resource management also needs to be considered given that CPU time is still consumed while GPU kernels are issued and executed via the API and ABI layers. In order to satisfy real-time requirements in GPU computing, therefore, resource management mechanisms for both the CPU and GPU must be coordinated. For example, the data transfer overhead for GPU computing is highly dominated by the coordination of CPU and GPU resource management [11].

As one of notable work on coordinated CPU and GPU resource management for real-time multi-tasking environments, GPUSync was presented by Elliott et al. [12], [13] to extend CPU scheduling for multiple GPU-aware contexts with budget enforcement, which was built on top of on the proprietary API and ABI layers. GPUSync provides a configurable framework that can verify the combination of task allocation policies for multicore CPUs as well as GPU kernel allocaton policies for multiple GPUs.

GPUSync is implemented using $LITMUS^{RT}$ [14], which introduces a significant amount of changes to the OS kernel. TimeGraph and Gdev also make some modifications to the device driver. Those customized approaches to the OS or device drivers require users to install patches, or developers are obliged to maintain patches in order to stay up to date with the latest OS kernel releases. This porting work indeed kills productivity of research and development, given that especially open-source software, such as Linux, is frequently updated with signficant code changes.

Linux supports the loadable kernel module (LKM), which can load and unload kernel modules to add and remove kernel functions. Many Linux-based real-time OSes use the LKM to extend real-time capabilities of Linux. In particular, RESCH [15], [16] provides a real-time scheduling framework called ExSched, which does not require any modification to the OS kernel and device drivers. Unfortunately, RESCH does not support GPU resource management. It has never been demonstrated that GPU resource management and its coordination with CPU resource management for real-time systems can be fully implemented using the LKM, without making modifications to the OS kernel and device drivers.

Contribution: This paper presents Linux-RTXG (Linux Real-Time eXtention with GPUs), which provides LKM-based real-time extension for coordinated CPU and GPU resource management in Linux. Linux-RTXG allows the system to easily re-configure scheduling algorithms and install their modules at runtime. The most significant contribution of Linux-RTXG is that resource management modules for not only CPUs but also GPUs can be added to Linux without any modification to the OS kernel and device drivers. CPU scheduling and resource reservation mechanisms are based on RESCH, while GPU scheduling and resource reservation mechanisms are implemented using Gdev. In addition to the integration of RESCH and Gdev, Linux-RTXG provides a new framework to coordinate CPU and GPU resource management, freeing from the built-in OS kernel and device drivers, thus is a competely patch-free approach.

Organization The rest of this paper is organized as follows. Section II describes the system model and basic approaches for Linux-RTXG. Section III presents the design and implementation of Linux-RTXG with a particular emphasis on GPU scheduling. Section IV evaluates the system overhead and reservation performance of Linux-RTXG. Section V discusses related work, and this paper is concluded in Section VI.

II. System Model

This section describes the model of GPU programming and GPU scheduling assumed in this paper. We also introduce some existing work to highlight a unsolved problem of GPU resource management. In addition to the problem of GPU resource management, we argue the obstacle that has prevented GPU resource management from patch-free implementation. Note that this work focuses on such systems that are composed of multiple GPUs and multi-core CPUs.

A. GPU Programming Model

General-purpose computing on GPUs (GPGPU) is supported by special programming languages, such as CUDA and OpenCL. This work assumes that GPU application programs are written in CUDA; however, the concept of GPU resource management studied in this paper is not limited by programming languages, i.e., the contribution of this work is applidable to OpenCL. We define a GPU task as a process running on the CPU that executes a function to launch a GPU kernel to the GPU, whose cyclic execution unit is referred to as a GPU job. The GPU kernel is a process that is executed on the GPU. Note that we also assume multi-tasking environments where multiple tasks of GPU applications are allowed to coexisit, though the current model of GPU programming does not allow multiple contexts to execute at the same time on the GPU but kernels from the same context. In other words, we can create and launch multiple GPU contexts, but they must be executed exclusively on the GPU by hardware nature.

GPU programming requires a set of APIs provided by runtime libraries such as CUDA Driver API or CUDA Runtime API. Typically, GPU application programs are executed through the following steps: (i) *cuCtxCreate* creates a GPU context; (ii) *cuMemAlloc* allocates memory space to the device memory; (iii) *cuModuleLoad* and *cuMemcpyHtoD* copy the data and the GPU kernel from the host memory to the allocated device memory space; (iv) *cuLaunchGrid* invokes the GPU kernel; (v) *cuCtxSynchronize* synchronizes a GPU task to wait for the completion of 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.

B. GPU Scheduling

Resource management problems have been addressed in many types of real-time OSes (RTOSes) [17], [18], [19], [20]. In particular, those in Linux-based RTOSes have received considerable attention [14], [21], [22], [23], [15]. GPUs are also supported in Linux. This work assumes that the design concept and function of OS are based on Linux, such as LKM.

To meet real-time requirements in shared resource environments, such as multi-core CPUs and GPUs, RTOSes should provide *scheduling* and *resource reservation* functions. Rate Monotonic (RM) and Earliest Deadline First (EDF) [24]) are well-known algorithms of priority-based scheduling for real-time systems. There are also many variants for resource reservation algorithms. Examples include Constant Bandwidth Server (CBS) [25] and Total Bandwidth Server (TBS) [26].

GPU computing must deal with data transfer bandwidth and compute cores as a shared resource. Therefore, scheduling and resource reservation must also be considered for GPUs as well as CPUs in real-time systems. TimeGraph and Gdev are involved with GPU resource management problems but are not much aware of the fact that GPU tasks consume CPU time to drive APIs. To support GPUs in real-time systems, the OS scheduler must be able to manage GPU tasks together with CPU tasks on the host side, while taking care of GPU time for the device side. Therefore, we consider the problem of CPU scheduling and GPU scheduling with resource reservation mechanisms in a coordinated manner.

The recently developed GPUSync system supports CPU scheduling and GPU scheduling. In GPUSync, the device driver and runtime library for GPU computing are proprietary black-box modules released by GPU vendors. On the other hand, TimeGraph and Gdev address this problem by using reverse-engineered open-source software. Both approaches are limited to some extent, respectively. Using black-box modules makes it difficult to manage the system in a fine-grained manner. Open-source software tends to lack some functionality due to incompletion of reverse engineering. GPUSync managed to incorporate black-box modules in scheduling and resource reservation by arbitrating interrupt handlers and runtime accesses. This GPUSync approach is preferred in a sense that we can utilize reliable proprietary driver and library, while we can still provide scheduling and resource reservation functions.

GPU Synchronization: Given that the GPU is a coprocessor connected to the host CPU, synchronization between the GPU and the CPU must be considered to guarantee the correctness of execution logics. Most GPU architectures support two synchronization mechanisms. One is based on memory-mapped registers called FENCE. To use FENCE, we send special commands 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 is polling to monitor this value via the mapped space. The other technique is interrupt-based synchronization called NOTIFY. To use NOTIFY, we also send special commands to the GPU similar to FENCE. Instead of writing to memory-mapped registers, NOTIFY raises the interrupt from the GPU to the CPU and at the same time writes some associated values to I/O registers of the GPU. On the host side, a GPU task is suspended to wait for the occurrence of interrupt. NOTIFY allows other tasks to use CPU resources when the GPU task is waiting for completion of its GPU kernel, though the scheduling overhead is involved. FENCE is easier to use and is more responsive than NOTIFY but is implemented at the expense of CPU utilization. More details about GPU architectures can be found in previous work [9], [10], [27].

Gdev supports both NOTIFY and FENCE to synchronize the CPU and the GPU. NOTIFY is primarily used for scheduling of GPU tasks, while FENCE is used in driver-level synchronization. In Gdev, the implementation of GPU synchronization involves additional commands to the GPU as aforementioned, which requires some modification to the device driver. GPUSync indirectly utilizes the NOTIFY technique with tasklet interception [28] on top of the proprietary black-box modules. Tasklet refers to Linux's soft-irq implementation. GPUSync identifies the interrupt that has invoked a GPU kernel using a callback pointer with a tasklet.

Loadable Kernel Modules: The main concept of our system is to enable both the CPU and GPU to be managed by the OS scheduler without any code changes to the OS kernel and device drivers. CPU scheduling has already been demonstrated by RESCH [15], [16] under this constraint. In order to schedule GPU tasks, we must be able to hook the scheduling points where the preceding GPU kernel is completed and the active context is switched to the next GPU kernel. The scheduling points can be hooked by two methods. The API-driven method presented by RGEM [29] explicitly awakens the scheduler after GPU synchronization invoked by the API, such as $cuCtxSynchronize()$. The interrupt-driven method presented by TimeGraph and Gdev, on the other hand, uses interrupts that can be configured by NOTIFY. GPUSync is also based on this interrupt-driven method. Especially in 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 one GPU kernel. In other words, if a GPU task invokes multiple GPU kernels, the interrupt-driven method is more approapriate to realize real-time capabilities.

The interrupt-driven method forced TimeGraph, Gdev, and GPUSync to modify the code of either the Linux kernel or device drivers. This is because the interrupt service routine (ISR) must be managed to create the scheduling points. Gdev has developed independent synchronization mechanisms on top of the proprietary software; however, Gdev still needs some modification to the Linux kernel for scheduling and resource reservation. As a result, the available release versions of the Linux kernel and device drivers for TimeGraph, Gdev, and GPUSync are very limited. A core challenge of this paper is to develop independent synchronization mechanisms that do not require any modification to the OS kernel and device drivers so that we can utilize the coordinated CPU and GPU resource management scheme with a wide range of the Linux kernel and device drivers.

Scope and Limitation: GPU resource management is involved with non-preemptive nature. For example, the execution of GPU kernels is non-preemptive. The data transfer between the CPU and the GPU is also non-preemptive. Some previous work have addressed the problem of response time by preventing overrun that occurs while dividing the kernel [30], [31]. The most difficult problem is the scheduling of self-suspending tasks because the GPU is treated as an I/O device in the system. For example, GPU tasks are suspending until their GPU kernels are completed. The self-suspending problem was presented as a NP-HARD problem in previous work [32], [33]. There are a lot of ongoing work on the scheduling of self-suspending tasks [34], [35].

Such a schedulability problem of GPU scheduling is not the scope of this paper. We rather focus on the design and implementation of GPU scheduling and resource reservation with existing algorithms. In this paper, we also restrict our attention to time resources but not memory resources. Our prototype system is limited to the Linux system and the CUDA environment, but the concept of our method is applicable to other OSes and programming languages, as far as they support the aforementioned NOTIFY mechanism.

III. Design and Implementation

In this section, we present the design and implementation of Linux-RTXG, which provides a framework of coordinated CPU and GPU resource management with LKMs. We describe our approach to GPU scheduling and its integration to CPU scheduling. Due to a space constraint, the detail of LKM-based CPU scheduling is referred to the RESCH project [15], [16].

A. Linux-RTXG

Figure 1 shows an architectural overview of Linux-RTXG. The system architecture of Linux-RTXG falls 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 using the input/output control (ioctl) system call, which is a standard

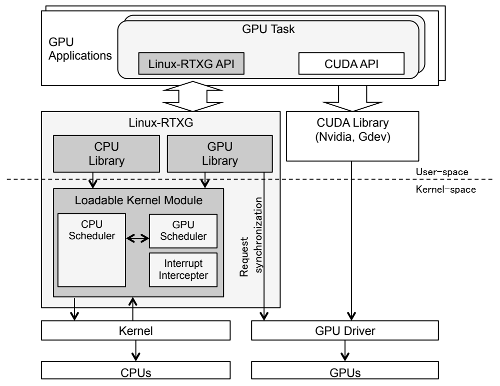


Fig. 1. Architectural overview of Linux-RTXG.

```
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 the Linux-RTXG APIs.

system call for device drivers. Secondly, the Linux-RTXG library contains an independent synchronization method for coordinated CPU and GPU resource management. The independent synchronization method can be used on top of a proprietary driver [7] as well as an open-source driver [36]. Note that this method is required to manage interrupts for GPU scheduling without any code modification to the OS kernel and device drivers.

B. GPU Scheduling

Linux-RTXG is based on the interrupt-driven method for GPU synchronization but is also 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 I. Note that some APIs have arguments whereas others do not. Linux-RTXG does not modify the existing CUDA API to cope with proprietary software, being independent of GPU runtimes. However, CUDA application programs must add the Linux-RTXG APIs to use the functionality of Linux-RTXG.

The sample code including the Linux-RTXG APIs is shown in Figure 2. GPU tasks are provided with function calls to Linux-RTXG at strategic points.

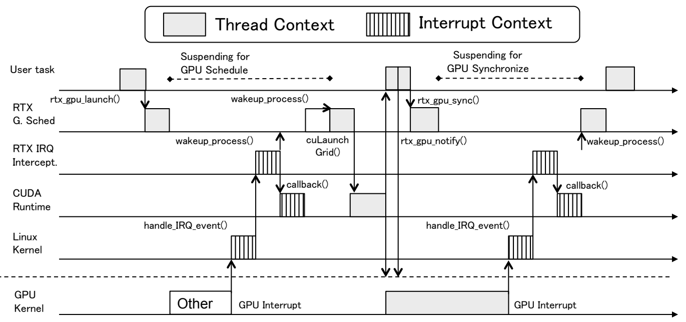


Fig. 3. Execution flow of the GPU task.

The execution flow of the GPU task managed by the Linux-RTXG APIs is described in Figure 3. Note that this example is restricted to a single GPU kernel. The GPU task can control the timing of GPU kernel invocation by calling $rtx\_gpu\_launch()$. After this function call, the task is suspended until it receives an interrupt so that other GPU kernels can be launched. When some GPU kernel is completed, an interrupt is raised by the GPU and the corresponding interrupt handler is executed by the Linux kernel. The interrupt interceptor awakens some suspending task according to the priority. The awakened task proceeds to launch the GPU kernel using the CUDA API, such as $cuLaunchGrid()$. After the GPU kernel is launched, the task is going to register NOTIFY to set up an interrupt, and is again put to the sleep mode until it receives the interrupt. Dispatching of the subsequent task is performed by the GPU scheduler, which is called upon the interrupt from the GPU. Linux-RTXG manages the order of task execution according to this flow.

We now present a hierarchal scheduling mechanism that uses the concept of virtual GPUs to combine specified GPU tasks by a group. The virtual GPUs are realized by a resource reservation mechanism, while GPU scheduling uses a priority mechanism. Specifically, each GPU kernel invocation is associated with a scheduling entity, and Linux-RTXG allocates the scheduling entities to virtual GPUs. The virtual GPUs can belong to any physical GPUs. In Linux-RTXG, computing resources are distributed to virtual GPUs.

Figure 4 shows the pseudo-code of the Linux-RTXG scheduler. This code works under the assumption that $on\_arrival$ is called when some GPU task requests to launch the GPU kernel. In $on\_arrival$, the GPU task checks whether the given execution permission is held by the allocated virtual GPU and is also held by the allocated scheduling entity. If the virtual GPU to which the GPU task belongs does not own the execution permission, the GPU task is enqueued to $wait\_queue$ and is suspended. Else, the GPU task can launch the GPU kernel. After a while, $on\_completion$ is called by the scheduler thread when the launched GPU kernel is completed, and we can select the next set of a virtual GPU and a GPU task. At the end of $on\_completion$, the selected GPU task is waken up.

C. GPU Synchronization

We next describe the independent synchronization mechanism and the interrupt interception approach. The independent

TABLE I. A BASIC SET OF APIS FOR LINUX-RTXG.

rtx_gpu_open()	Registers itself to Linux-RTXG and creates 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 timing, (i.e., a scheduling entry point). It must be called before the CUDA launch API.
rtx_gpu_sync()	Waits for the completion of GPU kernel execution by sleeping with TASK UNINTERRUPTIBLE status.
rtx_gpu_notify()	Sends NOTIFY/FENCE command to GPU. The FENCE/NOTIFY are selected by flag that is set by argument.
rtx_gpu_close()	Releases scheduling entity.

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

```
void on_arrival(se) {
 check_permit_vgpu(se->vgpu)
 while(!check_permit_se(se)){
   enqueue(se->vgpu,se);
   sleep_task(se->task);
 }
}
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);
   wakeup_task(se->task);
 }
 set_the_permit(se->vgpu, se)
}
```

Fig. 4. High-level pseudo-code of the Linux-RTXG scheduler.

synchronization mechanism invokes NOTIFY and FENCE without using the GPU runtime API. The interrupt interception enables interrupt-driven invocation of the scheduler without making any modification to the OS kernel and device drivers. By this means, Linux-RTXG does not require to modify the OS kernel and device drivers, while being able to create the scheduling points for GPU tasks.

Independent Synchronization Mechanism: We now present the independent synchronization mechanism using NOTIFY and FENCE. 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 the user space. These ioctl interfaces provide driver functions, such as device memory allocation, obtaining GPU states and memory mapping. Gdev contains a runtime library that can control the GPU on top of NVIDIA's proprietary driver using these ioctl interfaces. Our mechanism also uses an ioctl interface similar to Gdev in order to send commands to the GPU. Specifically, our mechanism is divided into two parts, (i) initialization and (ii) notification.

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

The notification process sends commands to a GPU compute engine or a GPU data-copy engine by 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 such GPU microcontrollers that control the states of GPU computation and data transfer. They are also used to switch GPU contexts on the GPU computation and data transfer. Note that this independent synchronization mechanism requires the information of ioctl interfaces. Therefore, it depends on the GPU architecture and implementation of device drivers.

Interrupt Interception: Interrupts are handled by the ISR that is registered to the Linux kernel by the device driver. The scheduler function is required to 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.

The Linux kernel has a structure that holds interrupt parameters called irq_desc for each interrupt number. This structure has an internal structure called irq_action, including the ISR callback pointer. The irq_desc structure is allocated to the global kernel memory space, and is freely accessible from the kernel space. Therefore, not only the Linux kernel but also external LKMs can obtain the information of irq_desc and the ISR callback pointer. We obtain the ISR callback pointer associated with the GPU device driver, and register a new interrupt interception ISR to the Linux kernel. Finally, we can intercept interrupts from the GPU through the ISR and retain the callback pointer. In addition, I/O registers are mapped to the kernel memory space by the device driver from the PCIe base address registers (BAR) [11], [37]. Therefore, Linux-RTXG remaps the BAR0 to our 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.

D. Scheduler Integration

The mainline Linux scheduler implements a few real-time scheduling policies:

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

$SCHED\_DEADLINE$ is the implementation of CBS and EDF, which is the latest real-time scheduler for Linux introduced in the version 3.14.0, while $SCHED\_FIFO$ and $SCHED\_RR$ represent fixed-priority scheduling. Unfortunately, synchronization does not work with the

SCHED_DEADLINE scheduling policy for GPU tasks. Let us describe two problems. 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, we can 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 execute until the runtime is replenished in the next period. This means that $sched\_yield()$ should not be called while polling in $SCHED\_DEADLINE$. However, $schedyield()$ is frequently used by device drivers and libraries. Real-time performance of GPU computing, even in CUDA, could be affected by this problem. We address this problem by limiting the GPU synchronization method to NOTIFY, 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 the check equation 1 when restoring a task from the sleep state. If the equation 1 holds, the runtime is replenished and the absolute deadline is set to the next cycle deadline.

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

We revise this check condition so that the GPU execution time is substracted from $RemainingRuntime$ when a task is restored by the GPU kernel execution, except when a task is restored by the period.

IV. Evaluation

Here, we evaluate scheduling overhead and scheduling performance. Scheduling experiments are limited to GPU scheduling because CPU scheduling performance has already been examined experimentallys [15]. In this evaluation, we focus on two points, identifying Linux-RTXG's disadvantages and demonstrating the quality of service (QoS) performance. We compare the functions and features of the proposed method and previous methods in a qualitative evaluation in the next section.

A. Evaluation Environment

Our experiments were conducted using Linux kernel 3.16.0, an NVIDIA Geforce GTX680 GPU, a 3.40 GHz Intel Core i7 2600 (8 cores, including two hyperthreadling cores), and 8 GB main memory. GPU programs were written in CUDA and compiled by NVCC v6.0.1. We used the NVIDIA 331.62 driver and Nouveau Linux-3.16.0 driver. In addition, we used NVIDIA CUDA-6.0 libraries and Gdev.

B. Interrupt Intercept Overhead

We measured the interrupt interception overhead using the Nouveau GPU driver to compare and identify the type of interrupt. We compared elapsed time from the beginning to the end of the ISR.

Figure 5 shows the results of measurements for the above setting. Here, Raw ISR is the ISR executed in the original routine. Note that ISR Intercept is the only intercept considered in our approach. ISR Intercept w/Func refers to interception with processing functions that identify the ISR and wake the scheduler thread. The results shown are average times (1000 executions); error bars indicate minimum and maximum values.

From the results, it is evident that overhead exists. Overhead for ISR Intercept was 247 ns, which is 0.8% of the total time required for the Raw ISR process. Overhead for ISR Intercept w/Func was 790 ns, which is 2.6% of the total time required for the Raw ISR process. Intuitively, these overheads unlikely to affect the system because are very small values; however, interrupts (e.g., timer interrupts) occur frequently, and cumulatively are likely to be significant.

Next, we compare response times to determine the impact of the interrupt intercept. The response times are elapsed times from the start of interrupt processing to the end of identify the each interrupt type (e.g., timer, compute, FIFO, and GPIO). The response times for ISRs with and without intercepts are shown in Figure 6. Response times with intercepts were approximately 1.4 times longer than response times for interrupts without intercepts. However, we target systems that do not use the GPU runtime resource management features. Therefore, we should aim to fast response time of intercepted ISR, and original ISR response time is slow there is not affected much.

We also evaluated response time of the ISR (top-half) and the tasklet (bottom-half) in an environment to compare our results with those of previous approaches. GPUSync intercepts the $tasklet\_schedule()$ if it is called by the NVIDIA driver. We compare the response times of a tasklet intercept using GPUSync and the ISR intercept. This evaluation measured response time until the responsible timing from the start of interrupt process which is do_IRQ function is called. Figure 7 shows the result of this evaluation. As can be seen, the tasklet interception response time is worse than the ISR time. This occurs because the tasklet is typically called after significant ISR processing.

C. Independent Synchronization Mechanism Overhead

We evaluated the overhead relative to the use of an independent synchronization mechanism. Such a mechanism must call $rtx\_nvrm\_notify()$ at the time of requested synchronization (e.g., after the kernel launch request) and $rtx\_nvrm\_init()$. In vanilla environments, Linux-RTX's APIs are not necessary. Therefore, overhead includes the API execution time. We measured overhead by measuring API execution time between each API call and return.

Figure 8 shows the measurement results. Initilize part is required to call a Linux process to allocate an indirect buffer and registers several engines such as compute and copy to the device driver. Notification ($rtx\_gpu\_notify()$) insert commands into a command steram to GPU devices that are called when synchronization is required, such as after a kernel launch is issued. Execution time have scatters that are affected by an ioctl system call. The average Initizalize time is approximately 4 millseconds. However, applications are not affected significantly because Initialize is only called once. Notification's NOTIFY requires approximately 3.5 microseconds and FENCE requires approximately 2 microseconds.

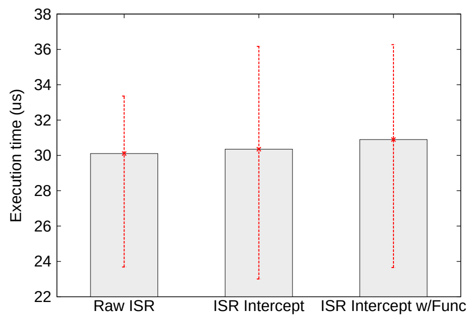


Fig. 5. Interrupt intercept overhead

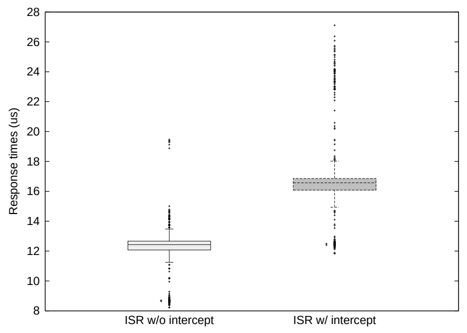


Fig. 6. Response time of interrupt w/o and w/ intercept

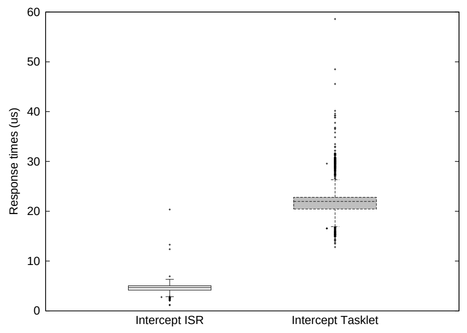
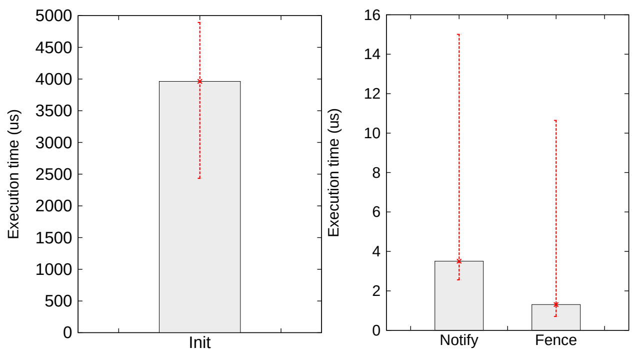


Fig. 7. Response time of the top-half intercept and bottom-half intercept



(a) Part of Initialize (b) Part of notify

Fig. 8. Interrupt raised method overhead

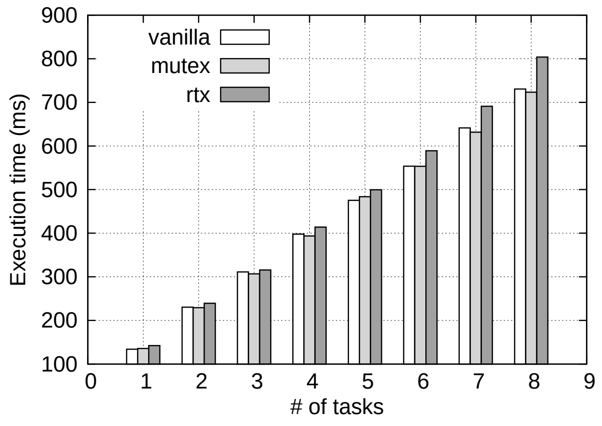


Fig. 9. Scheduling overhead (Time of entire task)

Consequently, the time required by these method is not a major consideration for applications. However, overhead must be considered in a shot application cycle.

D. Scheduling Overhead

We evaluated scheduling overhead using the proposed Linux-RTXG scheduler. We prepared three applications, i.e., vanilla, mutex, and rtx to measure overhead. These applications are based on a common application that is Gdev's microbenchmark, which has a GPU looping function. Changes were arranged to generate multiple GPU tasks by the $fork()$ systemcall. Each task has 10 jobs, and each job includes GPU data transfer and GPU kernel execution. The rtx application was scheduled by rtx. The mutex application was limited to a single kernel issue by exclusion control using mutex similar to rtx. The vanilla application was not changed. The CPU scheduling policy used in this evaluation was the simple fixed-priority scheduling of the proposed Linux-RTXG, which is similar to Linux's $SCHED\_FIFO$. Note that the difference between these policies is the presence or absence of job management. The GPU scheduling policy is the fixed-priority scheduling with resource reservation, i.e., BAND scheduling policy. Here, synchronization is used NOTIFY of the independent synchronization mechanism.

We measured the average time in 100 times GPU task execution (1000 jobs). The result shows in Figure 9. Overhead increases in proportion as the number of tasks increases because task scheduling processing times is increased. Max overhead is 10% at the eight tasks based on ”vanilla” time.

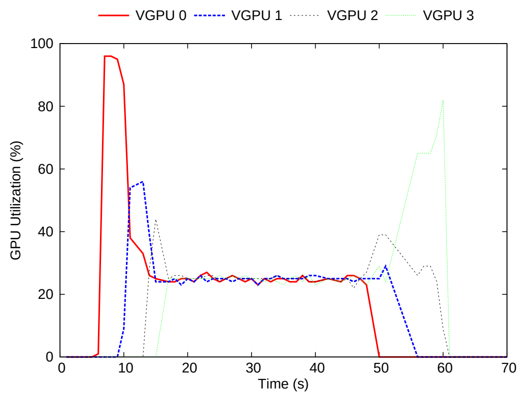
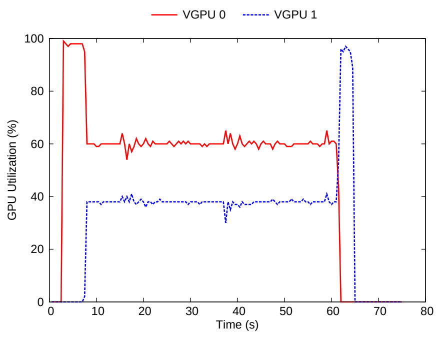


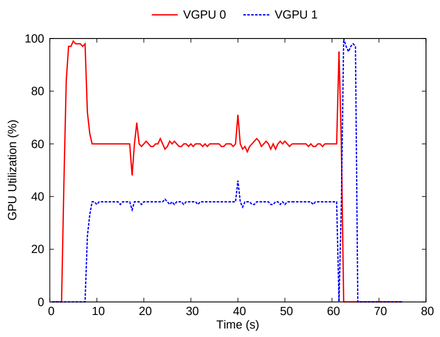
Fig. 11. Utilization of four tasks with the Linux-RTXG's BAND VGPU scheduling. Each tasks had a equal workload and a equal GPU resource allocation.

E. Performance of QoS management

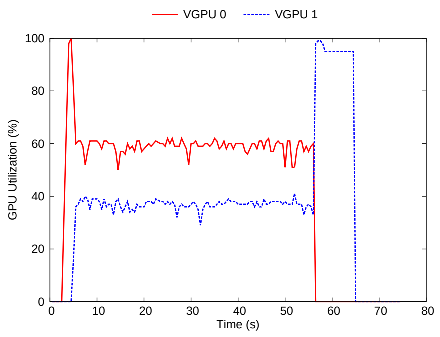
Experiments were also performed to evaluate QoS management performance. In this evaluation, we measured the utilization of each tasks in several environments, which indicates whether performance is affected by kernel modification.



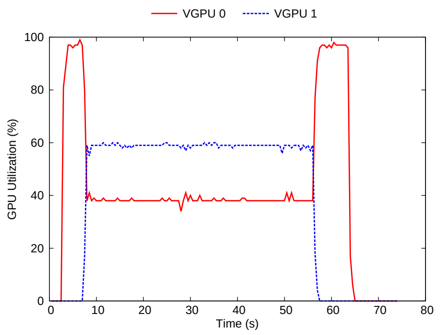
(a) FIFO with NVIDIA



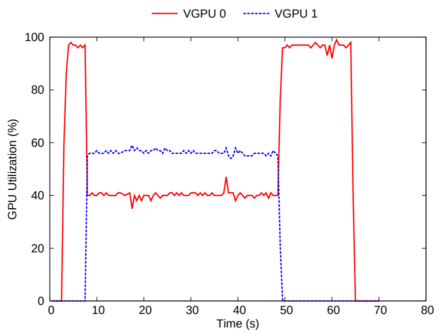
(c) FIFO with Nouveau



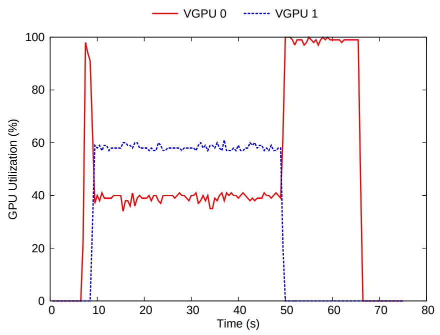
(e) FIFO on Gdev



(b) BAND with NVIDIA



(d) BAND with Nouveau



(f) BAND on Gdev

Fig. 10. Utilization of two tasks with each scheduler. Each task had different workloads and different resource allocations (VGPU0 = 40%, VGPU1 = 60%).

QoS performance use an indication of whether the task of the resource is guaranteed. We evaluate task isolation performances with Linux-RTXG's GPU scheduler using the NVIDIA GPU driver and the Nouveau GPU driver to confirm the guaranteed resources. First, we measured the utilization when running two GPU tasks. Each GPU tasks had a different workload and a different GPU resource. One GPU task was allocated VGPU0, and given the 40% GPU resource. This task has about 1.2 times the workload of other tasks. The other GPU task was allocated VGPU1 and given the 60% GPU resource. VGPU1 task was scheduled to start approximately 5 safter the VGPU0 task.

Figure 10(a) shows result of FIFO scheduling policies on the NVIDIA Driver, and the BAND scheduling policies is shown in Figure 10(b).

The Nouveau GPU driver results are shown in Figures 10(c) and (d). The FIFO scheduling policies shown in Figure 10(c), and the BAND scheduling policies shown in Figure 10(d).

The results shown in Figure 10(a) and (c) indicate that the GPU tasks are performed in accordance with the workload by fair scheduling. The results shown in Figure 10(b) and 10(d) indicate that the GPU tasks are performed in accordance with the utilization by resource reservation mechanisms.

The VGPU1 task's maximum error was approximately 3% for the initial BAND scheduling policies' resource management using the NVIDIA GPU driver. The VGPU0 task's maximum error was approximately 5%. The VGPU1 task's maximum error was approximately 2% under the initial BAND scheduling policies' resource management using the Nouveau GPU driver. The VGPU0 task's maximum error was approximately 2%. Large spikes occurred due to GPU kernel overrun. If the GPU scheduler need to large spike is reduced, the GPU scheduler is needed for runtime approaches such as making preemptive GPU kernel.

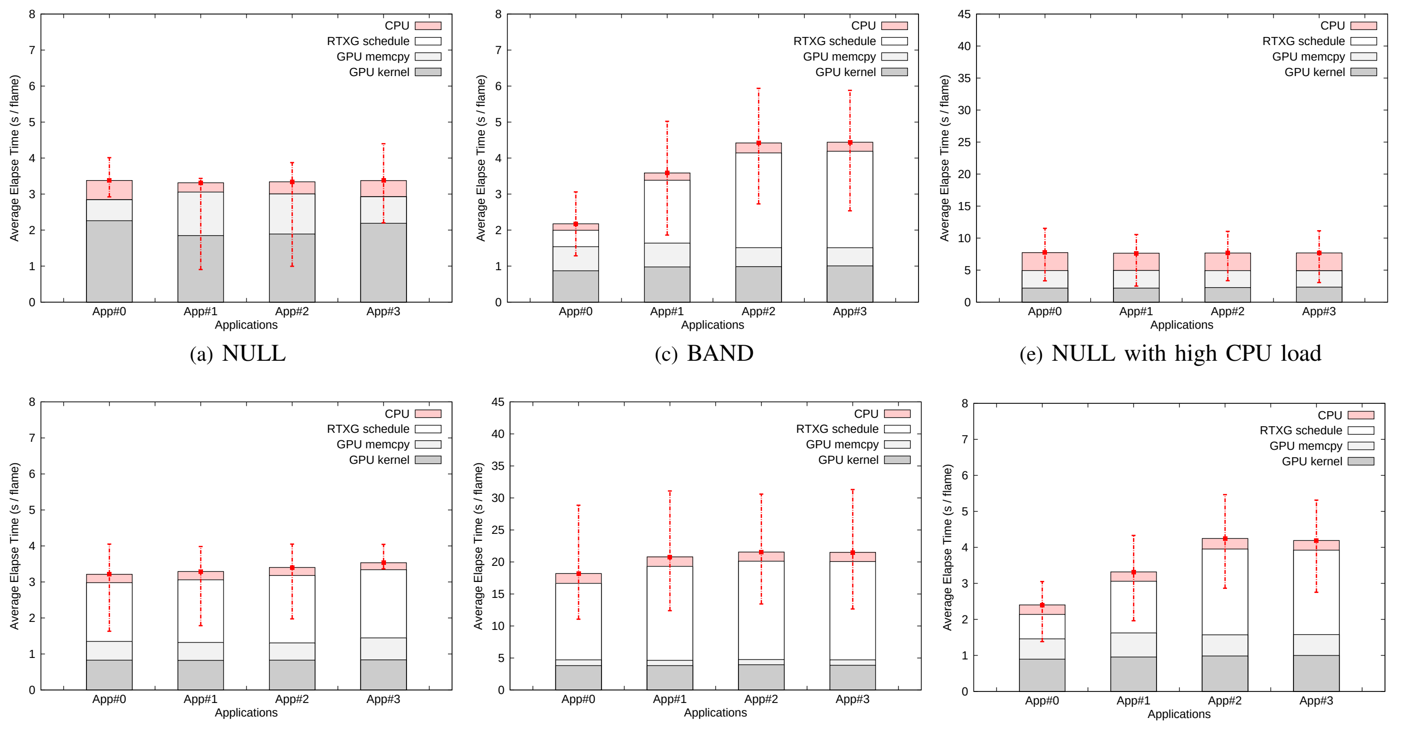
In addition, we compare performance with prior work that is Gdev. The Gdev scheduling results are shown in Figures 10(e) and (f). Linux-RTXG is seen almost no performance degradation as compared with Gdev. In details, the VGPU1 task's maximum error was approximately 3% for the initial BAND scheduling policies' resource management using the Gdev scheduler. The VGPU0 task's maximum error was approximately 5%. We guess that there is a large variation caused by runtime specification of the Gdev function must through the kernel module using Gdev scheduler.

We then measured utilization running four GPU tasks. Each GPU tasks had equal workload and equal GPU resource allocation. In addition, each GPU task was allocated to each VGPU. Results for BAND scheduling policies are shown in Figure 11. A maximum error of these VGPUs was appropriately 9%, it values are occurred because of the timing of budget replenishment and a synchronization latency.

Therefore, the results show that the synchronization mechanism of the proposed Linux-RTXG can schedule tasks without sacrificing performance.

F. Real-world Application

Here, we demonstrate scheduling performance using real-world oriented application. We employed real-world oriented application which is object detection application using the



(a) NULL (c) BAND (e) NULL with high CPU load

(b) FP (d) BAND with high CPU load (f) G-RM and BAND with high CPU load

Fig. 12. Elapse times of object detection applications on various situation.

HOG features. We run four tasks are assumed to recognize the four directions that are forward, backward, right, and left.

Algorithms and implementation of the applications are described in the previous work [2].

We measure the elapse time of processing one frame image in six situations. Each situation results are shown in Figure 12. Each GPU tasks are allocated difference resources are given 60%, 20%, 10%, and 10% respectively. Figure 12(a) is shown result of executing four tasks without scheduling. All tasks are performed equally on its situation. Figure 12(b) is shown result on executing four task runnning with a fixed-priority GPU kernel scheduling (FP). If a scheduler only uses fixed-priority GPU kernel scheduling, tasks are almost the same as the Figure 12(a). Figure 12(c) is addition of the BAND VGPU scheduling from the Figure 12(b) situation. In this case, the tasks have been adapted the priority of each tasks for which can suppress the GPU kernel execution. Figures 12(d) and (e) are shown result of the high CPU load environment. Note that high CPU load is generated by hackbench tasks as known as an UNIX's standard stress test tool. High CPU load tasks policy is Linux's $SCHED\_OTHER$. If an OS has some high-CPU load tasks, GPU tasks do not work well using only GPU scheduling because GPU tasks can not acquire the CPU to issue an API due to other tasks. Such a high-CPU load environment to occur well in autonomouse driving car systems because the systems require a lot of applications (e.g., SLAM, Navigation, and Car control) despite limit the resources. Linux-RTXG can provide real-time CPU scheduling and GPU scheduling. Therefore, Linux-RTXG can solves problem due to other CPU tasks. The result show in Figure 12(f). As can be seen, applications are not affected high-CPU load task by fixed-priority CPU scheduling (G-RM:Glocbal Rate-monotonic). These experimental results indicate that Linux-RTX making the adequate performance for real-world oriented applications.

V. Discussion

Here, we discuss the comparing between proposed Linux-RTXG framework and previous work.

RGEM and GPU-Sparc [31] have GPU resource management without kernel and device driver modification. However, the synchronization mechanisms of these methods depends on proprietary-software. TimeGraph, Gdev, Ptask, and GPUSync realize independent synchronization mechanisms that do modify the kernel and device drivers. To the best of our knowledge, the proposed Linux-RTXG is the only real-time GPU framework that uses a synchronization mechanisms that are independent of runtime and do not modifiy the kernel and device drivers.

Table II shows a comparison of the proposed Linux-RTXG and previous methods. GPU sync supports the fixed-priority and EDF CPU scheduling policies. GPUSparc employs the $SCHED\_FIFO$ Linux scheduling policies. Note that Linux-RTXG demonstrates all features shown in Table II. In particular, Linux-RTXG is both GPU runtime independendt and OS independent.

More in-depth resource management would require detailed information about the execution mechanisms in black–box GPU stacks. Menychtas et al. presented enabling to GPU using OS reserach by inferring interactions in a black–box GPU stack [38]. We have presented information about

TABLE II. LINUX-RTXG VS PRIOR WORK

	CPU FP	CPU EDF	GPU Prio. Sched.	Budget Enforcement	Data/Comp. Ovlp.	Closed Src. Compatible	Kernel Free	OS independent	GPU Runtime independent
RGEM			x			x	x	x	
Gdev			x	x	x				
PTask			x	x	x	x			x
GPUSync	x	x	x	x	x	x			x
GPUSparc	▲		x		x	x		x	
Linux-RTXG	x	x	x	x	x	x	x	x	x

GPU microcontrollers [27] and an open-source GPGPU runtime [10]. In addition, GPUSync verifies proprietary runtime chanism infomation. The Nouveau project provide an open-source GPU driver [36].

The more-depth resource management would require the detail executing mechanisms in the black-box GPU stack. Menychtas et al. present enabling GPU using OS research by inferring interaction in the black-box GPU stack [38]. We present information of GPU microcontrollers [27] and an open-source GPGPU runtime [10]. GPUSync presents details verification information on the proprietary runtime mechanisms. Nouveau project provides an open-source GPU driver [36].

VI. CONCLUSION

This paper has presented Linux-RTXG, a Linux-based real-time extension for CPU/GPU resource coordination. We have focused on a system that accomplishes GPU resource management without modifying the kernel. Linux-RTXG include CPU task scheduler, GPU task scheduler, and GPU resource reservation mechanisms. The CPU task scheduling is based on RESCH. The GPU taskscheduling provides prioritized scheduling using our synchronization mechanisms. By intercepting interrupots in the top-half ISRs, the proposed synchronization mechanisms do not need to modify the kernel or device drivers. We indicated a unit of task overhead met within about 10% at the eighth tasks, and within about 4% at the four tasks. In addition, the results of our experimental evaluations Linux-RTXG demonstrate that effective QoS management performance can be achieved by the proposed method without kernel modification. The proposed scheduling framework has realized real-time GPU computing. In future, we will address GPU function such as preemption, and peer-to-peer migration in order to handle more complex real-time problems.

REFERENCES

[1] M. McNaughton, C. Urmson, J. Dolan, and J.-W. Lee, "Motion planning for autonomous driving with a conformal spatiotemporal lattice," in *Proc. ICRA*. IEEE, 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*. IEEE, 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," *Review of Scientific Instruments*, vol. 85, no. 4, p. 045114, 2014.

[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*. ACM, 2010, pp. 94–103.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IISWC*. IEEE, 2009, pp. 44–54.

[7] "CUDA Zone," https://developer.nvidia.com/category /zone/cuda-zone, accessed January 12, 2015.

[8] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *IEEE Des. Test. Comput.*, vol. 12, no. 3, pp. 66–73, 2010.

[9] S. Kato, K. Lakshmanan, R. R. Rajkumar, and Y. Ishikawa, "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: First-Class GPU Resource Management in the Operating System." in *Proc. USENIX ATC*, 2012, pp. 401–412.

[11] 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.

[12] 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.

[13] G. A. Elliott and J. H. Anderson, "Exploring the Multitude of Real-Time Multi-GPU Configurations," in *Proc. RTSS*. IEEE, 2014.

[14] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "$LITMUS^{RT}$: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers," in *Proc. RTSS*, 2006, pp. 111–126.

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

[16] M. Asberg, T. Nolte, S. Kato, R. Rajkumar, and Y. Ishikawa, "ExSched: An External CPU Scheduler Framework for Real-Time Systems," Tech. Rep. CMU-ECE-TR09-12, Tech. Rep., 2009.

[17] 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, 1991.

[18] 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. OSDI*, vol. 8, 2008, pp. 73–86.

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

[20] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time and multimedia systems," in *Photonics West'98 Electronic Imaging*. International Society for Optics and Photonics, 1997, pp. 150–164.

[21] S. Oikawa and R. Rajkumar, "Portable RK: A portable resource kernel for guaranteed and enforced timing behavior," in *Proc. RTAS*. IEEE, 1999, pp. 111–120.

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

[23] V. Yodaiken, "The RTLinux Manifesto," in *Proc. Fifth Linux Expo*, 1999.

[24] 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.

[25] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proc. RTSS*. IEEE, 1998, pp. 4–13.

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

[27] Y. Fujii, T. Azumi, N. Nishio, and S. Kato, “Exploring microcontrollers in GPUs,” in *Proc. APSys*. ACM, 2013, p. 2.

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

[29] 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.

[30] C. Basaran and K.-D. Kang, “Supporting Preemptive Task Executions and Memory Copies in GPGPUs,” in *Proc. ECRTS*, 2012, pp. 287–296.

[31] W. Han, H. Bae, H. Kim, J. Lee, and I. Shin, “GPU-SPARC: Accelerating Parallelism in Multi-GPU Real-Time Systems,” Tech. Rep., 2014.

[32] F. Ridouard, P. Richard, and F. Cottet, “Negative results for scheduling independent hard real-time tasks with self-suspensions,” in *Proc. RTSS*. IEEE, 2004, pp. 47–56.

[33] K. Lakshmanan, S. Kato, and R. R. Rajkumar, “Open problems in scheduling self-suspending tasks,” in *Proc. Real-Time Scheduling Open Problems Seminar*, 2010.

[34] B. Chattopadhyay and S. Baruah, “Limited-Preemption Scheduling on Multiprocessors,” in *Proc. Int. Conf. RTNS*. ACM, 2014, p. 225.

[35] J. Kim, B. Andersson, D. d. Niz, and R. R. Rajkumar, “Segment-Fixed Priority Scheduling for Self-Suspending Real-Time Tasks,” in *Proc. RTSS*. IEEE, 2013, pp. 246–257.

[36] “Nouveau,” http://nouveau.freedesktop.org/wiki/, accessed January 12, 2015.

[37] S. Kato, J. Aumiller, and S. Brandt, “Zero-copy I/O processing for low-latency GPU computing,” in *Proc. Int. Conf. ICCPS*. ACM, 2013, pp. 170–178.

[38] K. Menychtas, K. Shen, and M. L. Scott, “Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack.” in *Proc. USENIX ATC*, 2013, pp. 291–296.