Real-Time 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 increasingly well-designed for high-performance computing. GPU programming environments have also become matured for general-purpose computing on GPUs (GPGPU). Originally, the GPU software stack is tailored to accelerate particular best-effort applications. In recent years, however, GPU technologies have been applied for real-time systems, extending the operating system (OS) modules to support real-time GPU resource management. Unfortunately, such a system extension makes it difficult to maintain the system with version updates, since the OS kernel and device drivers need to be modified at the source code level, preventing continuous research and development of GPU technologies for real-time systems. In this paper, we present a loadable kernel module (LKM) framework called Linux-RTXG, which implements real-time GPU resource management in Linux without making any modification to the OS kernel and device drivers. Linux-RTXG provides novel interrupt interception and independent synchronization mechanisms 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 of introducing Linux-RTXG is comparable to existing kernel-dependent approaches, and multiple GPU applications can be successfully scheduled by Linux-RTXG to meet their priority and quality of service (QoS) requirements in real-time.**

***Keywords*—Graphics Processing Units, Resource Management, Real-Time Scheduling, Operating Systems, Linux**

I. Introduction

Graphics processing units (GPUs) are now common platforms for various data-parallel and compute-intensive applications. Although GPUs are primarily used to accelerate high-performance computing (HPC) applications, their performance advantage is more and more recognized in real-time systems. 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 deployed for various pieces of workload [6].

GPU applications were often best-effort oriented, whose main purpose is to accelerate particular computing blocks. Therefore, conventional GPU technologies are particularly designed for individual data-parallel and compute-intensive workload. However, due to emergence of real-time systems using GPUs, system software support for real-time GPU resource management is becoming a more significant problem. Since the standard system software released by GPU vendors and communities, such as CUDA [7] and OpenCL [8], is not tailored to support real-time systems, a complex undertaking is required to extend the system software including the operating system (OS) kernel and device drivers.

In previous work, GPU resource management has been developed for real-time systems. 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 [10] is a rich set of runtime libraries and device drivers to achieve first-class GPU resource management, where GPU contexts are fully managed in the OS space. The main drawback of these work is that they require detailed information on the implementation of GPU runtime libraries and device drivers, mostly obtained based on reverse engineering.

In general, the GPU software stack is encapsulated by an application programming interfance (API) in runtime 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, 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. In order to satisfy real-time requirements in GPU computing, therefore, CPU and GPU resources must be managed in coordination.

One of notable work on coordinated CPU and GPU resource management for real-time systems is GPUSync [11], [12], which was presented by Elliott et al. to extend CPU scheduling for multiple GPU-aware contexts with budget enforcement. GPUSync was built on top of on the API and ABI layers of proprietary software, providing a configurable framework that can verify the combination of task allocation policies for multicore CPUs and GPU resource allocaton policies for multiple GPUs.

GPUSync is implemented using $LITMUS^{RT}$ [13], which introduces a significant amount of changes to the OS kernel. TimeGraph and Gdev also make some modifications to the device driver. Those built-in approaches to the OS kernel and device drivers require users to install patches, or developers are obliged to maintain patches in order to stay up to date with the latest version releases. This porting work is a complex undertaking against research and development and is not sustainable, given that especially 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 OSes use the LKM to extend real-time capabilities of Linux. In particular, RESCH [14], [15] provides a real-time scheduling framework called ExSched, which 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 by the LKM, without making any modification 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 GPU resource management in Linux. Linux-RTXG allows the system to easily re-configure scheduling algorithms and install their modules at runtime for GPU applications. 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 existing work to highlight an unsolved problem of GPU resource management. In addition to the problem of GPU resource management, we argue what has prevented GPU resource management from patch-free implementation. The system is assumed to be 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. Conceptually, the contribution of this work is applidable to OpenCL and other languages. We define a GPU task as a process running on the CPU that launches 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. 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. In other words, we can create and launch multiple GPU contexts, but they must be executed exclusively on the GPU.

GPU programming requires a set of APIs provided by runtime libraries, such as CUDA Driver API and CUDA Runtime API. A typical approach to GPU programming follows several 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 variants of real-time OSes (RTOSes) [16], [17], [18], [19]. Of particular interest includes those in Linux-based RTOSes [13], [20], [21], [22], [14]. GPUs are also supported in Linux. This work assumes that the OS architecture refers to Linux.

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

GPU computing must deal with the data transfer bandwidth and compute cores as shared resources. Therefore, scheduling and resource reservation must be considered for the GPU as well as the CPU 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 in coordination with CPU tasks on the host side, while monitoring GPU time for GPU kernels. Therefore, we consider the problem of CPU and GPU coordinated scheduling with resource reservation mechanisms.

The recently developed GPUSync framework indeed employs CPU and GPU coordinated scheduling. In 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 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 incompleted 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 of 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], [26].

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 [27] 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 [14], [15] 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 [28] 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 [29], [30]. 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 [31], [32]. There are a lot of ongoing work on the scheduling of self-suspending tasks [33], [34].

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. The scope of this paper is also restricted to time resources but not memory resources. In other words, we consider GPU scheduling problems but not device memory allocation problems.

We support efficient data transfer between the CPU and the GPU using GPU microcontrollers [35], but is not considered as the contribution of this paper. Finally, 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 FENCE and NOTIFY primitives.

III. Design and Implementation

In this section, we present the design and implementation of Linux-RTXG, which provides a framework of CPU and GPU coordinated scheduling based on the LKM. 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 [14], [15].

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 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

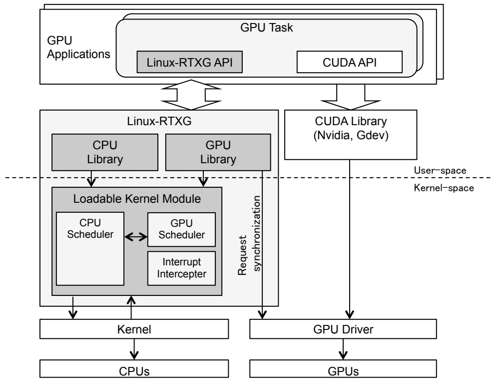


Fig. 1. Architectural overview of Linux-RTXG.

```
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.

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.

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

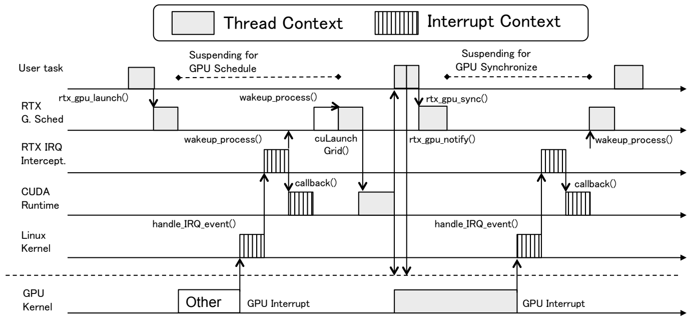


Fig. 3. Execution flow of the GPU task.

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 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.

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.

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) [35], [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, $sched\_yield()$ 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.

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

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

In this section, we evaluate the runtime overhead and real-time performance of Linux-RTXG. The runtime overhead is classified into interrupt interception, independent synchronization, and priority-driven scheduling in Linux. We demonstrate that the overhead of our 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 workload and real-world applications on top of three different device drivers – NVIDIA, Nouveau, and Gdev. We demonstrate that multiple GPU applications co-scheduled by our LKM-based real-time scheduler are successfully prioritized and maintained at the desired frame rate even in the presence of high CPU load.

Experiments are limited to GPU scheduling performance given that CPU scheduling performance has already been demonstrated in previous work [14]. Considering the results of this work and those from the 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.

A. Experimental Setup

Our 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 hyperthreadling cores), and 8 GB main memory. GPU application 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 with NVIDIA CUDA 6.0 and Gdev.

B. Interrupt Interception Overhead

We measured the interrupt interception overhead using the Nouveau GPU driver to quantify the overhead of interception in varied interrupt types. As performance metrics, we adopted elapsed time from the beginning to the end of the ISR.

Figure 5 shows the overhead of interrupt interception. "Raw ISR" represents the original implementation of ISR. "ISR Intercept" represents the modified ISR with the interrupt interception mechanism, while "ISR Intercept w/Func" includes the overhead of identifying the ISR and calling the scheduler thread in addition to "ISR Intercept". The average execution time in 1000 executions is presented with error bars.

Comparing Raw ISR and ISR Intercept, the overhead of introducing the interrupt interception mechanism was $247ns$, which is only 0.8% of Raw ISR. Similarly, the overhead for ISR Intercept w/Func was $790ns$, which is only 2.6% of Raw ISR. This result indicates that the interrupt interception overhead is not significant at all for total performance.

We next compare response times of ISR with and without interrupt interception, as shown in Figure 6. The response time is defined by the elapsed time from the beginning of interrupt processing to the point where the interrupt type (e.g., timer, compute, FIFO, and GPIO) is identified. According to the result, the impact of interrupt interception brings 1.4x longer response times.

We also compared response times when interrupt interception is self-contained in the ISR (top-half) and when it is expanded to the tasklet (bottom-half). This comparison differentiates Linux-RTXG from GPUSync in performance, since GPUSync intercepts the $tasklet\_schedule()$ on top of the proprietary driver, while Linux-RTXG realized ISR-based interception. In this case, the start time of measuring the response time is set at the function call to do_IRQ. Figure 7 shows the result of this comparison. As can be seen, the tasklet-based approach requires about 5x longer response times than the IRQ-based approach. This occurs because the tasklet is typically called after significant ISR processings.

C. Independent Synchronization Overhead

We next quantify the overhead of the independent synchronization mechanism. This mechanism must call $rtx\_gpu\_notify()$ at the time of requested synchronization (e.g., after launching a GPU kernel). To use this mechanism, some initialization procesure is also required. In this measurement, the overhead is defined by the execution time of corresponding APIs.

Figure 8 shows that the initialization overhead could reach $5000us$, whereas the notification overhead is no more than a few microseconds. The initilization procedure calls a Linux process to allocate indirect buffers and control registers for several GPU engines. Albeit a significant overhead, the application program is not much affected by this procedure, since it is called only once at the beginning. On the other hand, the notification overhead is not a major consideration for the application program, though it is a little scattered due to an ioctl system call. When NOTIFY is used to set notificaiton, the overhead was $3.5us$, while it was reduced to $2us$ when FENCE is used.

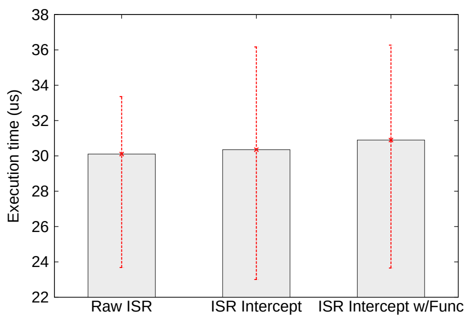


Fig. 5. Interrupt interception overhead.

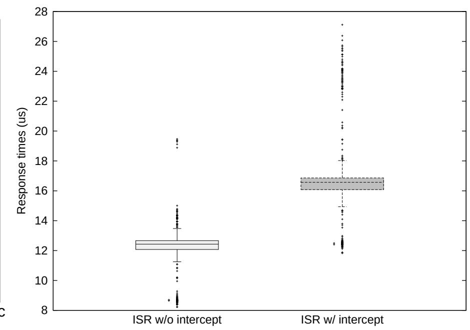


Fig. 6. Impact of interrupt interception.

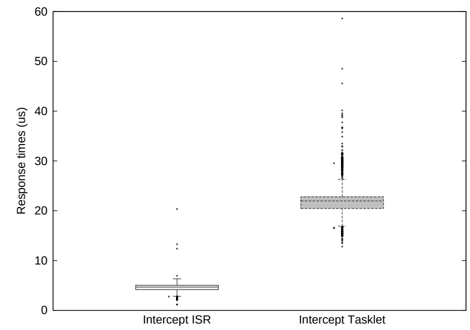
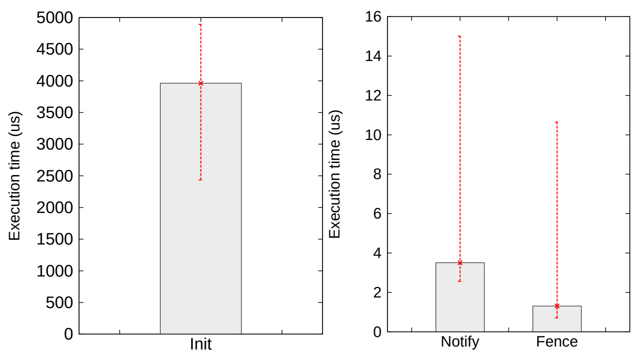


Fig. 7. Comparison of ISR and tasklet.



(a) Overhead of initialization (b) Overhead of notification

Fig. 8. Independent synchronization overhead.

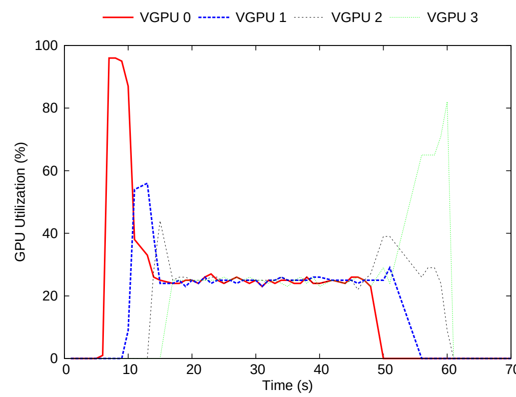


Fig. 11. Utilization of four tasks with the Linux-RTXG's BAND VGPU scheduling. Each tasks had a equal workload and a equal GPU resource allocation.

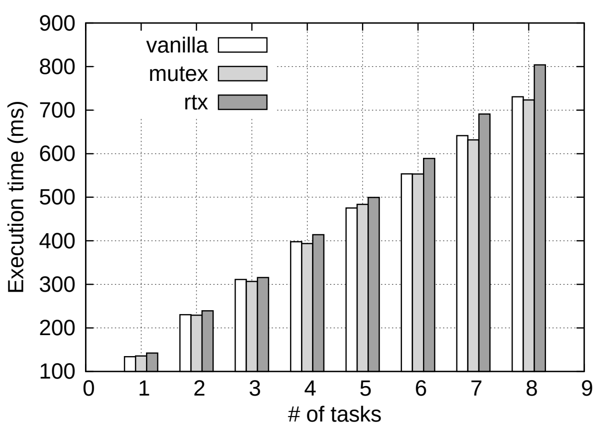


Fig. 9. Scheduling overhead.

D. Scheduling Overhead

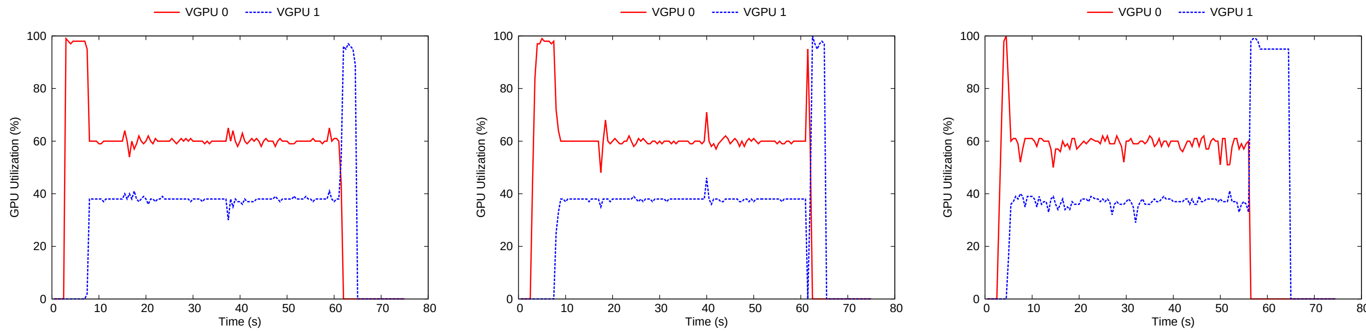
We now evaluate the scheduling overhead posed by the presented Linux-RTXG scheduler. We executed three synthetic tasks – (i) vanilla, (ii) mutex, and (iii) rtx – to measure the overhead. These tasks are based on the microbenchmark program provided by the Gdev project, which tests a GPU loop function. We modified this program to generate multiple GPU tasks by the $fork()$ system call. Each task releases a job ten times periodically, each of which includes the data transfer between the CPU and the GPU, followed by GPU kernel execution. The rtx task was scheduled by Linux-RTXG. The mutex task was limited to a single GPU kernel by using explicit mutual exclusion control similar to the rtx task. The vanilla task was not changed from the original microbenchmark program. The CPU scheduling policy was set to $SCHED\_FIFO$, while the GPU scheduling policy is also based on fixed priorities with GPU resource reservation, called the BAND scheduling policy [10]. The independent synchronization mechanism is employed with NOTIFY.

We measured the average time in 100 times of GPU task execution (1000 jobs). The result is provided in Figure 9. The scheduling overhead increases in proportion as the number of tasks, because the time consumed in queueing GPU tasks is increased. The maximum overhead correponds to 10% of the vanilla task at eight tasks.

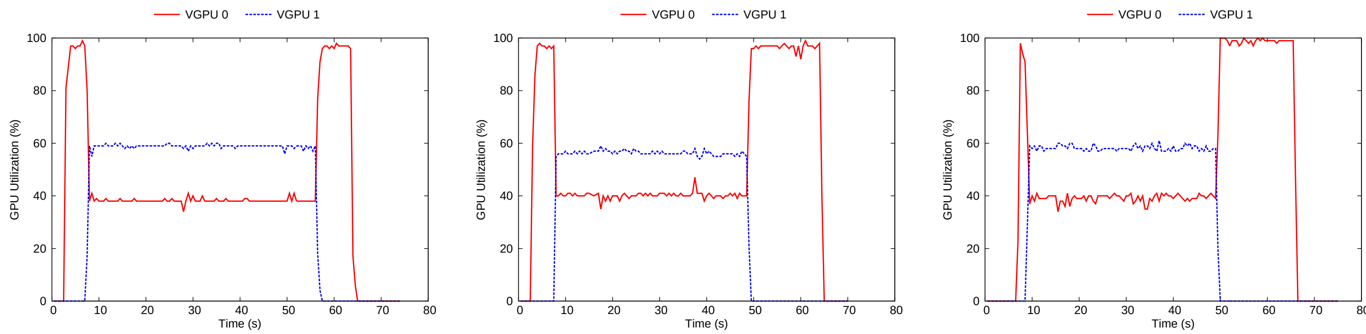
E. Prioritization and QoS Performance

Experiments were also performed to evaluate prioritization and QoS performance for GPU applications provided by Linux-RTXG. We evaluated these pieces of performance of Linux-RTXG on both the proprietary driver (NVIDIA) and the open-source driver (Nouveau), as compared to the built-in kernel approach (Gdev).

QoS management indicates if GPU time of the corresponding task is guaranteed. We evaluated QoS performance by observing how well the tasks are isolated. First, we measured GPU utilization when running two GPU tasks. Each GPU task



(a) FIFO on NVIDIA (c) FIFO on Nouveau (e) FIFO on Gdev



(b) BAND on NVIDIA (d) BAND on Nouveau (f) BAND on Gdev

Fig. 10. GPU utilization of two tasks. Each task executes different types of GPU-intensive workloads with specified GPU reserves (VGPU0 = 40%, VGPU1 = 60%).

has a unique workload with a different size of GPU reserve. One task was allocated to VGPU0, and given 40% of GPU reserve. This task with a high priority has 1.2 times higher workload than the other, which was allocated to VGPU1 with 60% of GPU reserve. The VGPU1 task was scheduled to start approximately $5s$ after the VGPU0 task.

Figure 10 (a) shows the result obtained under the FIFO scheduling policy on the NVIDIA Driver, while that under the BAND scheduling policy is shown in Figure 10(b). The corresponding results on the Nouveau GPU driver are shown in Figures 10 (c) and (d).

Since VGPU0 has higher workload than VGPU1, the GPU tasks are performed in accordance with their workload as shown in Figures 10 (a) and (c). On the other hand, the GPU resource reservation mechanism provided by Linux-RTXG can successfully perform the GPU tasks in accordance with their reserves as indicated in Figures 10 (b) and (d). It is important to remind that these prioritization and resource reservation for GPU applications can be achieved without modifying the OS kernel and device drivers in Linux-RTXG.

The maximum error of the VGPU1 task was approximately 3% under the BAND scheduling policy on the NVIDIA driver, while that of the VGPU0 task was approximately 5%. Using the Nouveau driver, those numbers were 2% and 6%, respectively. The large spikes occurred due to GPU kernel overruns.

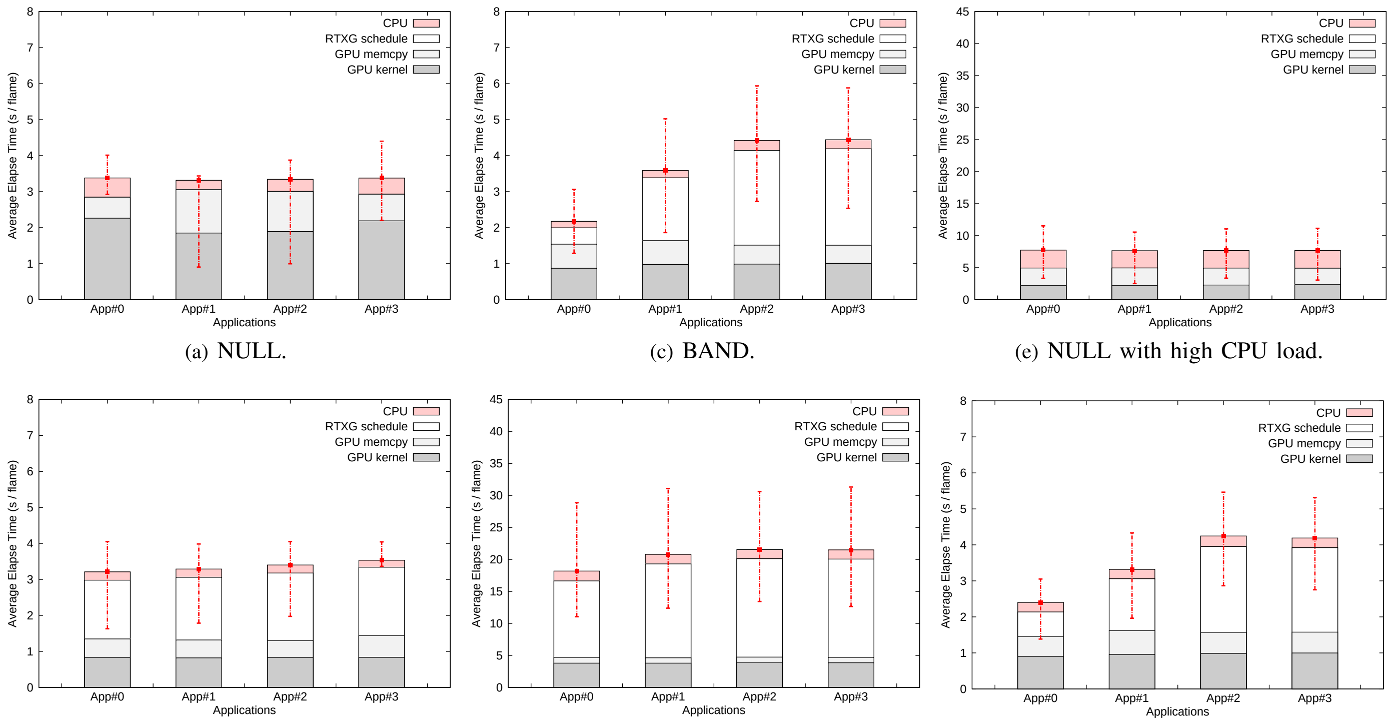
In addition to our loadable kernel approach, we compared performance with the prior work, Gdev. The Gdev scheduling results are shown in Figures 10 (e) and (f). Almost no performance loss is imposed on Linux-RTXG as compared to Gdev. In detail, using the Gdev scheduler, the maximum error of the VGPU1 task was approximately 3% under the BAND scheduling policy, while that of the VGPU0 task was 5%. There is also a large variation on a time basis when using the Gdev scheduler. This is because the runtime functions of Gdev must be called in the kernel space, where other system calls can easily block their operation.

We next measured GPU utilization running four GPU tasks. Each GPU task has the same workload and the same GPU reserve on a different VGPU. Figure 11 shows the isolation result of this scenario. The maximum error of VGPUs was at most appropriately 9%, which is incurred due to the timing of budget replenishment and synchronization latency. In fact, this result almost matches the previous work from Gdev [10] using the built-in kernel approach. As a result, the independent synchronization mechanism employed in Linux-RTXG does not sacrifice scheduling performance.

F. Real-World Application

We finally demonstrate the performance of Linux-RTXG on a real-world application. We tested with the GPU-accelerated object detection program [2] based on the well-known DPM and HOG algorithms. We assume a monitoring system covering the four different directions to East, West, South, and North.

We measure the execution time per frame in six different setups, as shown in Figure 12, where no scheduling is applied (a); fixed priorities are applied with GPU scheduling (b); GPU resource reservation is further applied with the BAND scheduling policy (c); and high CPU load is contending (d,



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

(b) FP. (d) BAND with high CPU load. (f) FP and BAND with high CPU load.

Fig. 12. Execution time of the GPU-accelerated object detection program.

e, f) with the $SCHED\_OTHER$ hackbench task. The GPU tasks are allocated with 60%, 20%, 10%, and 10% of GPU reserves, respectively.

What we can obtain from this experiment is that if a high-CPU load task exists, GPU tasks could be woefully affected in performance, since GPU tasks can not acquire CPU time needed 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 Figure 12 (f). As can be seen, the target programs are not affected by the high CPU load task thanks to the $SCHED\_FIFO$ scheduling, while the GPU execution can also be maintained at the desired frame rate thanks to the GPU scheduling and resource reservation capabilities. These results demonstrate that Linux-RTXG can supply prioritization and QoS performance to real-world applications.

V. Related Work

RGEM [28] and GPU-Sparc [30] provide real-time GPU resource management without the OS kernel and device driver modifications. However, their synchronization mechanism depends on proprietary software. TimeGraph [9], Gdev [10], Ptask [38], 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 can provide real-time GPU resource management with a synchronization mechanism, without modifying the OS kernel and device drivers.

Table II shows a comparison of Linux-RTXG and previous work. GPUSync supports the fixed-priority and the EDF scheduling policies for CPU tasks, while GPUSparc employs the $SCHED\_FIFO$ scheduling policy. Note that Linux-RTXG has demonstrated all features shown in Table II. 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 black-box GPU stacks. Menychtas et al. presented enabling GPU resource management by inferring interactions in the black-box GPU stack [39]. GPU resource management using GPU microcontrollers [26] and in-kernel runtime functions [10] has also been demonstrated to manage the GPU. For these pieces of open-source work, the Nouveau project has been used as a baseline driver [36].

VI. Conclusion

This paper has presented Linux-RTXG, a Linux extention with the LKM-based real-time GPU resource management module. Linux-RTXG accomplished GPU resource management without modifying the OS kernel and device drivers in addition to CPU resource management from the previsou work [14], [15]. We also developed new schemes for GPU resource management. By intercepting the interrupts from the GPU in the top-half ISRs, the developed synchronization mechanism can wait for completion of specific GPU kernels, without the need of modifying the OS kernel and device drivers. The experiments indicated a unit of task overhead met within about 10% at eight tasks, while within about 4% at four tasks. Furthermore, the prioritization and QoS performance of Linux-RTXG were demonstrated with a real-world object detection application, where GPU programs can be protected

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

from high CPU load, while successfully maintaining the desired frame rate on the GPU. To the best of our knowledge, this is the first complete work that achieved GPU resource management with loadable kernel modules.

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] 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. RTSS*, 2006, pp. 111–126.

[14] 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.

[15] 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.

[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, 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. OSDI*, vol. 8, 2008, 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 *Photonics West'98 Electronic Imaging*. International Society for Optics and Photonics, 1997, pp. 150–164.

[20] 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.

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

[22] V. Yodaiken, "The RTLinux Manifesto," in *Proc. Fifth Linux Expo*, 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. RTSS*. IEEE, 1998, pp. 4–13.

[25] M. Spuri and G. Buttazzo, "Scheduling aperiodic tasks in dynamic priority systems," *Real-Time Systems*, 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. 2.

[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. RTSS*. IEEE, 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*, 2010.

[33] B. Chattopadhyay and S. Baruah, "Limited-Preemption Scheduling on Multiprocessors," in *Proc. Int. Conf. RTNS*. ACM, 2014, p. 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. RTSS*. IEEE, 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," 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] 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.

[39] 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.