Coordinated CPU and GPU Resource Management with Loadable Kernel Modules

Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato

***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, 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 being used at an increasing frequency to accelerate various applications, navigation [1] for autonomous driving, object detection [2], tokamak control for fusion reactors [3], user-interactive applications [4], databases [5], and various benchmarks [6] that contain many other applications. GPU performance has been demonstrated by such research.

Previously, GPU applications were required to be only "real-fast" because they were best effort oriented. Recent GPU applications are required to be both "real-time" and "real-fast" due to an increasing number of real-time oriented applications that target real-world problems.

GPUs runtime environments, such as CUDA [7] and OpenCL [8] primarily target best-effort applications; thus, they do not support real-time requirements. Therefore, GPU runtime environments must support real-time scheduling.

In our previous work, we showed how to support GPU resource management[9], [10]. TimeGraph [9] provides GPU scheduling and reservation mechanisms at the device driver level to queue and dispatch GPU commands based on task priorities. Gdev [10] is a resource management system. However, these methods have a weak point; they do not provide fast update architecture and full-functionality because these methods are based on reverse engineering.

In GPU environments, GPU functions are provided by an API of a library and the application binary interface (ABI) of device drivers. Note that CPU time is consumed when GPU kernels are issued by the API or ABI. Thus, if GPUs are to truly satisfy real-time requirements, host-side task management and GPU resource management are required. We have confirmed that a large amount of latency occurs when other tasks appropriate resources at the host side [11] evaluate the data transfer time between host memory and device memory.

GPUSync [12], [13] proposed by Elliott et al. provides CPU task scheduling for multiple GPU aware and budget enforcement on the proprietary runtime. GPUSync realizes a configurable framework that can verify the combination of task allocation policies for multi-core and multi-processor CPUs and GPU kernel allocaton policies for multiple GPUs.

However, GPUSync is implemented using $LITMUS^{RT}$ [14], which introduces a significant amount of changes to the kernel. Gdev also requires the modification of the device driver. Many of these modifications require the user install patches, which introduces a significant burden to both developers and users. Specifically, developers are obliged to maintain patches in order to stay up to date with the latest kernel releases. However, open source software (e.g., Linux) is commonly update before, developers can complete porting their work to the latest kernel.

Linux supports the loadable kernel module (LKM), which can load/unload kernel modules to provide additional kernel functions. We employ the LKM real-time extension called RESCH[15]. RESCH provides a real-time scheduling framework but does not modify the kernel and device drivers. Note that RESCH does not support scheduling of GPU resources.

Contribution: This paper presents a Linux loadable real-time extension for CPU and GPU resource coordination called the Linux real-time extension including GPU resource management (Linux-RTXG). This extension can easily re-configure a resource management algorithms and easily install framework. Linux-RTXG's most significant contribution is the achievement of real-time task scheduling of a GPU environment without kernel modification. To achieve this, Linux-RTXG provides a CPU task scheduler, a GPU kernel scheduler, and GPU kernel reservation mechanisms. The CPU task scheduler is based on RESCH. The GPU kernel scheduler and the GPU kernel reservation mechanisms are based on Gdev. They are integrated in Linux-RTXG using a kernel modification free approach that does not modify the kernel or device driver source code.

Organization The rest of this thesis is as follows. Sec-

tion II discusses the kernel modification free GPU real-time scheduling. Section III presents the design and implementation of Linux-RTXG with specific focus on GPU scheduling. Section IV discusses Linux-RTXG's advantages and disadvantages, and presents experimental results(i.e., quantitative overhead and reservation performance). Section V discusses related work, and we present concluding remarks in Section VI.

II. System Model

Here, we explain the GPU task model employed in this work. In addition, we discuss GPU scheduling and previous works. Next, we describe the obstacles to implement GPU scheduling without kernel modification. Note that this work focuses on a system comprised of multiple GPUs and multi-core CPUs.

A. GPU Task Model

For general purpose GPU computing (GPGPU), CUDA and OpenCL are used to implement GPU applications. This study focuses on CUDA; however, it is possible to adapt the proposed approach to OpenCL. We define a GPU task as a process that executes GPU function whose cyclic executions unit is referred to as a GPU job, and a GPU kernel is a process to be executed by the GPU. Note that the proposed approach supports GPU applications with multiple tasks.

GPUs use a set of the API supported by the runtime environment such as CUDA. Typically, a GPU application has the following steps: (i) *cuCtxCreate* creates a GPU context; (ii) *cuMemAlloc* allocates memory spaces to device memory; (iii) *cuModuleLoad* and *cuMemcpyHtoD* copy the data and the GPU kernel from the host memory to the allocated device memory spaces; (iv) *cuLaunchGrid* invokes the GPU kernel; (v) *cuCtxSynchronize* synchronizes a waiting GPU task to that is completion of GPU kernel, (vi) *cuMemcpyDtoH* transfers resultant data to host memory from device memory; and (vii) *cuMemFree*, *cuModuleUnload*, and *cuCtxDestroy* release the allocated memory spaces and the GPU context.

B. GPU Scheduling

Real-time operating system (RTOS) research [16], [17], [18], [19] has been ongoing for a long time. Among such research, many studies have considered a Lonux-based RTOS [14], [20], [21], [22], [15]. The available OSs for GPUs are limited to Windows, Mac OS and Linux. We have selected Linux in order to achieve real-time processing in GPU environments.

To meet meet real-time constraints in a shared resource environment, such as a multi-core environment, there are two requirements for the scheduler.

- The use of resources according to a specified order
- Limiting the use of shared resources

A basic approach to satisfy the first requirement in a previous study is priority-based scheduling (e.g. rate-monotonic [23] and earliest deadline first [24]) combined with a technique to prevent priority inversion. A basic approach to satisfy the second requirement is resource reservation-based scheduling(e.g. constant bandwidth server [25], total bandwidth server[26]). GPUs must handle data transfer bandwidth and a processing core as a shared resource; thus, we satisfy the two requirements in a manner that is similat to the above mentioned multi-core environment. Our previous study's Gdev involved only the scheduling of GPU access. However, a GPU kernel consumes CPU time because a GPU task is driven by an API. To support real-time scheduling, the scheduler must be able to schedule GPU tasks on the CPU side. Therefore, the framework must have a CPU priority-based scheduler, a GPU priority-based scheduler, and GPU resource reservation mechanisms to realize real-time GPU scheduling. Recently, GPUSync has targeted CPU scheduler and GPU scheduler interdependence. GPUs runtime environments are black-box mechanisms results from GPU environments are provided only GPU vendors, and these environments are closed-source. TimeGraph and Gdev address this problem by ensure transparency using reverse engineering and an open-source driver. GPUSync achieve closed-source compatible by GPU resource management that approaches are the interrupt handling and the arbitrate runtime access.

The other problem occurs by non-preemptive GPU executions and non-preemptive data transfers. Several researches [27], [28] have improvemed the response time by preventing overrun which occurs while dividing the kernel. However, these existing methods concerning real-time GPU are typically experimental, not practical enough. The most difficult problem is self-suspending because GPU is treated as an I/O device. GPU tasks suspend until it receives the results from invoking the processing to GPU, referred to as self-suspending. The self-suspension has been proven as a cause the NP-HARD problem in previous work [29], [30], several researches [31], [32] are working on the scheduling analysis for self-suspension task, they have not been solved yet completely.

Hence, the proposed scheduling framework aims at easier expansion and installation.

GPU Synchronization: The synchronization must be considered for GPU system such as a heterogeneous system. GPUs have two different synchronization techniques. The first technique is memory map-based synchronization called FENCE. FENCE first sends commands to the GPU after the command to take a GPU kernel launch, and then GPU writes the values to memory-mapped space after the GPU kernel is completed. A GPU task monitors the value in the mapped space using polling. Therefore, the task has exclusive access to CPU resources; therefore, response time will be the fastest. The other technique is interruption-based synchronization called NOTIFY. NOTIFY sends GPU commands similar to FENCE, and then GPU will call the interrupt and write a value to the GPU I/O registers. A GPU task is then suspended until interrupt called. Therefore, a task can share CPU resources with other tasks; however, response time is reduced. A detailed architecture is omitted in this thesis; the architecture has been explained in [9], [10], [33].

Gdev uses both techniques, NOTIFY and FENCE to wakeup the waiting task. In addition, NOTIFY is used by the scheduler, and FENCE is used by kernel synchronization. In Gdev, synchronization implementation involves the additional commands sent to a GPU and modification of the device driver 's interrupt handler. GPUSync employs the NOTIFY technique using tasklet intercept [34] on the closed sourced GPU environments. Tasklet is Linux's soft-irq implementation.

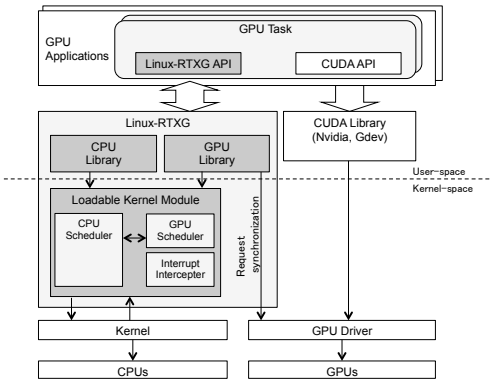


Fig. 1. Overview of Linux-RTXG

GPUSync identifies the interrupt that invokes a kernel using a callback pointer with a tasklet.

Kernel Modification Free Scheduler: We must not modify the kernel and the device driver code to achieve the target kernel modification free characteristic. To select the next GPU kernel to execute, the GPU scheduler must receive a GPU kernel completion notification. This is realized by two methods, i.e., an API driven method and an interrupt-driven method. The API-driven method explicitly wakes the scheduler after the synchronized by API such as $cuCtxSynchronize()$ provided GPU runtime which is used by RGEM[35] . The interrupt-driven method is awaken by an interrupt trigger that is issued by NOTIFY which is used by Gdev, TimeGraph, and GPU Sync. The general $cuCtxSynchronize()$ API synchronizes the completion of all GPU kernels. Therefore, the API-driven method can be used when a GPU context has issued only single kernel. Thus, if a GPU task invokes multiple kernels, we must use the interrupt-driven method to guarantee that no overrun occurs due to the reduction in response time.

The Interrupt-driven method can be synchronized for each kernel; however, it must modify the kernel or the device driver ' s interrupt service routine (ISR). Gdev has achieved independent synchronization mechanisms on the proprietary software; however, independent synchronization mechanisms must also modify the kernel. The challenge is realizing independent synchronization mechanisms that do not modify the kernel.

III. Design and Implementation

In this section, we present the design and implementation of Linux-RTXG, which achieves a real-time GPU scheduling framework without kernel modification. We describe the main contribution of the GPU scheduler and its integration to a CPU scheduler. Note that the discussion of CPU scheduling is minimized because Linux-RTXG is based on RESCH.

A. Linux-RTXG

Figure 1 shows an overview of Linux-RTXG. Linux-RTXG is divided into two parts, the core and the library. The Linux-RTXG core has a CPU task scheduler, a GPU task scheduler, and a GPU resource reservation mechanism. The Linux-RTXG core is loaded into kernel-space; thus, it can use exported kernel functions, such as $schedule()$, $mod\_timer()$, $wake\_up\_process()$, and $set\_cpus\_allowed\_ptr()$. These interfaces are implemented using an input/output control(ioctl) system call, which is a standard way of communicating with a driver. The library includes an independent synchronization method. The independent synchronization method is used only on the NVIDIA driver. If using an open-source GPU driver, such as Nouveau [36], GPU runtime must use part of Gdev. Gdev can manage arbitrary interrupts of the GPU kernel in the user-space mode; therefore, the GPU scheduler does not require independent synchronization.

```
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 of using Linux-RTXG scheduler

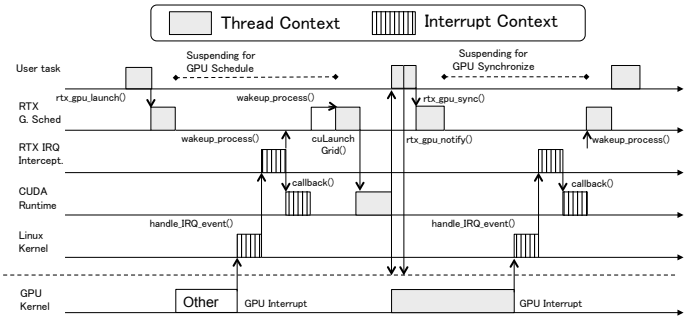


Fig. 3. GPU Scheduling control flow

B. GPU Scheduling

Linux-RTXG is a scheduler that a combination of an API-driven and interrupt-driven. 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 and others do not. Linux-RTXG does not modify the existing CUDA API to cope with proprietary software in order to be independent of GPU runtime. However, the user must add the Linux-RTXG API to existing CUDA applications to use the Linux-RTXG scheduler.

Sample Linux-RTXG scheduler code is shown in Figure 2. GPU tasks are provided with a function by calling Linux-RTXG's API at strategic points.

In Figure 3 is restricted to a single GPU kernel. A GPU

TABLE I. BASIC LINUX-RTXG APIS

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.

task can control the timing of the GPU kernel execution by calling $rtx\_gpu\_launch()$. The task sleeps until it receives an interrupt because an additional GPU kernel cannot be issued until a current task completes execution. When the current task's GPU kernel completes execution, the Linux kernel handles the interrupt. The interrupt interceptor then wakes the sleeping task. The woken task is issued the GPU kernel via a CUDA API, such as $cuLaunchGrid()$. After the GPU kernel is issued, the task registers NOTIFY to monitor interrupt, and is put to sleep until it receives an interrupt. Selection of the subsequent task is performed by the GPU scheduler caused by the interruption of GPU kernel finish. Linux-RTXG controls the order of task execution according to the above flow.

We present a hierarchal scheduling that includes group (virtual GPU) scheduling and GPU kernel scheduling. The virtual GPU scheduling uses a resource reservation mechanism. The GPU kernel scheduling uses a fixed-priority scheduling. Specifically, GPU kernel execution is associated with each scheduling entity while Linux-RTXG groups scheduling entity to virtual GPUs. The virtual GPUs can belong to any of the physical GPUs. In Linux-RTXG, resources are distributed to virtual GPUs.

Figure 4 shows pseudo-code of the scheduling mechanism. Here, $on\_arrival$ is called when a GPU task is requested GPU kernel launch. In $on\_arrival$, a GPU task checks whether the given executions permission to virtual GPU of task itself, and then check the se permit. If a virtual GPU to which the GPU task belongs has not executing permission, the GPU task is queued to wait_queue and is suspended (i.e., it is put to sleep). If the virtual GPU has execution permission, the GPU task is launched. $on\_completion$ is called by a scheduler thread when the execution of the GPU kernel is complete. $on\_completion$ selects up the next virtual GPU and the next GPU task. Next, $on\_completion$ wakes the selected GPU task.

```
  se: The scheduling entity
  se->vgpu: The group that is belonged se
  se->task: The task that is associated with se
  vgpu->parent: The physical GPU identification
 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. Scheduling mechanisms of high-level pseudo-code

C. GPU Synchronization

Here, we describe the independent synchronization mechanism and the interrupt intercept. The independent synchronization mechanism invokes NOTIFY and FENCE without using a GPU runtime API. The interrupt intercept realizes interrupt-driven wakeup of the scheduler without kernel modification. Linux-RTXG uses the independent synchronization mechanisms as much as possible because we do not want to use black–box resource management.

Independent synchronization mechanism from runtime: We present an independent synchronization mechanism for NOTIFY and FENCE. The mechanism invokes an interrupt for NOTIFY, and writes the fence value with the GPU microcontrollers for FENCE. NVIDIA's proprietary software uses the ioctl interface to communicate between the kernel-space and the user-space. These ioctl interfaces provide driver functions, such as device memory allocation, obtaining GPU information and memory mapping. Gdev builds infrastructure that can execute on NVIDIA's driver using these ioctl interfaces. The proposed method uses an ioctl interface similar to Gdev's method for sending commands. Specifically, the proposed method is divided into two parts, Initialize and Notify. Initialize processess generate a dedicated GPU context. This processess include creating virtual address space, allocating an indirect buffer object for sending a command, and creating a context object is required to prepare the FIFO engine and includes, allocating a kernel memory object and mapping the FIFO engine register to host memory space using memory-mapped I/O. The FIFO engine is a GPU microcontroller that receives commands. The Notify processes send commands to a compute engine or a copy engine by $iowrite$ function to the mapped FIFO engine's register. The compute engine and the copy engine do function that switching a GPU context for computing or data copy. This independent synchronization mechanism uses reverse engineering. Therefore, this method depends on the GPU device architecture and the proprietary software interfaces.

Interrupt interception: Interrupts are handled by the ISR registered to the kernel by the device driver. In addition, a scheduler is required to receive the interrupt and to identify the interrupt by reading the GPU status register. The GPU status register must be read before the original ISR resets the GPU status register.

The Linux kernel has structures that hold interrupt parameters called irq_desc for each interrupt number. These

structures have structures called irq_action, including the ISR callback pointer. An irq_desc is allocated to global kernel memory space, and is freely accessible from kernel space. Linux loadable kernel modules can obtain an irq_desc and an ISR callback pointer. We obtain the GPU device driver's ISR callback pointer, and then we register an interrupt interception ISR to the kernel. Thus, we obtain an interrupt interception from the interrupt interception ISR and retain the callback pointer. In addition, I/O registers are mapped to 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 initialized. The interrupt interception identifies an interrupt by reading this mapped-space.

D. Scheduler Integration

The native Linux scheduler has various real-time scheduling policies, such as $SCHED\_DEADLINE$, $SCHED\_FIFO$ and $SCHED\_RR$. $SCHED\_DEADLINE$ is an implementation of a constant bandwidth server and a global earliest deadline first. $SCHED\_DEADLINE$ is included in the Linux 3.14.0 kernel. However, synchronization does not work well with the SCHED_DEADLINE scheduling policy for GPU tasks. Here there are two problems. The first is the implementation of $sched\_yield$—.$sched\_yield()$ uses $yield()$ in kernel space—. The second is the implementation of waking from a sleeping state.

The first problem occurs by releasing the CPU using $sched\_yield()$ while waiting for I/O in polling. Polling (spin) is the exclusive CPU; therefore, a task may once better to release the CPU can obtain good results. However, $sched\_yield$ will set the polling task's remaining execution time to 0 by treating it as a parameter of $SCHED\_DEADLINE$. Thus, the task cannot execute until the runtime is replenished in the next period. Therefore, the task cannot call $sched\_yield()$ between polling. $schedyield()$ is frequently used by device drivers and libraries, as well as GPU environment, and such software is affected by this problem. Depending on the setting, even NVIDIA's CUDA can be affected by this problem. We address this problem by limiting the GPU synchronization method to NOTIFY in the $SCHED\_DEADLINE$ policies.

The second problem is subjected to a check equation 1 when restoring a task from the sleep state. If equation 1 holds, the runtime is replenished and absolute deadline is set to the next cycle deadline.

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

We implement this check by subtracting GPU execution time from $RemainingRuntime$ when a task is restored by GPU kernel execution, with the exception of a task that is restored by 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

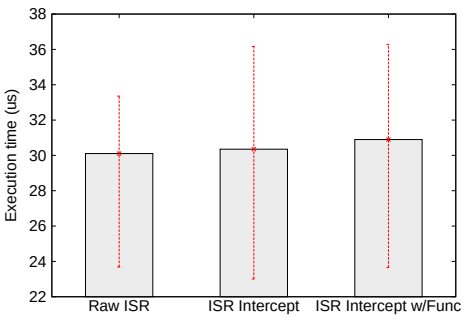


Fig. 5. Interrupt intercept overhead

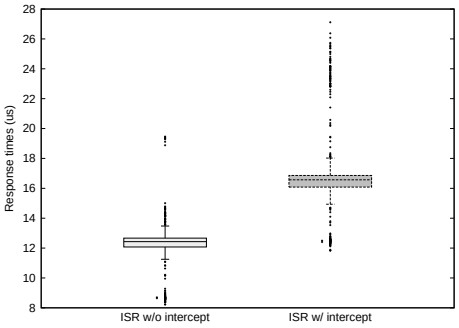


Fig. 6. Response time of interrupt w/o and w/ intercept

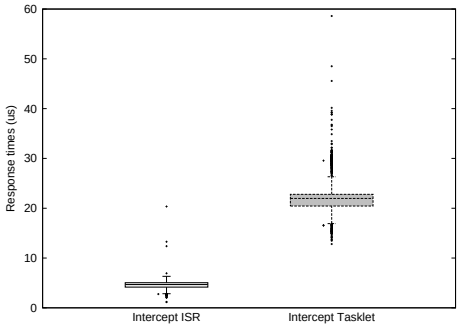
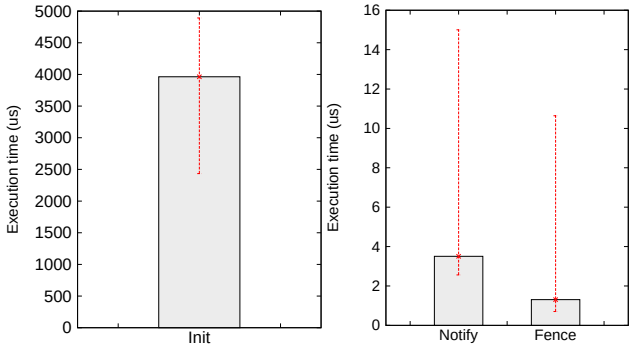


Fig. 7. Response time of the top-half intercept and bottom-half intercept



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

Fig. 8. Interrupt raised method overhead

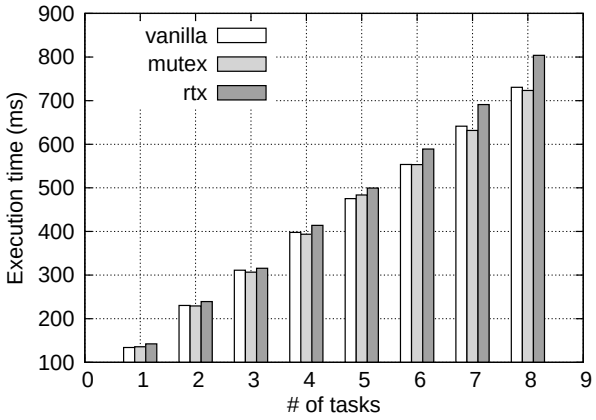


Fig. 9. Scheduling overhead (Time of entire task)

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. 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

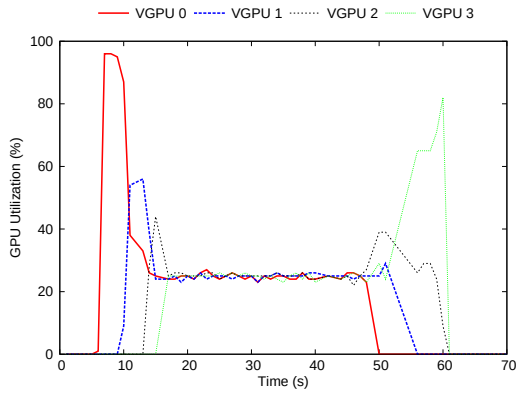


Fig. 11. Utilization of four tasks with the Linux-RTXG's BAND VGPU scheduling. Each tasks had a equal workload and a equal GPU resource allocation.

because task scheduling processing times is increased. Max overhead is 10% at the eight tasks based on "vanilla" time.

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.

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.

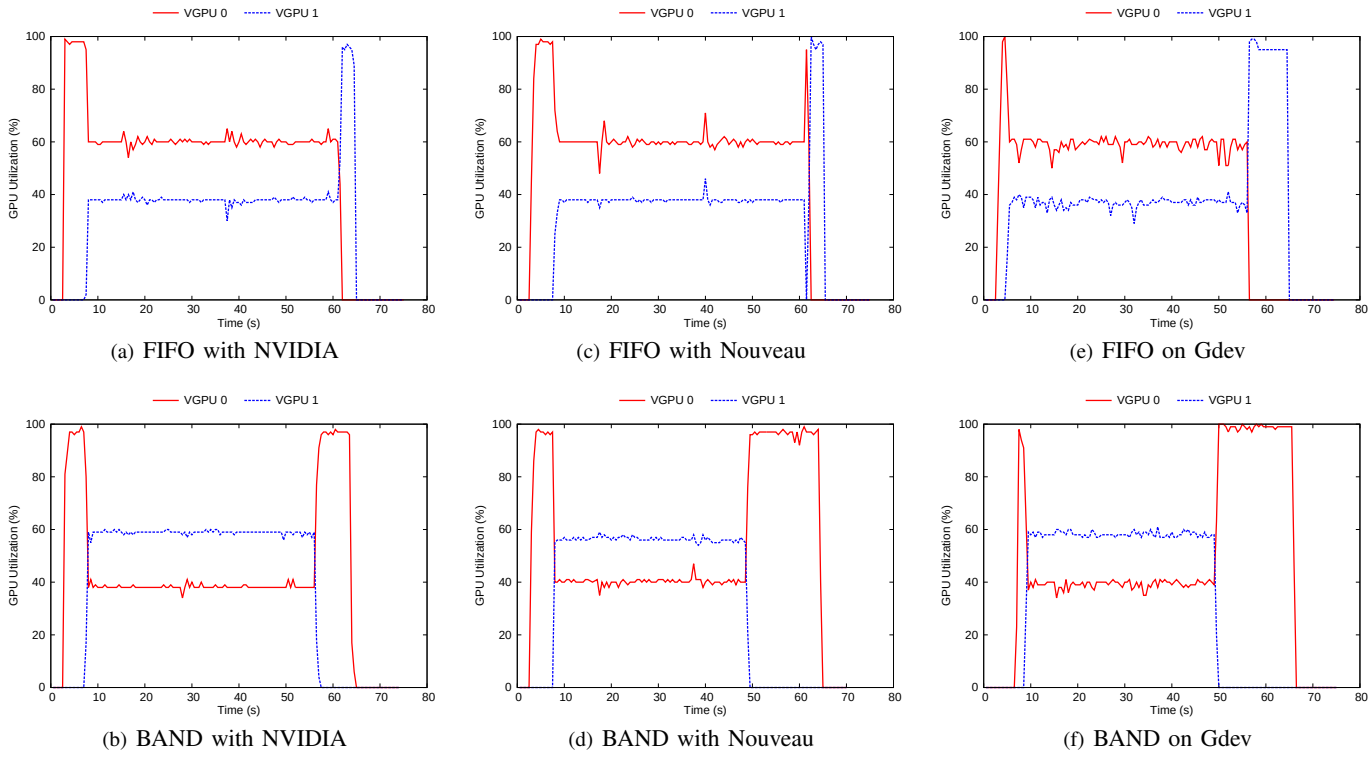
Therefore, the results show that the synchronization mechanism of the proposed Linux-RTXG can schedule tasks without sacrificing performance.

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 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].

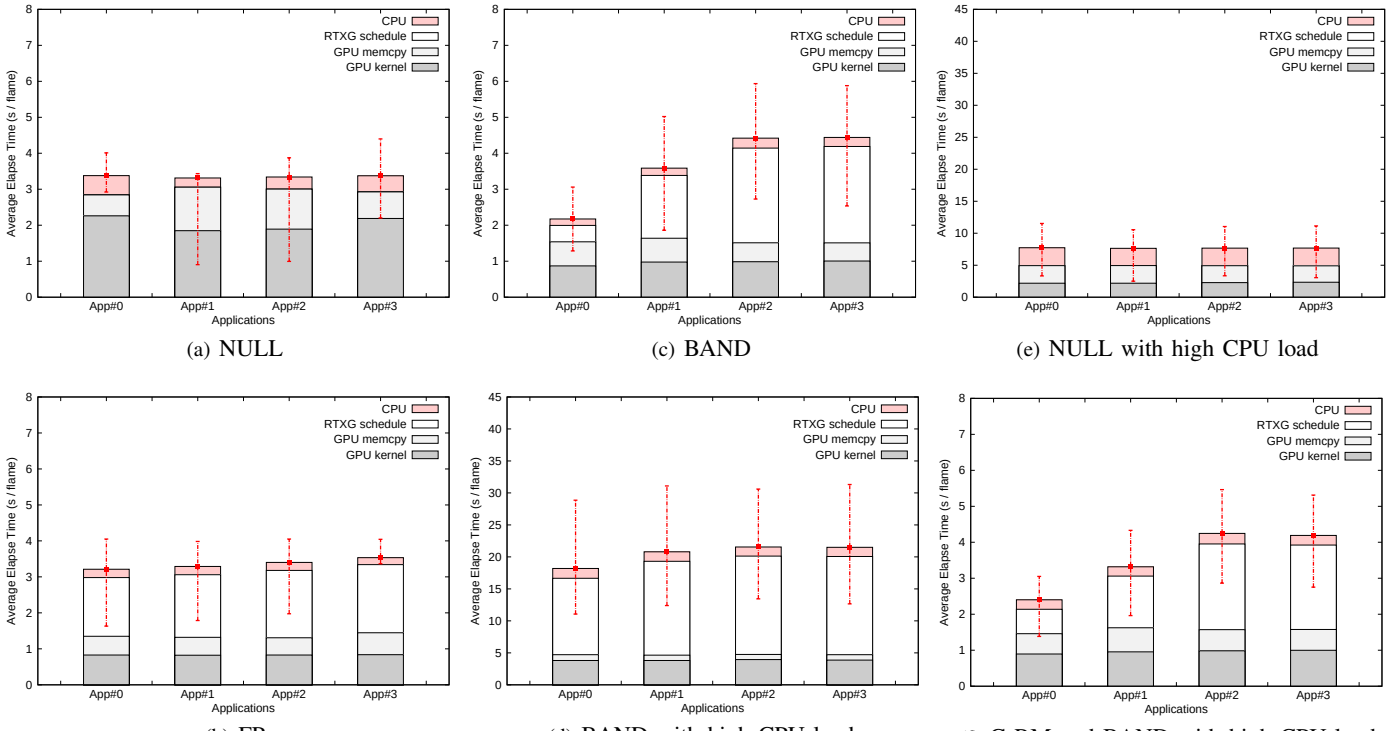
We measure the elapse time of processing one frame image in six situations. Each situation results are shown in Figure 12. Each GPU tasks are allocated difference resources are given 60%, 20%, 10%, and 10% respectively. Figure 12(a) is shown result of executing four tasks without scheduling. All tasks 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).



(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%).



(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.

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.

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

V. DISCUSSION

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

RGEM and GPU-Sparc [28] 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 GPU microcontrollers [33] 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 [33] 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] 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] ——, "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] C. Basaran and K.-D. Kang, "Supporting Preemptive Task Executions and Memory Copies in GPGPUs," in *Proc. ECRTS*, 2012, pp. 287–296.

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

[29] 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.

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

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

[32] 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.

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

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

[35] 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.

[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.