Resource Sharing in GPU-accelerated Windowing Systems

Shinpei Kato[†‡], Karthik Lakshmanan[†], Yutaka Ishikawa[‡], and Ragunathan (Raj) Rajkumar[†]
[†]Department of Electrical and Computer Engineering, Carnegie Mellon University
[‡]Department of Computer Science, The University of Tokyo

Abstract

Recent windowing systems allow graphics applications to directly access the graphics processing unit (GPU) for fast rendering. However, application tasks that render frames on the GPU contend heavily with the windowing server that also accesses the GPU to blit the rendered frames to the screen. This resource-sharing nature of direct rendering introduces core challenges of priority inversion and temporal isolation in multi-tasking environments.

In this paper, we identify and address resource-sharing problems raised in GPU-accelerated windowing systems. Specifically, we propose two protocols that enable application tasks to efficiently share the GPU resource in the X Window System. The Priority Inheritance with X server (PIX) protocol eliminates priority inversion caused in accessing the GPU, and the Reserve Inheritance with X server (RIX) protocol addresses the same problem for resource-reservation systems. Our design and implementation of these protocols highlight the fact that neither the X server nor user applications need modifications to use our solutions. Our evaluation demonstrates that multiple GPU-accelerated graphics applications running concurrently in the X Window System can be correctly prioritized and isolated by the PIX and the RIX protocols.

1 Introduction

Windowing systems in modern operating systems, like the X Window System in Linux/FreeBSD, increasingly use the graphics processing unit (GPU) to provide smart windowing, high-quality graphics, smooth transitions, and improved organization. Recent trends on 3-D graphics applications, such as Compiz Fusion, BumpTop, CoolIris, and Flip3D, are all such intriguing possibilities that benefit from the GPU to enhance the user experience.

In the X Window System, the direct rendering infrastructure (DRI) [19] is commonly used in conjunction with OpenGL – an open-standard graphics library – as a software framework that allows user-space application tasks to directly access the GPU for fast rendering, without using X protocols. The X server itself also uses DRI to access the GPU. Particularly, Gallium3D [8] is a popular open-source OpenGL implementation based on DRI, which is available for many platforms including the Nouveau [23] open-source GPU driver.

Our preliminary evaluation using this open-source graphics stack, provided on the Intel Xeon E5504 machine operating at 2.0 GHz and the NVIDIA GeForce 9500 GT graphics card operating at 432 MHz, has demonstrated that Gallium3D demo programs [8] accelerated on the GPU execute more than 10 times faster than those without acceleration. Newer graphics cards with more advanced GPU architectures, like the NVIDIA GeForce GTX series, would provide further performance improvements. A performance comparison between representative Intel CPU architectures and NVIDIA GPU architectures is listed in Table 1. Observe that GPUs provide considerable benefits over CPUs in performance per power consumption. These performance benefits from GPUs would also apply to data-parallel compute-intensive real-time processing, such as Fast Fourier Transform [24], weather modeling [30], and visual tracking [17].

While the GPU is promising for many graphics applications, commodity GPU drivers are tailored to accelerate *one* particular application in the system, like a video game. In order to support multiple applications running concurrently in real-time, we have developed TimeGraph [27], which provides GPU scheduling and reservation mechanisms at the device-driver level to queue and dispatch GPU commands based on task priorities. However, priority assignment and reservation setup need to be managed by system designers. Particularly, the parameter setup for the X server can significantly affect the overall system performance, since the X server should not disturb application tasks that render frames on the GPU, whereas the X server itself needs to be responsive when these tasks use X protocols to blit frames to the screen.

Unfortunately, there is not yet an explicit solution about how to address such resource-sharing problems raised on the GPU, though interactions between the windowing server and graphics applications in real-time windowing systems have been somewhat studied [5, 7, 18, 28]. Therefore, we need to identify the performance implications of GPU-accelerated windowing systems. Implementation choices, their trade-offs, and their implications must also be studied.

Contributions: This paper presents two resource-sharing protocols for the GPU-accelerated X Window System, which enable the X server to be scheduled with application tasks on a timely basis. The *Priority Inheritance with X server* (PIX) protocol extends the priority inheritance protocol [29] to eliminate priority inversion raised due to the existence of the X

Table 1. Comparison of the Intel CPU architectures and the NVIDIA GPU architectures.

	Core 2 QX9650	Core i7 980XE	GeForce 9500 GT	GeForce GTX 285	GeForce GTX 480
# of processing cores	4	6	32	240	480
Single-precision performance (GFLOPS)	< 100	107	134	933	1350
Memory bandwidth (GB/sec)	7	25	16	159	177
Power consumption (Watt)	130	130	50	183	250

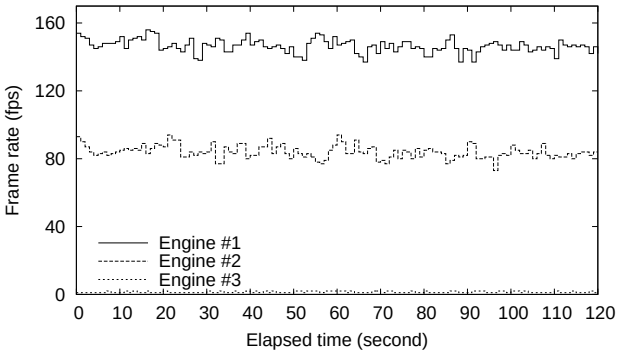


Figure 1. Frame-rates of three graphics tasks with priorities lower than the X server.

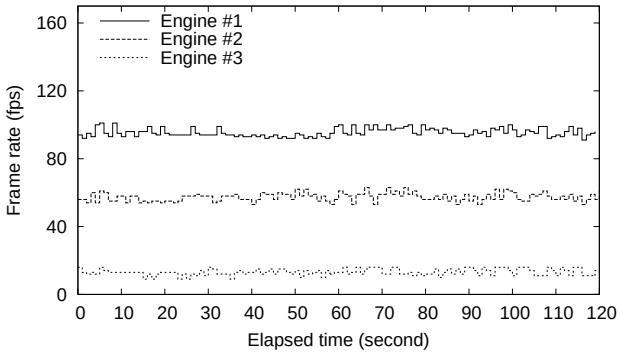


Figure 2. Frame-rates of three graphics tasks with priorities lower than the X server in the presence of an X-client task.

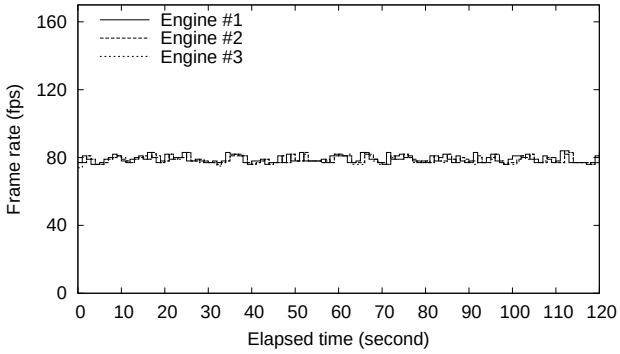


Figure 3. Frame-rates of three graphics tasks with priorities higher than the X server in the presence of an X-client task.

server when accessing the GPU. The *Reserve Inheritance with X server* (RIX) protocol, on the other hand, extends the reserve inheritance protocol [22] to reduce the blocking time imposed on tasks that contend with the X server on the GPU in resource-reservation [26] systems. Although our protocols build upon earlier approaches, new windowing systems need an in-depth look and specialization. Specifically, we describe that these classical protocols need to be modified to address resource-sharing problems specific to the GPU-accelerated X Window System. We also implement our protocols in TimeGraph to evaluate practical performance.

Organization: The rest of this paper is organized as follows. Section 2 identifies resource-sharing problems related to the GPU-accelerated X Window System. Section 3 introduces our system model. Section 4 and Section 5 present the PIX and the RIX protocols respectively. Section 6 describes how to design and implement our protocols in TimeGraph. The capabilities of the PIX and the RIX protocols are evaluated in Section 7. Section 8 discusses the related work in this area. We provide our concluding remarks in Section 9.

2 Problem Observation

In order to understand resource-sharing problems specific to the GPU-accelerated X Window System, we provide a preliminary evaluation using TimeGraph to see how different priorities assigned to the X server affect the performance of application tasks. In the following, the CPU and the GPU share the same priority level.

Figure 1 shows the performance of three instances of the *Engine* widget, which is a Gallium3D demo program [8] that displays a V-6 3-D engine revolving as fast as possible. We first configure the X server to have a priority higher than all the Engine tasks. Among the three Engine tasks, meanwhile, we assign the high priority to *Engine* #1, the medium priority to *Engine* #2, and the low priority to *Engine* #3. In this setup, TimeGraph can correctly prioritize the three Engine tasks as observed in Figure 1, since the X server accesses the GPU only when it blits the rendered frames to the screen.

Assigning a higher priority to the X server, however, causes X client tasks to affect the performance of primary 3-D graphics tasks. Figure 2 shows the performance of three instances of the Engine widget with the same priority order as the previous setup, where another Gallium3D demo program, called *Gears*, is competing with them, as an X-client task that does not directly access the GPU but through the X server. Since the X server requires more GPU resources to serve the Gears task, the performances of the *Engine* #1 and *Engine* #2 tasks degrade accordingly. This performance degradation increases as the X-client workload increases. In contrast, it is interesting to observe that the performance of the *Engine* #3 task is slightly improved, since the X server invokes more frequently executes due to the existence of the Gears task, which decreases the execution rates of the *Engine* #1 and *Engine* #2 task, and reduces interference to the *Engine* #3 task.

Assigning a lower priority to the X server, on the other hand, jeopardizes prioritization among the three Engine tasks, as observed in Figure 3. In this setup, the *Engine* #3 task scheduled at the lowest priority level among the three Engine tasks can preempt the X server that serves the requests issued from the *Engine* #1 and *Engine* #2 tasks that are scheduled at higher priority levels. Hence, there are more interferences.

The performance interference observed above can be attributed to the fact that the X server is a shared resource among application tasks, while the X server itself contends with these tasks for the GPU resource. Specifically, these tasks access the GPU to render frames, while the X server sharing the GPU is used to blit the rendered frames to the screen.

3 System Model

We consider the GPU-accelerated X Window System in soft real-time environments. Application tasks are assigned static (fixed) priorities. If optimal priority assignment is required, the Rate-Monotonic [16] and the Deadline-Monotonic [14] algorithms can be used. This paper is particularly focused on GPU resource management, and thereby we assume that all tasks are provided with sufficient CPU resources.

Application programs use Gallium3D [8], as a DRI-based OpenGL implementation, to directly access the GPU for acceleration. Every time the current frame is rendered on the GPU, each task sends specific requests to the X server to blit the rendered frame to the screen. Such a procedure is often referred to as *double buffering*. Frames may or may not arrive in a periodic fashion. X-client tasks can contend with OpenGL tasks, but these X-client tasks *never* access the GPU directly.

OpenGL tasks and the X server continuously generate *GPU commands* via the user-space driver part of the graphics software stack. GPU commands are collected into *atomic execution regions*. Each atomic set of GPU commands is called a *GPU command group*. While each GPU command group is non-preemptive, the GPU can be preempted between GPU command groups using TimeGraph. It should be noted that when we refer to preemption on the GPU, we denote the preemption at GPU command group boundaries in this paper. The number of GPU command groups submitted to render each single frame is not known a priori.

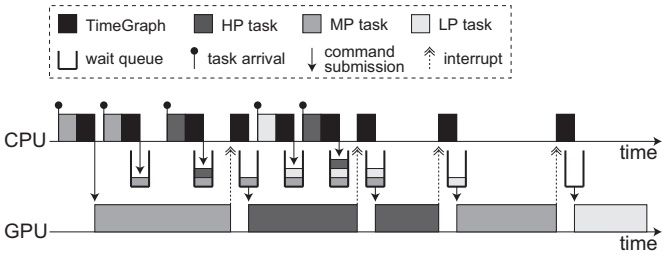


Figure 4. Example of GPU scheduling [27].

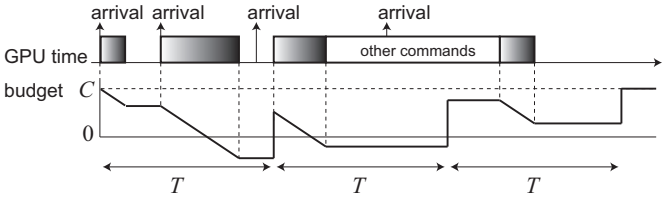


Figure 5. Example of GPU reservation [27].

OpenGL tasks and the X server are scheduled under TimeGraph, where GPU command groups issued from the user space are queued if there are on-the-fly GPU command groups executing on the GPU, and then dispatched to the GPU in accordance with task priorities upon completions of GPU command groups. Coarse-grained GPU reservation support is also provided. More details can be obtained from our TimeGraph website [27]. In order to demonstrate the basic functionality of TimeGraph, however, we quote examples of GPU scheduling and GPU reservation below.

Figure 4 indicates how three tasks with different priorities, a high-priority (HP) task, a medium-priority (MP) task, and a low-priority (LP) task, are scheduled on the GPU under TimeGraph. When the MP task arrives, its GPU command group can execute on the GPU, since no GPU command groups are executing. Given that the GPU and CPU operate asynchronously, the MP task can arrive again while its previous GPU command group is executing. However, the MP task is queued because the GPU is not idle. Even the next HP task is also queued, since further higher-priority tasks may arrive soon. TimeGraph configures the GPU to generate an interrupt to the CPU upon every completion of GPU command group, in order to invoke TimeGraph itself to wake up the highest-priority task in the wait queue. Hence, the HP task is next chosen to execute on the GPU. In this manner, the next instance of the LP task and the second instance of the HP task are scheduled in accordance with their priorities. Thus, higher-priority tasks can be more responsive on the GPU.

Figure 5 shows how GPU reservation works for four GPU command groups issued by the same task. Under GPU reser-

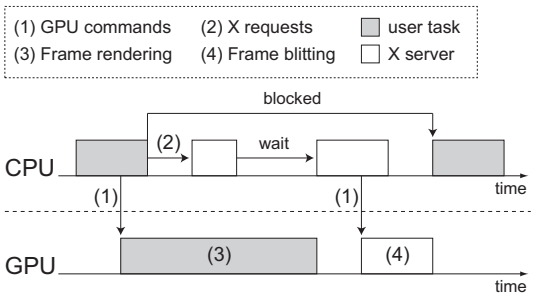


Figure 6. Interaction between a user task and the X server in the DRI-based X Window System.

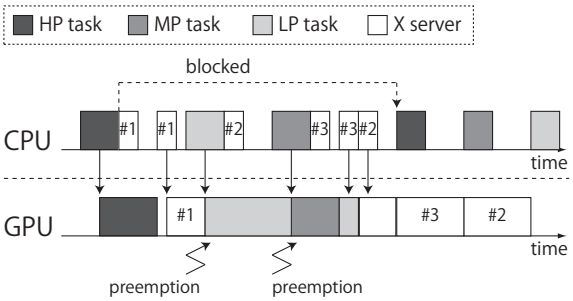


Figure 7. Example of priority inversion in the GPU-accelerated X Window System.

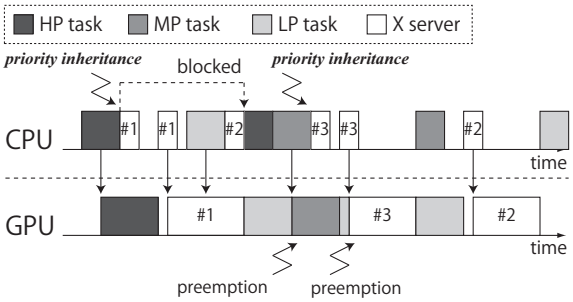


Figure 8. Example of PIX procedure.

vation, tasks can submit GPU command groups to the GPU, only if their budget is greater than zero. It is therefore possible for the budget to be negative, meaning that the task is overrunning out of reservation. The overrun penalty is, however, imposed on the next budget replenishment. Hence, the budget for the next period is given by $e = \min(C, e + C)$. As observed in Figure 5, the budget is initialized to C. When the second GPU command group completes, the budget becomes negative. Hence, the third GPU command group must wait for the budget to be replenished, even though the GPU remains to be idle. Due to the scheduling policy explained above, the fourth GPU command group is blocked, though the budget is positive, when another GPU command group is executing.

Figure 6 illustrates how a user task interacts with the X server in the DRI-based X Window System. When all GPU command groups for rendering the current frame are submitted to the GPU, the user task sends a request to the X server. The user task is then blocked until the X server responds. When the X server is dispatched on the CPU, it receives the request, and waits for the frame, associated with the request, to be rendered on the GPU. Once the frame is rendered, the X server in turn issues another set of GPU command groups to blit the frame to the screen. Finally, the user task is notified of the frame to be displayed on the screen, and resumes execution.

It depends on an X server implementation about how to schedule X requests within the X server. The X requests may or may not be scheduled according to task priorities. For instance, the X server in commodity operating systems is designed to maximize the overall throughput, though priorities are also effective to some extent. If stronger timing support for the X server is required, the X server itself needs to be modified accordingly [5, 18]. In order to make our contribution available for as many versions of the X Window System as possible, we design our system without modifying any part of X server implementation.

4 Priority Inheritance with X Server

We now investigate in depth how the resource-sharing problems observed in Figure 3 are caused. Since the X server is a shared resource in the X Window System, tasks eventually access the X server to blit the frames rendered on the GPU to the screen. Especially, OpenGL tasks are likely to send requests to the X server at the end of each frame to swap data from a hidden space to the screen. Suppose that the X server is assigned a priority lower than these tasks. The execution of the X server on the GPU, servicing the requests issued by a higher-priority task, can be preempted by a lower-priority task that is directly accessing the GPU for rendering frames, since the priority of the lower-priority task is still higher than that of the X server. This leads to *priority inversion* [29].

Figure 7 illustrates how priority inversion occurs on the GPU for three tasks with different priorities higher than the priority of the X server. For simplicity of description, we assume that (i) X requests are scheduled in accordance with task priorities, and (ii) the executions of lower-priority tasks on the GPU can be preempted immediately when higher-priority tasks need to execute, though they must wait for the preceding GPU command group to complete in real environments. In this example, the high-priority task first submits all GPU command groups associated with the current frame to the GPU, and then sends a request (*Request* #1) to the X server to blit its frame to the screen later. The frame is, however, still being rendered on the GPU. Hence, the X server needs to wait for its completion. Once the frame is rendered, the X server accesses the GPU to blit it to the screen. Suppose that the X server is preempted on the GPU by the low-priority task in the middle of frame blitting. This low-priority task also sends a request (*Request* #2) to

the X server. At some later point of time, the medium-priority task further preempts the low-priority task on the GPU, and similarly sends a request (*Request* #3) to the X server. Only after the frame of the medium-priority task is rendered, the X server can resume on the GPU to respond to the high-priority task. The pending X requests are also processed on the GPU in accordance with their priorities, and the X server returns the requests to the medium- and the low-priority tasks respectively. As a consequence, the high-priority task is blocked for a long interval due to priority inversion introduced by the medium-priority task that preempts the X server on the GPU.

This priority inversion problem is slightly different from what was considered in the previous work [29], where a shared resource causing priority inversion is a *critical section* guarded by a lock. A shared resource considered herein, on the other hand, is the X server that is at the same time a *task*. However, we can still apply the same idea. We consider that the real shared resource considered in our scenario is the *GPU resource* used by the X server, and this shared resource is *always* held by the X server. Hence, priority inversion *will* occur every time tasks access the shared resource in our scenario, while it *may or may not* occur in the original scenario.

We now propose the *Priority Inheritance with X server (PIX)* protocol to avoid the priority inversion specific to GPU processing. It is a specialization of the priority inheritance protocol [29] for the GPU-accelerated X Window System. The protocol is defined as follows:

1. Every time a task sends a request to the X server, the priority of the task is inherited by the X server, *only if* it is greater than the current priority of the X server.

2. Every time the X server completes a task request, the X server is assigned the highest priority among remaining tasks whose X requests are pending.

The first policy feature is exactly subject to the original priority inheritance protocol. The second feature is, however, somewhat modified. According to the original protocol, when a task exits from the shared resource, it resumes with the priority that it had at the point of entry into the shared resource [29]. In the X Window System, on the other hand, it is possible that a low-priority task can receive a response from the X server before a high-priority task that sent an X request after the low-priority task in a FIFO order, unless the X server is explicitly implemented to respond in accordance with priorities of tasks sending X requests. In such a case, the X server should still remain at the priority level of the high-priority task. The second policy feature is thus required.

Figure 8 illustrates how the PIX protocol avoids the priority inversion observed earlier in Figure 7. It ensures that the execution of the X server that services *Request* #1 issued by the high-priority task is not preempted on the GPU by the low-priority task. The X server can also preempt the low-priority task on the GPU to service *Request* #3 issued by the medium-priority task. The response times of the high- and the medium-priority tasks are therefore improved accordingly. However,

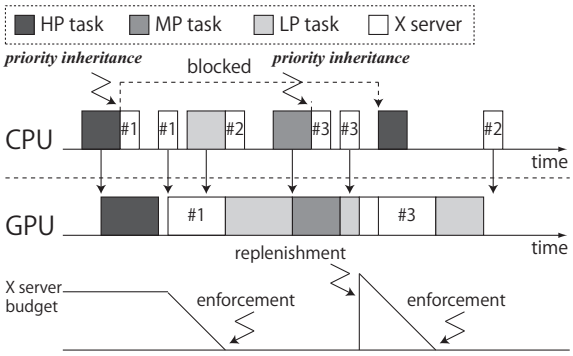


Figure 9. Example of untimely enforcement in the GPU-accelerated X Window System.

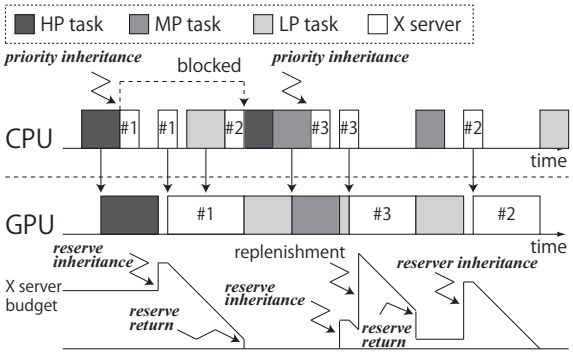


Figure 10. Example of RIX procedure.

the cumulative response time may be increased, as seen in case of *Request* #2, because the X server needs to synchronize with all tasks that access the GPU.

5 Reserve Inheritance with X Server

Resource reservation has been used to provide real-time tasks with temporal isolation [1, 21, 26]. It has also been used in the X Window System [18] to deliver reliable quality of service (QoS) to graphics applications. In fact, it is natural to exploit resource reservation for the X Window System, given that graphics applications may send X requests arbitrarily, and usually have variability in execution times. We however need to remind that the X server is a shared resource.

Figure 9 shows an example of GPU reservation that poses untimely enforcement during the execution of the X server, when the same set of three tasks as used in the previous examples are co-scheduled. Each task is assigned a *reserve* that abstracts the GPU resource by a budget and a period. Suppose that these tasks are assigned reserves of sufficient sizes, while the X server has a small reserve. We claim that such a misspecification likely occurs if system behavior is not precisely known a priori. As observed in Figure 9, the X server is forced to suspend, since its budget is exhausted. All X requests is-

sued by the tasks therefore need to wait until the budget of the X server is replenished, even if the PIX protocol is enabled.

We propose the *Reserve Inheritance with X server (RIX)* protocol that specializes the *reserve inheritance* protocol [22] to account and enforce the execution of the X server efficiently in the GPU-accelerated X Window System. The protocol is defined as follows:

1. Every time a task sends a request to the X server, the X server inherits the reserve of the task, regardless of the priority levels.

2. The reserve may be enforced and replenished even while it has been inherited by the X server.

3. Every time the X server completes a task request, it returns the remaining budget in the inherited reserve to the corresponding task. Let C_i be the budget in the reserve inherited from a task τ_i at some earlier point of time, C_Σ be the total amount of budgets in the reserves inherited by the X server from the tasks waiting for the X server to respond to at the current time t, and C_x be the remaining budget in the reserve of the X server at t. The amount C'_i of budget returned from the X server to τ_i is given by:

$$C'_i = \frac{\max\{\min(C_\Sigma, C_x), 0\} \times C_i}{C_\Sigma}. \quad (1)$$

While the first and second policy features are subject to the original reserve inheritance protocol, the third feature is different, particularly when the X server inherits the reserve from more than one task. Since all X requests are encapsulated by the execution of the X server, the graphics software stack does not recognize which request is currently being serviced by the X server, unless an X server implementation is modified. In other words, we cannot precisely account the execution of the X server on a per-request basis. We therefore approximate the amount of budget to be returned to the task by Equation (1). If the budget C_x of the X server is no less than the total amount C_Σ of budgets in the reserves being inherited by the X server, the same amount C_i of budget as it was inherited from a task τ_i earlier is returned to τ_i, since this condition indicates that the X server could complete servicing the requests within its own reserve. Else, the X server must have used at least some portion of the inherited budgets to service the requests. Since exact accounting per request is not provided, the amount of budget to be returned to τ_i is approximately computed so as to be proportional to the amount of budget inherited from τ_i. As mentioned in Section 3, the budget can be negative under our GPU reservation model, and, in this case, the budget of the next period needs to be decreased accordingly. Hence, we bound the amount of budget to be returned by zero in case that C_x is negative, and the overrun penalty for servicing X requests is imposed on the X server itself later. The resource reservation bandwidth never changes, even for different reservation periods, since exchanging the budget among tasks neither increases nor decreases the total available budget.

Figure 10 shows how the RIX protocol improves response times according to priorities for the same reservation set as the previous example observed in Figure 9. When the high-priority task sends a request to the X server, the reserve of the high-priority task is inherited by the X server. The X server can hence complete servicing *Request* #1 without being forced to suspend, and it promptly responds to the high-priority task. The budget of the X server is then returned to the high-priority task based on Equation (1). The X server next inherits the budget from the medium-priority task before it starts servicing *Request* #3. In this case, however, the budget of the X server is soon replenished to the amount enough to complete servicing *Request* #3 without using any portion of the inherited reserve. Hence, the same amount of budget as it was inherited is returned to the medium-priority task. In contrast, since the X server exhausts the budget inherited from the low-priority task to service it, no budget is returned to it.

6 Implementation

We now describe how to implement the PIX and the RIX protocols in TimeGraph. Since X requests are issued by OpenGL applications only when the `glxSwapBuffers` OpenGL function is called, the points of entry into and exit from the X server must also be specified explicitly to use the PIX and the RIX protocols. We therefore provide two new API functions, `enter_x_server` and `leave_x_server`, to be used in conjunction with the `glxSwapBuffers` function. The `ioctl` system call is internally used to notify the graphics driver of the corresponding API function calls.

In our experience, there are two conceivable approaches to make our API functions available. One may insert our API functions implicitly into the OpenGL library so that the `enter_x_server` and the `leave_x_server` functions are respectively called at the beginning and the end of the `glxSwapBuffers` function. The other approach may leave the decision to user applications when to call our API functions. For instance, they may want to call the `enter_x_server` and the `leave_x_server` functions respectively before and after they call the `glxSwapBuffers` function. From the perspective of legacy applications, modifications to the OpenGL library allow all existing OpenGL applications to take advantage of our solutions without the need of being instrumented and recompiled. We hence adopt the first approach that modifies the `glxSwapBuffers` function.

Applying modifications to the X server would enable more precise budget usage accounting, instead of being forced to roughly approximate it by Equation (1). However, we emphasize that our solution targets all existing implementation of the DRI-based X server and OpenGL applications without applying modifications. More in-depth investigation into how performance isolation is improved by modifying the X server is left open for future work.

Figure 11 illustrates the pseudo-code of our APIs functions implementing both the PIX and the RIX protocols. C_i and P_i

```
1:  function enter_x_server(τ_i) do
2:      if P_i > P_x then
3:          insert τ_i to Q;
4:          P_x ← P_i;
5:      end if
6:      if C_i > 0 then C_x ← C_x + C_i end if
7:      reschedule the X server;
8:  end function

9:  function leave_x_server(τ_i) do
10:     get the highest-priority task τ_k from Q;
11:     if τ_k exists then P_x ← P_k;
12:     else P_x ← P_default; end if
13:     if τ_i exists in Q then
14:         C_Σ = Σ_{τ_j∈Q} C_j;
15:         C_i = max{min(C_Σ, C_x), 0} × C_i/C_Σ;
16:         C_x ← C_x − C_i;
17:         remove τ_i from Q;
18:     end if
19:     reschedule the X server;
20: end function
```

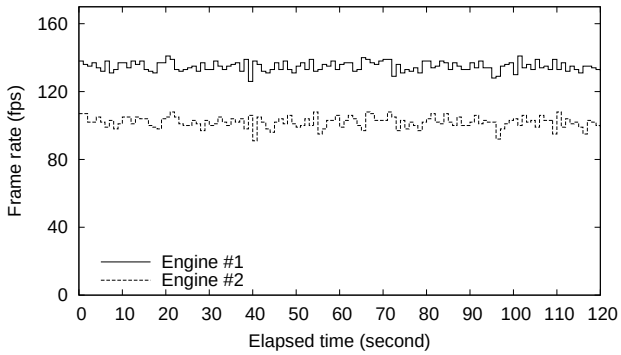
Figure 11. Pseudo-code of the PIX and the RIX protocols implemented on top of TimeGraph.

denote the budget and the priority of a caller task τ_i respectively, while C_x and P_x denote those of the X server. Q is a list of tasks whose reserves are currently inherited by the X server. $P_{default}$ denotes the default priority in the system. We assume that the priorities are statically assigned in advance, and the X server initially has the default priority.

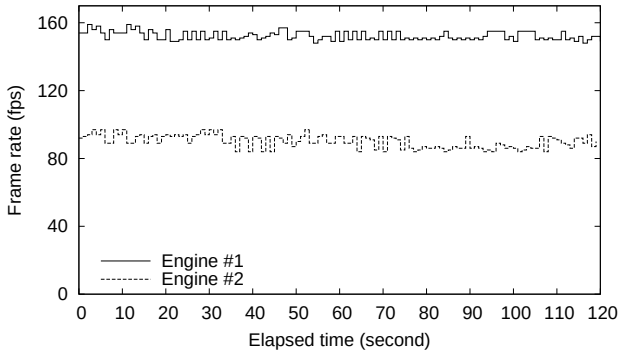
7 Evaluation

We now provide a quantitative evaluation of the PIX and the RIX protocols implemented in TimeGraph, using the Linux kernel 2.6.36 and the Gallium3D OpenGL library. Our experiments perform on the Intel Xeon E5504 CPU (2.0 GHz) and the NVIDIA GeForce 9500 GT graphics card (432 MHz), which is the same environment as we used for our preliminary evaluation in Section 1. Two representative Gallium3D demo programs [8], (i) the Engine widget as a primary OpenGL application and (ii) the Gears widget as an X client application, are used for our evaluation. These programs are greedy, and attempt to get as high a frame-rate as possible. We also use *MPlayer* [10] as another X client application that represents a regularly-behaved periodic graphics workload.

In our experiments, the X server is by default assigned a priority lower than OpenGL application tasks, since assigning a higher priority to the X server would cause an unbounded performance degradation of OpenGL applications, as discussed in Section 2. Each task has the same priority level on the CPU and on the GPU. The tasks involved in our evaluation are assigned *real-time* Linux priorities on the CPU to protect them from being affected by other background tasks.



(a) w/o the PIX protocol.



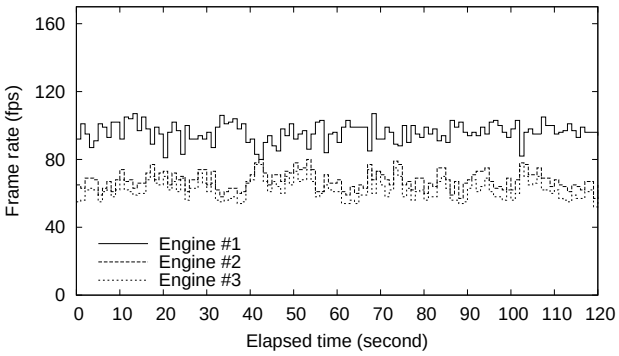
(b) w/ the PIX protocol.

Figure 12. Frame-rates of two graphics tasks.

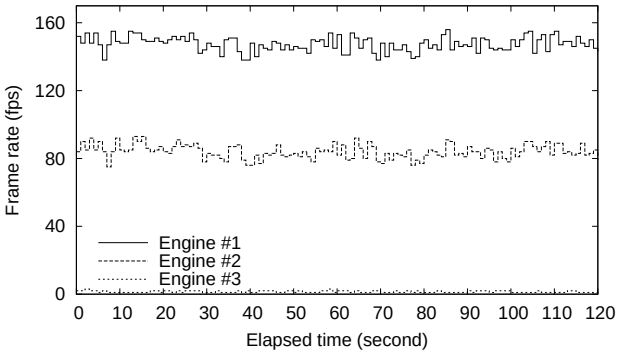
7.1 Priority Inheritance

We first evaluate the effectiveness of the PIX protocol. Figure 12 shows the impact on prioritization among two instances of the Engine widget executing with and without the PIX protocol, where the *Engine* #1 task is assigned a priority higher than the *Engine* #2 task. Comparing Figure 12 (a) and (b), we can observe that the PIX protocol improves the frame-rate of the *Engine* #1 task (a higher-priority task) to 156 fps from 138 fps on average, by reducing the adverse effect from the *Engine* #2 task (a lower-priority task). This illustrates that stronger prioritization can be provided by the PIX protocol in the GPU-accelerated X Window System.

We next evaluate the scalability in performance of the PIX protocol by adding another instance of the Engine widget, *Engine* #3, with a priority lower than the *Engine* #1 and the *Engine* #2 tasks. As shown in Figure 13 (a), prioritization among the three tasks misbehaves in a worse fashion without the PIX protocol, since lower-priority tasks are more likely to preempt the X server servicing the requests issued by higher-priority tasks. On the other hand, the three tasks are correctly prioritized by the PIX protocol, as shown in Figure 13 (b). Specifically, the the frame-rates of the *Engine* #1 and the *Engine* #2 tasks are improved to 150 fps from 98 fps and to 82 fps from 64 fps respectively on average, by preventing the *Engine* #3 task from interfering with them.



(a) w/o the PIX protocol.



(b) w/ the PIX protocol.

Figure 13. Frame-rates of three graphics tasks.

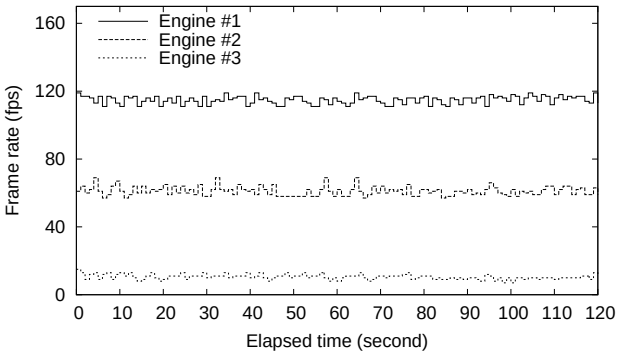
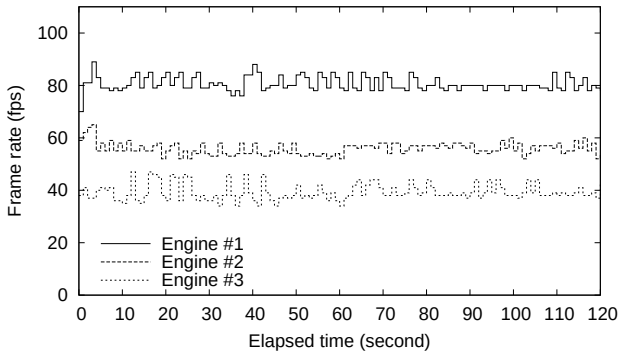
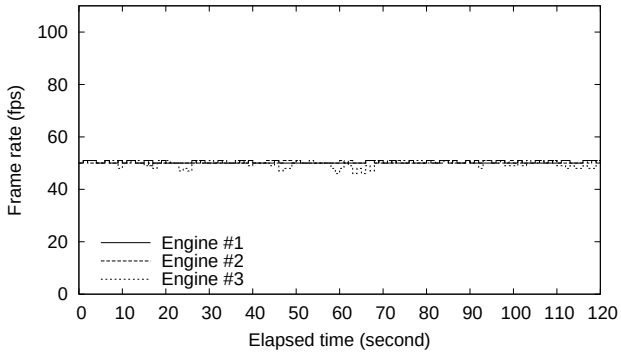


Figure 14. Frame-rates of three graphics tasks in the presence of an X-client task under PIX.

We now evaluate the performance degradation of OpenGL tasks in the presence of an X-client task competing for the GPU resource. Figure 14 shows how the frame-rates of the three Engine tasks are degraded, when the Gears task contends with them as an X-client task. The *Engine* #1 and *Engine* #2 tasks have frame-rates decreased by about 20 ∼ 30 fps as compared with the previous case observed in Figure 13 (b). The *Engine* #3 task, on the other hand, obtains a slightly higher frame-rate for the same reason described in Section 2 regarding Figure 2. Comparing Figure 2 and Figure 14, however, we can observe that the performance loss for the Engine tasks is reduced by the PIX protocol.



(a) w/ the PIX protocol and w/o the RIX protocol.

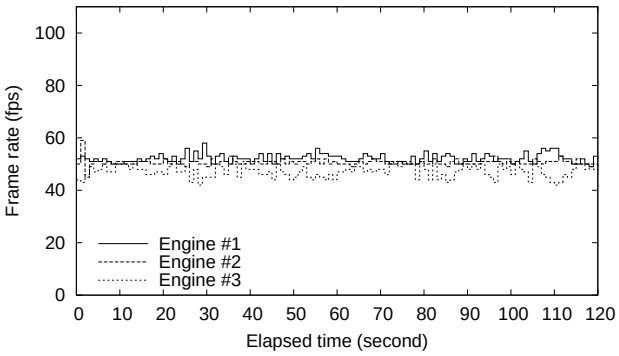


(b) w/ both the PIX and the RIX protocols.

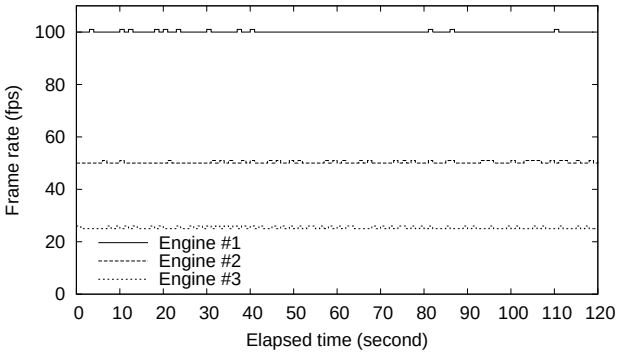
Figure 15. Frame-rates of three graphics tasks assigned reserves of the same size.

7.2 Reserve Inheritance

We next evaluate the effectiveness of the RIX protocol. Figure 15 shows how the three Engine tasks, assigned reserves of the same size of 5*ms* every 20*ms*, are isolated under the RIX protocol, when the Gears task contends with them as an X-client task. The X server is also assigned a reserve of 5*ms* every 20*ms*. The PIX protocol is by default used in this setup. As shown in Figure 15 (a), performance isolation among the three Engine tasks is not adequate without the RIX protocol. Since the budget of each task is simply carried over for the next frame of itself, the *Engine* #1 task, assigned the highest priority, can execute at a faster rate than the reserve period. This reserve overrun of the *Engine* #1 task in turn causes the frame-rate of the *Engine* #3 task, assigned the lowest priority, to decrease below the reserve period. When the RIX protocol is used, however, the three Engine tasks are well-isolated at a frame-rate of 50 fps, corresponding to a reserve period of 20*ms*. Since the remaining reserves are inherited by the X server, each task hardly executes at a faster rate than the reserve period. Thus, the X server is less of a bottleneck.



(a) w/ the PIX protocol and w/o the RIX protocol.



(b) w/ both the PIX and the RIX protocols.

Figure 16. Frame-rates of three graphics tasks assigned reserves of different sizes.

Figure 16 shows the impact on performance isolation in a setup similar to the case observed in Figure 15, except that the three Engine tasks are assigned reserves of different sizes. Specifically, reserves are assigned in such a way that *Engine* #1 runs for 5*ms* every 10*ms*, *Engine* #2 runs for 5*ms* every 20*ms*, *Engine* #3 runs for 5*ms* every 40*ms*, and the X server runs for 1*ms* every 10*ms*. Due to a small reserve of the X server, the X server is likely a bottleneck without the RIX protocol in this setup, which affects performance isolation, as shown in Figure 16 (a). However, the RIX protocol removes this bottleneck, which results in providing sufficient performance isolation, as shown in Figure 16 (b).

We have so far studied the impact on OpenGL tasks. We now consider a likely scenario where users are watching videos, while 3-D browser and desktop widgets are running in the background. To test this scenario, we run *MPlayer* as an X-client task, playing a H264-compressed video with a frame size of 1920×800 and a frame-rate of 24 fps, when three Engine tasks contend with the MPlayer task. The same setup as the case observed in Figure 16 is applied for the reserve of each task. The MPlayer task is not assigned a reserve, since it accesses the GPU through the X server. Unfortunately, MPlayer does not synchronize with the X server, meaning that it can start the next frame before the previous frame is blitted to the screen. As a result, the delay of frame rendering cannot be observed inside the MPlayer task, but can only be recognized in the X server. We therefore measure the total amount of time consumed from the beginning to the end of video playback, and compute the average frame-rate. According to our measurement, the average frame-rate achieved by MPlayer is 12.7 fps without the RIX protocol, while it is maintained at 23.9 fps by using the RIX protocol. This result illustrates that the RIX protocol plays an important role in providing not only OpenGL tasks but also X-client tasks with isolation.

8 Related Work

Windowing Systems: CPU scheduling for windowing systems has been studied in the real-time systems literature [5, 7, 18, 28]. Although these previous work considered prioritization and/or isolation for better windowing server management, resource sharing among the windowing server and application tasks was not the focus of study, and modifications to the windowing server itself were required. On the other hand, our solutions focus on GPU resource-sharing problems to provide reliable prioritization and isolation in the X Window System, without modifications to the X server.

Resource-Sharing Protocols: Resource sharing has been a primary concern for real-time systems. The priority inheritance protocol [29] addressed the priority inversion problem for resource-sharing fixed-priority systems. This protocol was extended as the reserve inheritance protocol [22] to incorporate the resource reservation [26] strategy. The same set of resource-sharing problems was also addressed for dynamic-priority systems [2, 13]. Our goal is to further extend these protocols as the PIX and the RIX protocols for GPU-accelerated windowing systems. In fact, the RIX protocol is similar to *reservation reclaiming* approaches [4, 11, 15, 20] in that the remaining reserve capacity is distributed to other tasks. However, our contribution is distinguished, since our approach is specialized to improve the responsiveness of the X server shared among tasks, while the previous reclaiming approaches aimed for improving the total reserve usage.

GPU Resource Management: GERM [3] provided GPU resource management primitives for fairness. While GERM is useful for multi-tasking environments, prioritization and isolation are not supported. WDDM [25] was developed as a graphics driver model for the Microsoft Windows to provide abstraction for GPU resource management. However, resource management policies are not explicitly exposed to user applications. VMGL [12], GViM [9], and VMware's Virtual GPU [6] enabled virtualization on the GPU. However, these work do not provide mechanisms to schedule GPU commands. TimeGraph [27] has been developed as a real-time GPU scheduler that provides prioritization and isolation support for multi-tasking environments. In this paper, we use TimeGraph as the underlying GPU driver for design and implementation of the PIX and the RIX protocols to address resource-sharing problems in the GPU-accelerated X Window System.

9 Concluding Remarks

We have presented two resource-sharing protocols for the GPU-accelerated X Window System, *Priority Inheritance with X server (PIX)* and *Reserve Inheritance with X server (RIX)*, which address the priority inversion problem introduced on the GPU. We have also designed and implemented these protocols in TimeGraph without applying any modifications to the X server and user applications to support as many existing systems as possible. Our evaluation using an NVIDIA graphics card demonstrated that the PIX protocol significantly improved the frame-rates of high-priority OpenGL tasks, while the RIX protocol provided reliable isolation among competing OpenGL and X-client tasks. To the best of our knowledge, this is the first piece of work that addressed resource-sharing problems for GPU-accelerated windowing systems.

In future work, we would like to address resource-sharing problems across the CPU and GPU. We are also interested in the analysis of blocking time introduced by the PIX and the RIX protocols to use them in hard(er) real-time environments. Generalization of our solution beyond the X Window System is a further challenging issue.

References

[1] L. Abeni and G. Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 3–13, 1998.

[2] T.P. Baker. A Stack-Based Resource Allocation Policy for Real-Time Processes. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 191–200, 1990.

[3] M. Bautin, A. Dwarakinath, and T. Chiueh. Graphics Engine Resource Management. In *Proc. of the Annual Multimedia Computing and Networking Conference*, 2008.

[4] M. Caccamo, G. Buttazzo, and D. Thomas. Efficient Reclaiming in Reservation-Based Real-Time Systems. *Real-Time Systems*, 54(2):198–213, 2005.

[5] S. Childs and D. Ingram. The Linux-SRT Integrated Multimedia Operating Systems: Bringing QoS to the Desktop. In *Proc. of the IEEE Real-Time Technology and Applications Symposium*, pages 135–140, 2001.

[6] M. Dowty and J. Sugeman. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[7] N. Feske and H. Hartig. DOPE – A Window Server for Real-Time and Embedded Systems. In *Proc. of the ACM Workshop on System-level Virtualization for High Performance Computing*, pages 74–77, 2003.

[8] Gallium3D. `http://www.mesa3d.org/`.

[9] V. Gupta, A. Gavrilovska, N. Tolia, and V. Talwar. GViM: GPU-accelerated Virtual Machines. In *Proc. of the ACM Workshop on System-level Virtualization for High Performance Computing*, pages 17–24, 2009.

[10] MPlayer Headquarters. `http://www.mplayerhq.hu/`.

[11] S. Kato, R. Rajkumar, and Y. Ishikawa. AIRS: Supporting Interactive Real-Time Applications on Multicore Platforms. In *Proc. of the Euromicro Conference on Real-Time Systems*, 2010.

[12] H.A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara. VMM-Independent Graphics Acceleration. In *Proc. of the ACM/Usenix International Conference on Virtual Execution Environments*, pages 33–43, 2007.

[13] G. Lamastra, G. Lipari, and L. Abeni. A Bandwidth Inheritance Algorithm for Real-Time Synchronization in Open Systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 151–160, 2001.

[14] J. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. *Performance Evaluation, Elsevier Science*, 22:237–250, 1982.

[15] C. Lin and S. Brandt. Improving Soft Real-Time Performance through Better Slack Reclaiming. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 410–421, 2005.

[16] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20:46–61, 1973.

[17] O.M. Lozano and K. Otsuka. Real-Time Visual Tracker by Stream Processing. *Journal of Signal Processing Systems*, 57(2):285–295, 2009.

[18] N. Manica, L. Abeni, and L. Palopoli. QoS Support in the X11 Window Systems. In *Proc. of the IEEE Real-Time Technology and Applications Symposium*, pages 103–112, 2008.

[19] K.E. Martin, R.E. Faith, J. Owen, and A. Akin. *Direct Rendering Infrastructure, Low-Level Design Document*. Precision Insight, Inc., May 1999.

[20] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. IRIS: A New Reclaiming Algorithm for Server-Based Real-Time Systems. In *Proc. of the IEEE Real-Time and Embedded Technology and Aplications Symposium*, pages 211–218, 2004.

[21] C.W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proc. of the IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, 1994.

[22] D. Niz, L. Abeni, S. Saewong, and R. Rajkumar. Resource Sharing in Reservation-Based Systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 171–180, 2001.

[23] Nouveau. `http://nouveau.freedesktop.org/`.

[24] A. Nukada and S. Matsuoka. Auto-Tuning 3-D FFT Library for CUDA GPUs. In *Proc. of the ACM/IEEE Conference on Supercomputing*, 2009.

[25] S. Pronovost, H. Moreton, and T. Kelley. Windows Display Driver Model (WDDM v2) And Beyond. In *Windows Hardware Engineering Conference*, 2006.

[26] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A Resource Allocation Model for QoS Management. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 298–307, 1997.

[27] Real-Time and Multimedia Laboratory at CMU. `http://rtml.ece.cmu.edu/projects/timegraph/`.

[28] J.E. Sasinowski and J.K. Strosnider. ARTIFACT: A Platform for Evaluating Real-Time Window System Designs. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 342–352, 1995.

[29] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39:1175–1185, 1990.

[30] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka. An 80-Fold Speedup, 15.0 TFlops, Full GPU Acceleration of Non-Hydrostaic Weather Model ASUCA Production Code. In *Proc. of the ACM/IEEE Conference on Supercomputing*, 2010.