TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments

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

Abstract

The Graphics Processing Unit (GPU) is now commonly used for graphics and data-parallel computing. As more and more applications tend to accelerate on the GPU in multi-tasking environments where multiple tasks access the GPU concurrently, operating systems must provide prioritization and isolation capabilities in GPU resource management, particularly in real-time setups.

We present TimeGraph, a real-time GPU scheduler at the device-driver level for protecting important GPU workloads from performance interference. TimeGraph adopts a new event-driven model that synchronizes the GPU with the CPU to monitor GPU commands issued from the user space and control GPU resource usage in a responsive manner. TimeGraph supports two priority-based scheduling policies in order to address the trade-off between response times and throughput introduced by the asynchronous and non-preemptive nature of GPU processing. Resource reservation mechanisms are also employed to account and enforce GPU resource usage, which prevent misbehaving tasks from exhausting GPU resources. Prediction of GPU command execution costs is further provided to enhance isolation.

Our experiments using OpenGL graphics benchmarks demonstrate that TimeGraph maintains the frame-rates of primary GPU tasks at the desired level even in the face of extreme GPU workloads, whereas these tasks become nearly unresponsive without TimeGraph support. Our findings also include that the performance overhead imposed on TimeGraph can be limited to 4-10%, and its event-driven scheduler improves throughput by about 30 times over the existing tick-driven scheduler.

1 Introduction

The Graphics Processing Unit (GPU) is the burgeoning platform to support high-performance graphics and data-parallel computing, as its peak performance is exceeding 1000 GFLOPS, which is nearly equivalent of 10 times that of traditional microprocessors. User-end windowing systems, for instance, use GPUs to present a more *lively* interface that improves the user experience significantly through 3-D windows, high-quality graphics, and smooth transition. Especially recent trends on 3-D browser and desktop applications, such as SpaceTime, Web3D, 3D-Desktop, Compiz Fusion, BumpTop, Cooliris, and Windows Aero, are all intriguing possibilities for future user interfaces. GPUs are also leveraged in various domains of general-purpose GPU (GPGPU) processing to facilitate data-parallel compute-intensive applications.

Real-time multi-tasking support is a key requirement for such emerging GPU applications. For example, users could launch multiple GPU applications concurrently in their desktop computers, including games, video players, web browsers, and live messengers, sharing the same GPU. In such a case, quality-aware soft real-time applications like games and video players should be prioritized over live messengers and any other applications accessing the GPU in the background. Other examples include GPGPU-based cloud computing services, such as Amazon EC2, where virtual machines sharing GPUs must be prioritized and isolated from each other. More in general, important applications must be well-isolated from others for quality and security issues on GPUs, as on-line and user-space programs can create *any* arbitrary set of GPU commands, and access the GPU *directly* through generic I/O system calls, meaning that malicious and buggy programs can easily cause the GPU to be overloaded. Thus, GPU resource management consolidating prioritization and isolation capabilities plays a vital role in real-time multi-tasking environments.

GPU resource management is usually supported at the operating-system level, while GPU program code itself including GPU commands is generated through libraries, compilers, and runtime frameworks. Particularly, it is a device driver that transfers GPU commands from the CPU to the GPU, regardless of whether they produce graphics or GPGPU workloads. Hence, the development of a robust GPU device driver is of significant impact for many GPU applications. Unfortunately, existing GPU device drivers [1, 5, 7, 19, 25] are not tailored to support real-time multi-tasking environments, but accelerate *one* particular high-performance application in the system or provide *fairness* among applications.

We have conducted a preliminary evaluation to see the performance of existing GPU drivers, (i) the NVIDIA proprietary driver [19] and (ii) the Nouveau open-source

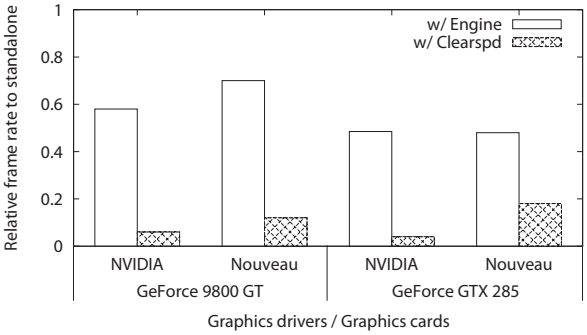


Figure 1: Decrease in performance of the OpenArena application competing with different GPU applications.

driver [7], in multi-tasking environments, using two different NVIDIA graphics cards, (i) GeForce 9800 GT and (ii) GeForce GTX 285, where the Linux 2.6.35 kernel is used as the underlying operating system. It should be noted that this NVIDIA driver evaluated on Linux is also expected to closely match the performance of the Windows driver (WDDM [25]), as they share about 90% of code [23]. Figure 1 shows the relative decrease in performance (frame-rate) of an OpenGL game (*OpenArena*) competing with two GPU-accelerated programs (*Engine* and *Clearspd* [15]) respectively. The *Engine* program represents a regularly-behaved GPU workload, while the *Clearspd* program produces a GPU command *bomb* causing the GPU to be overloaded, which represents a malicious or buggy program. To achieve the best possible performance, this preliminary evaluation assigns the highest CPU (`nice`) priority to the *OpenArena* application as an important application. As observed in Figure 1, the performance of the important *OpenArena* application drops significantly due to the existence of competing GPU applications. It highlights the fact that GPU resource management in the current state of the art is woefully inadequate, lacking prioritization and isolation capabilities for multiple GPU applications.

Contributions: We propose, design, and implement *TimeGraph*, a GPU scheduler to provide prioritization and isolation capabilities for GPU applications in *soft* real-time multi-tasking environments. We address a core challenge for GPU resource management posed due to the asynchronous and non-preemptive nature of GPU processing. Specifically, TimeGraph adopts an *event-driven* scheduler model that synchronizes the GPU with the CPU in a responsive manner, using GPU-to-CPU interrupts, to schedule non-preemptive GPU commands for the asynchronously-operating GPU. Under this event-driven model, TimeGraph supports two scheduling policies to *prioritize* tasks on the GPU, which address the trade-off between response times and throughput. TimeGraph also employs two resource reservation policies to *isolate* tasks on the GPU, which provide different levels of quality of service (QoS) at the expense of different levels of overhead. To the best of our knowledge, this is the first work that enables GPU applications to be prioritized and isolated in real-time multi-tasking environments.

Organization: The rest of this paper is organized as follows. Section 2 introduces our system model, including the scope and limitations of TimeGraph. Section 3 provides the system architecture of TimeGraph. Section 4 and Section 5 describe the design and implementation of TimeGraph GPU scheduling and reservation mechanisms respectively. In Section 6, the performance of TimeGraph is evaluated, and its capabilities are demonstrated. Related work is discussed in Section 7. Our concluding remarks are provided in Section 8.

2 System Model

Scope and Limitations: We assume a system composed of a generic multi-core CPU and an on-board GPU. We do not manipulate any GPU-internal units, and hence GPU commands are not preempted once they are submitted to the GPU. TimeGraph is independent of libraries, compilers, and runtime engines. The principles of TimeGraph are therefore applicable for different GPU architectures (e.g., NVIDIA Fermi/Tesla and ATI Stream) and programming frameworks (e.g., OpenGL, OpenCL, CUDA, and HMPP). Currently, TimeGraph is designed and implemented for Nouveau [7] available in the Gallium3D [15] OpenGL software stack, which is also planned to support OpenCL. Moreover, TimeGraph has been ported to the PSCNV open-source driver [22] packaged in the PathScale ENZO suite [21], which supports CUDA and HMPP. This paper is, however, focused on OpenGL workloads, given the currently-available set of open-source solutions: Nouveau and Gallium3D.

Driver Model: TimeGraph is part of the device driver, which is an interface for user-space programs to submit GPU commands to the GPU. We assume that the device driver is designed based on the *Direct Rendering Infrastructure* (*DRI*) [14] model that is adopted in most UNIX-like operating systems, as part of the X Window System. Under the DRI model, user-space programs are allowed to access the GPU directly to render frames without using windowing protocols, while they still use the windowing server to blit the rendered frames to the screen. GPGPU frameworks require no such windowing procedures, and hence their model is more simplified.

In order to submit GPU commands to the GPU, user-space programs must be allocated GPU *channels*, which conceptually represent separate address spaces on the GPU. For instance, the NVIDIA Fermi and Tesla architectures support 128 channels. Our GPU command submission model for each channel is shown in Figure 2.

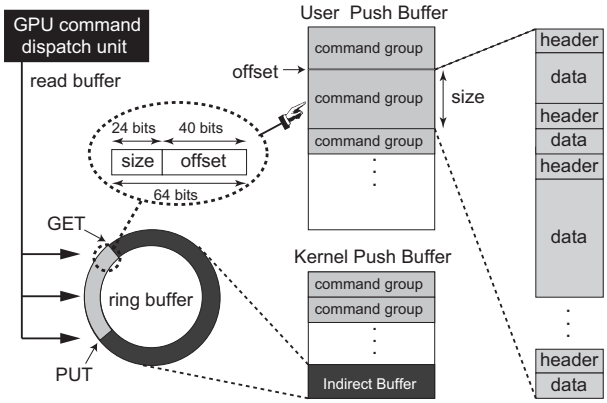


Figure 2: GPU command submission model.

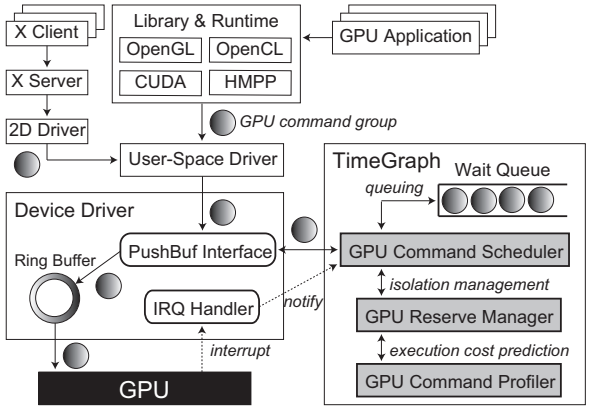


Figure 3: TimeGraph system architecture.

Each channel uses two types of kernel-space buffers: *User Push Buffer* and *Kernel Push Buffer*. The User Push Buffer is mapped on to the address space of the corresponding task, where GPU commands are pushed from the user space. GPU commands are usually *grouped* as *non-preemptive* regions to match user-space atomicity assumptions. The Kernel Push Buffer, meanwhile, is used for kernel primitives, such as host-device synchronization, GPU initialization, and GPU mode setting.

While user-space programs push GPU commands into the User Push Buffer, they also write *packets*, each of which is a (*size* and *address*) tuple to locate a certain GPU command group, into a specific ring buffer part of the Kernel Push Buffer, called *Indirect Buffer*. The driver configures the command dispatch unit on the GPU to read the buffer for command submission. This ring buffer is controlled by `GET` and `PUT` pointers. The pointers start from the same place. Every time packets are written to the buffer, the driver moves the `PUT` pointer to the tail of the packets, and sends a signal to the GPU command dispatch unit to download the GPU command groups located by the packets between the `GET` and `PUT` pointers. The `GET` pointer is then automatically updated to the same place as the `PUT` pointer. Once these GPU command groups are submitted to the GPU, the driver does not manage them any longer, and continues to submit the next set of GPU command groups, if any. Thus, this Indirect Buffer plays a role of a command queue.

Each GPU command group may include multiple GPU commands. Each GPU command is composed of the header and data. The header contains *methods* and the data size, while the data contains the values being passed to the methods. Methods represent GPU instructions, some of which are shared between compute and graphics, and others are specific for each. We assume that the device driver does not preempt on-the-fly GPU command groups, once they are offloaded on to the GPU. GPU command execution is out-of-order within the same GPU channel. The GPU channels are switched automatically by the GPU engines.

Our driver model described above is based on *Direct Rendering Manager* (DRM) [6], and especially target the NVIDIA Fermi and Tesla architectures, but can also be used for other architectures with minor modification.

3 TimeGraph System Architecture

The architecture of TimeGraph and its interaction with the rest of the software stack is illustrated in Figure 3. No modification is required for user-space programs, and GPU command groups can be generated through existing software frameworks. However, TimeGraph needs to communicate with a specific interface, called PushBuf, in the device driver space. The PushBuf interface enables the user space to submit GPU command groups stored in the User Push Buffer. TimeGraph uses this PushBuf interface to queue GPU command groups. It also uses the IRQ handler prepared for GPU-to-CPU interrupts to dispatch the next available GPU command groups.

TimeGraph is composed of *GPU command scheduler*, *GPU reserve manager*, and *GPU command profiler*. The GPU command scheduler queues and dispatches GPU command groups based on task priorities. It also coordinates with the GPU reserve manager to account and enforce GPU execution times of tasks. The GPU command profiler supports prediction of GPU command execution costs to avoid overruns out of reservation. There are two scheduling policies supported to address the trade-off between response times and throughput:

- Predictable-Response-Time (PRT): This policy minimizes priority inversion on the GPU to provide predictable response times based on priorities.
- High-Throughput (HT): This policy increases total throughput, allowing additional priority inversion.

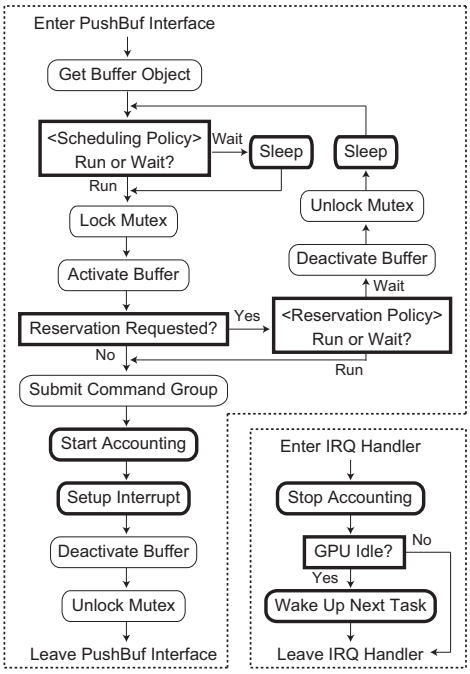


Figure 4: Diagram of the PushBuf interface and the IRQ handler with the TimeGraph scheme.

It also supports two GPU reservation policies that address the trade-off between isolation and throughput:

- Posterior Enforcement (PE): This policy enforces GPU resource usage after GPU command groups are completed without sacrificing throughput.
- Apriori Enforcement (AE): This policy enforces GPU resource usage before GPU command groups are submitted using prediction of GPU execution costs at the expense of additional overhead.

In order to unify multiple tasks into a single reserve, the TimeGraph reservation mechanism provides the Shared reservation mode. Particularly, TimeGraph creates a special Shared reserve instance with the PE policy when loaded, called Background, which serves all GPU-accelerated tasks that do not belong to any specific reserves. The detailed design and implementation for GPU scheduling and GPU reservation will be described in Section 4 and Section 5 respectively.

Figure 4 shows a high-level diagram of the PushBuf interface and the IRQ handler, where modifications introduced by TimeGraph are highlighted by bold frames. This diagram is based on the Nouveau implementation, but most GPU drivers should have similar control flows.

The PushBuf interface first acquires the buffer object associated with the incoming GPU command group. It then applies the scheduling policy to determine whether this GPU command group can execute on the GPU. If it should not be dispatched immediately, the corresponding task goes to sleep. Else, the User Push Buffer object is activated for command submission with the mutex lock to ensure the GPU command group to be located in the place accessible from the GPU, though the necessity of this procedure depends on driver implementation. TimeGraph next checks if GPU reservation is requested for this task. If so, it applies the reservation policy to verify the GPU resource usage of this task. If it overruns, TimeGraph winds up buffer activation, and suspends this task until its resource budget becomes available. This task will be rescheduled later when it is waken up, since some higher-priority tasks may arrive by then. Finally, if the GPU command group is qualified by the scheduling and reservation policies, it is submitted to the GPU. As the reservation policies need to track GPU resource usage, TimeGraph starts accounting for the GPU execution time of this task. It then configures the GPU command group to generate an interrupt to the CPU upon completion so that TimeGraph can dispatch the next GPU command group. After deactivating the buffer and unlocking the mutex, the PushBuf interface returns.

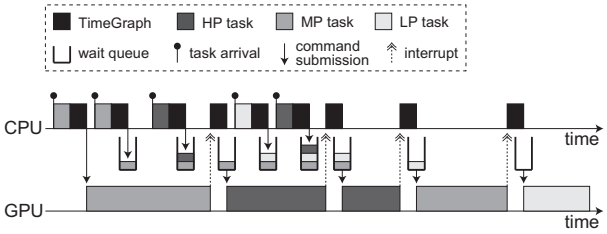
The IRQ handler receives an interrupt notifying the completion of the current GPU command group, where TimeGraph stops accounting for the GPU execution time, and wakes up the next task to execute on the GPU based on the scheduling policy, if the GPU is idle.

Specification: System designers may use a *specification* primitive to activate the TimeGraph functionality, which is inspired by the Redline system [31]. For each application, system designers can specify the scheduling parameters as: `<name:sched:resv:prio:C:T>`, where `name` is the application name, `sched` is its scheduling policy, `resv` is its reservation policy, `prio` is its priority, and a set of `C` and `T` represents that the application task is allowed to execute on the GPU for `C` microseconds every `T` microseconds. The specification is a text file (`/etc/timegraph.spec`), and TimeGraph reads it every time a new GPU channel is allocated to a task. If there is a matching entry based on the application name associated with the task, the specification is applied to the task. Otherwise, the task is assigned the lowest GPU priority and the Background reserve.

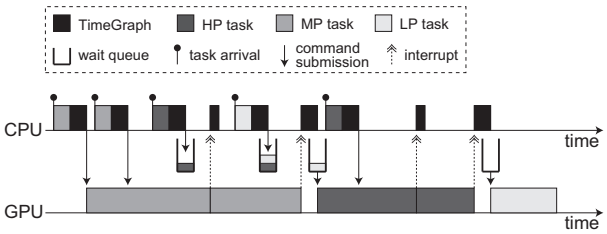
Priority Assignment: While system designers may assign static GPU priorities in their specification, TimeGraph also supports automatic GPU priority assignment (AGPA), which is enabled by using a wild-card "`*`" entry in the `prio` field. TimeGraph provides a user-space daemon executing periodically to identify the task with the foreground window through a window programming interface, such as the `_NET_ACTIVE_WINDOW` and the `_NET_WM_PID` properties in the X Window System. TimeGraph receives the foreground task information via a system call, and assigns the highest priority to this

task among those running under the AGPA mechanism. These tasks execute at the *default* static GPU priority level. Hence, different tasks can be prioritized over them by assigning higher static GPU priorities. AGPA is, however, not available if the above window programming interface is not supported. TimeGraph instead provides another user-space tool for system designers to assign priorities. For instance, designers can provide an optimal priority assignment based on reserve periods [13], as widely adopted in real-time systems.

Admission Control: In order to achieve predictable services in overloaded situations, TimeGraph provides an admission control scheme that forces the new reserve to be a background reserve so that currently active reserves continue to execute in a predictable manner. TimeGraph provides a simple interface where designers specify the limit of total GPU resource usage by 0-100% in a text file (/etc/timegraph.ac). The amount of limit is computed by a traditional resource-reservation model based on C and T of each reserve [26].



(a) PRT scheduling.



(b) HT scheduling.

Figure 5: Example of GPU scheduling in TimeGraph.

4 TimeGraph GPU Scheduling

The goal of the GPU command scheduler is to *queue* and *dispatch* non-preemptive GPU command groups in accordance with task priorities. To this end, TimeGraph contains a *wait queue* to stall tasks. It also manages a *GPU-online list*, a list of pointers to the GPU command groups currently executing on the GPU.

The GPU-online list is used to check if there are currently-executing GPU command groups, when a GPU command group enters into the PushBuf interface. If the list is empty, the corresponding task is inserted into it, and the GPU command group is submitted to the GPU. Else, the task is inserted into the wait queue to be scheduled. The scheduling policies supported by TimeGraph will be presented in Section 4.1.

Management of the GPU-online list requires the information about when GPU command groups complete. TimeGraph adopts an event-driven model that uses GPU-to-CPU interrupts to notify the completion of each GPU command group, rather than a tick-driven model adopted in the previous work [1, 5]. Upon every interrupt, the corresponding GPU command group is removed from the GPU-online list. Our GPU-to-CPU interrupt setting and handling mechanisms will be described in Section 4.2.

4.1 Scheduling Policies

TimeGraph supports two GPU scheduling policies. The Predictable-Response-Time (PRT) policy encourages such tasks that should behave on a timely basis without affecting important tasks. This policy is predictable in a sense that GPU command groups are scheduled based on task priorities to make high-priority tasks responsive on the GPU. The High-Throughput (HT) policy, on the other hand, is suitable for such tasks that should execute as fast as possible. There is a trade-off that the PRT policy prevents tasks from interference at the expense of throughput, while the HT policy achieves high throughput for one task but may block others. For instance, desktop-widget, browser-plugin, and video-player tasks are desired to use the PRT policy, while 3-D game and interactive 3-D interfacing tasks can use the HT policy.

PRT Scheduling: The PRT policy forces any GPU command groups to wait for the completion of the preceding GPU command group, if any. Specifically, a new GPU command group arriving at the device driver can be submitted to the GPU immediately, if the GPU-online list is empty. Else, the corresponding task must sleep in the wait queue. The highest-priority task in the wait queue, if any, is waken up upon every interrupt from the GPU.

Figure 5 (a) indicates how three tasks with different priorities, high-priority, medium-priority (MP), and low-priority (LP), are scheduled on the GPU under the PRT policy. 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 this time, because the GPU is not idle, according to the PRT policy. Even the next HP task is also queued due to the same reason, since further higher-priority tasks may arrive soon. The specific set of GPU commands appended at the end of every GPU command group by TimeGraph generates an interrupt to the CPU, and the TimeGraph scheduler is invoked ac-

cordingly to wake up the highest-priority task in the wait queue. Hence, the HP task is next chosen to execute on the GPU rather than the MP task. 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.

Given that the arrival times of GPU command groups are not known a priori, and each GPU command group is non-preemptive, we believe that the PRT policy is the best possible approach to provide predictable response times. However, it inevitably incurs overhead to make a scheduling decision at every GPU command group boundary, as shown in Figure 5 (a).

HT Scheduling: The HT policy reduces this scheduling overhead, compromising predictable response times a bit. It allows GPU command groups to be submitted to the GPU immediately, if (i) the currently-executing GPU command group was submitted by the same task, and (ii) no higher-priority tasks are ready in the wait queue. Otherwise, they must suspend in the same manner as the PRT policy. Upon an interrupt, the highest-priority task in the wait queue is waken up, *only when* the GPU-online list is empty (the GPU is idle).

Figure 5 (b) depicts how the same set of GPU command groups used in Figure 5 (a) is scheduled under the HT policy. Unlike the PRT policy, the second instance of the MP task can submit its GPU command group immediately, because the currently-executing GPU command group was issued by itself. These two GPU command groups of the MP task can execute successively without producing the idle time. The same is true for the two GPU command groups of the HP task. Thus, the HT policy is more for throughput-oriented tasks, but the HP task is blocked by the MP task for a longer internal. This is a trade-off, and if priority inversion is critical, the PRT policy is more appropriate.

4.2 Interrupt Setting and Handling

In order to provide an event-driven model, TimeGraph configures the GPU to generate an interrupt to the CPU upon the completion of each GPU command group. The scheduling point is thus made at every GPU command group boundary. We now describe how the interrupt is generated. For simplicity of description, we here focus on the NVIDIA GPU architecture.

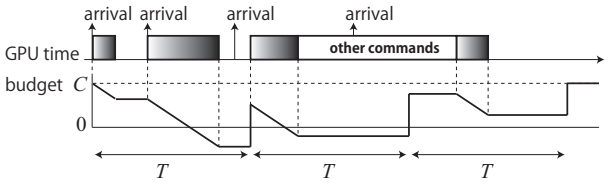
Completion Notifier: The NVIDIA GPU provides the `NOTIFY` command to generate an interrupt from the GPU to the CPU. TimeGraph puts this command at the end of each GPU command group. However, the interrupt is not launched immediately when the `NOTIFY` command is operated but when the next command is dispatched. TimeGraph therefore adds the `NOP` command after the `NOTIFY` command, as a dummy command. We also need to consider that GPU commands execute out of order on the GPU. If the `NOTIFY` command is operated before all commands in the original GPU command group are operated, the generated interrupt is not timely at all. TimeGraph hence adds the `SERIALIZE` command right before the `NOTIFY` command, which forces the GPU to stall until all on-the-fly commands complete. There is no need to add another piece of the `SERIALIZE` command after the `NOTIFY` command, since we know that no tasks other than the current task can use the GPU until TimeGraph is invoked upon the interrupt.

Interrupt Association: All interrupts from the GPU caught in the IRQ handler are relayed to TimeGraph. When TimeGraph receives an interrupt, it first references the head of the GPU-online list to obtain the task information associated with the corresponding GPU command group. TimeGraph next needs to verify whether this interrupt is truly generated by the commands that TimeGraph inserted into at the end of the GPU command group, given that user-space programs may also use the `NOTIFY` command. In order to recognize the right interrupt, TimeGraph further adds the `SET_REF` command before the `SERIALIZE` command, which instructs the GPU to write a specified sequence number to a particular GPU register. This number is identical for each task, and is simply incremented by TimeGraph. TimeGraph reads this GPU register when an interrupt is received. If the register value is less than the expected sequence number associated with the corresponding GPU command group, this interrupt should be ignored, since it must have been caused by someone else before the `SET_REF` command. Another piece of the `SERIALIZE` command also needs to be added before the `SET_REF` command to ensure in-order command execution. As a consequence, TimeGraph inserts the following commands at the end of each GPU command group: `SERIALIZE`, `SET_REF`, `SERIALIZE`, `NOTIFY`, `NOP`.

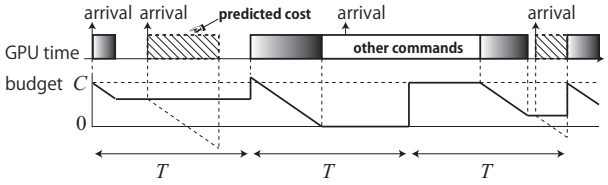
Task Wake-Up: Once the interrupt is verified, TimeGraph removes the GPU command group at the head of the GPU-online list. If the corresponding task is scheduled under the PRT policy, TimeGraph wakes up the highest-priority task in the wait queue, and inserts its GPU command group into the GPU-online list. If the task is assigned the HT policy, meanwhile, TimeGraph wakes up the highest-priority task in the same manner as the PRT policy, *only when* the GPU-online list is empty.

5 TimeGraph GPU Reservation

TimeGraph provides GPU reservation mechanisms to regulate GPU resource usage for tasks scheduled under the PRT policy. Each task is assigned a *reserve* that is represented by capacity C and period T. Budget e is the amount of time that a task is entitled for execution. TimeGraph uses a popular rule for budget consumption and



(a) PE reservation.



(b) AE reservation.

Figure 6: Example of GPU reservation in TimeGraph.

replenishment used in real-time systems [20, 26]. Specifically, the budget is decreased by the amount of time consumed on the GPU, and is replenished by at most capacity C once every period T. However, we need different reservation policies than previous work due to the asynchronous and non-preemptive nature of GPU processing, as we will describe in Section 5.1. Our GPU resource accounting and enforcement mechanisms will be described in Section 5.2. TimeGraph further supports prediction of GPU execution costs for strict isolation. Section 5.3 will describe our approach to GPU execution cost prediction.

5.1 Reservation Policies

TimeGraph supports two GPU reservation policies. The Posterior Enforcement (PE) policy is aimed for lightweight reservation, allowing tasks to overrun out of their reserves to an extent. The Apriori Enforcement (AE) policy reduces reserve overruns by predicting GPU execution costs a priori at the expense of additional overhead. We recommend that the PE policy be primarily used when isolation is required, and the AE policy be used only if extremely time-critical applications are concurrently executed on the GPU.

PE Reservation: The PE policy permits GPU command groups to be submitted to the GPU, if their budget is greater than zero. Else, the task goes to sleep until the budget is replenished. The budget can be negative, when the task overruns out of reservation. The overrun penalty is, however, imposed on the next budget replenishment. The budget for the next period is therefore given by $e = \min(C, e + C)$.

Figure 6 (a) shows how four GPU command groups of the same task are enforced under the PE policy. The budget is initialized to C. When the second GPU command group completes, the budget is negative. Hence, the third GPU command group must wait for the budget to be replenished, even though the GPU remains idle. Since GPU reservation is available under the PRT policy, the fourth GPU command group is blocked even though the budget is greater than zero, since another GPU command group is currently executing.

AE Reservation: For each GPU command group submission, the AE policy first predicts a GPU execution cost x. The GPU command group can be submitted to the GPU, only if the predicted cost is no greater than the budget. Else, the task goes to sleep until the budget is replenished. The next replenishment amount depends on the predicted cost x and the currently-remaining budget e. If the predicted cost x is no greater than the capacity C, the budget for the next period is bounded by $e = C$ to avoid transient overload. Else, it is set to $e = \min\{x, e + C\}$. The task can be waken up only when $e \geq x$.

Figure 6 (b) depicts how the same set of four GPU command groups used in Figure 6 (a) is controlled under the AE policy. For simplicity of description, we assume for now that prediction of GPU execution costs is perfectly accurate, and Section 5.3 will describe how to practically predict GPU execution costs. Unlike the PE policy, the second GPU command group is not submitted to the GPU, as its budget is less than the predicted cost, but is submitted later when the budget is replenished to be $e = \min\{x, e + C\} > x$. The fourth GPU command group also needs to wait until the budget is sufficiently replenished. However, unlike the second GPU command group, the replenished budget is bounded by C, since $x < C$. This avoids transient overload.

Shared Reservation: TimeGraph allows multiple tasks to share a single reserve under the Shared mode. When some task creates a Shared reserve, other tasks can join it. The Shared mode can be used together with both the PE and AE policies. The Shared mode is useful when users want to cap the GPU resource usage of multiple tasks to a certain range. There is no need to adjust the capacity and period for each task. It can also reduce the overhead of reservation, since it only needs to manage one reserve for multiple tasks.

5.2 Accounting and Enforcement

GPU execution times are accounted in the PushBuf interface and the IRQ handler as illustrated in Figure 4. TimeGraph saves CPU timestamps when GPU command groups start and complete. Specifically, when each GPU command group is qualified to be submitted to the GPU, TimeGraph records the current CPU time as its *start time* in the PushBuf interface, and at some later point of time when TimeGraph is notified of the completion of this GPU command group, the current CPU time is recorded as its *finish time* in the IRQ handler. The difference be-

tween the start time and the finish time is accounted for as the execution time of this GPU command group, and is subtracted from the budget.

Enforcement works differently for the PE and the AE policies. In the PushBuf interface, the AE policy predicts the execution cost x of each GPU command group based on the idea presented in Section 5.3, while the PE policy always assumes $x = 0$. Then, both policies compare the budget e and the cost x. Only if $e > x$ is satisfied, the GPU command group can be submitted to the GPU. Otherwise, the corresponding task is suspended until the budget is replenished. It should be noted that this enforcement mechanism is very different from traditional CPU reservation mechanisms [20, 26] that use timers or ticks to suspend tasks, since GPU command groups are non-preemptive, and hence we need to perform enforcement at GPU command group boundary. TimeGraph however still uses timers to replenish the budget periodically. Every time the budget is replenished, it compares e and x again. If $e > x$ is satisfied, the task is waken up, but it needs to be rescheduled, as illustrated in Figure 4.

5.3 Command Profiling

TimeGraph contains the GPU command profiler to predict GPU execution costs for AE reservation. Each GPU command is composed of the header and data, as shown in Figure 2. We hence parse the methods and the data sizes from the headers.

We now explain how to predict GPU execution costs from these pieces of information. GPU applications tend to repeatedly create GPU command groups with the same methods and data sizes, since they use the same set of API functions, e.g., OpenGL, and each function likely generates the same sequence of GPU commands in terms of methods and data sizes, while data values are quite variant. Given that GPU execution costs depend highly on methods and data sizes, but not on data values, we propose a history-based prediction approach.

TimeGraph manages a history table to record the GPU command group information. Each record consists of a *GPU command group matrix* and the average GPU execution cost associated to this matrix. The row and the column of the matrix contain the methods and their data sizes respectively. TimeGraph also attaches a flag to each GPU command group, indicating if it hits some record. When the methods and the data sizes of the GPU command group are obtained from the remapped User Push Buffer, TimeGraph looks at the history table. If there exists a record that contains exactly the same GPU command group matrix, i.e., the same set of methods and data sizes, it uses the average GPU execution cost stored in this record, and the flag is set. Otherwise, the flag is cleared, and TimeGraph uses the worst-case GPU execution cost among all the records. Upon the completion of the GPU command group, TimeGraph references the flag attached to the corresponding task. If the flag is set, it updates the average GPU execution cost of the record with the actual execution time of this GPU command group. Otherwise, it inserts a new record where the matrix has the methods and the data size of this GPU command group, and the average GPU execution time is initialized with its actual execution time. The size of the history table is configurable by designers. If the total number of the records exceeds the table size, the least-recently-used (LRU) record is removed.

Preemption Impact: Even the same GPU command group may consume very different GPU execution times. For example, if reusable texture data is cached, graphics operation is much faster. We realize that when the GPU contexts (channels) are switched, GPU execution times can vary. Hence, TimeGraph verifies GPU context switches at every scheduling point. If the context is switched, TimeGraph will not update the average GPU execution cost, since the context switch may have affected the actual GPU execution time. Instead, it saves the difference between the actual GPU execution time and the average GPU execution cost as the *preemption impact*. TimeGraph keeps updating the average preemption impact. A single preemption cost is measured beforehand when TimeGraph is loaded. The preemption impact is then added to the predicted cost.

6 Evaluation

We now provide a detailed quantitative evaluation of TimeGraph on the NVIDIA GeForce 9800 GT graphics card with the default frequency and 1 GB of video memory. Our underlying platform is the Linux 2.6.35 kernel running on the Intel Xeon E5504 CPU and 4 GB of main memory. While our evaluation and discussion are focused on this graphics card, Similar performance benefits from TimeGraph have also been observed with different graphics cards viz, GeForce GTX 285 and GTX 480.

As primary 3-D graphics benchmarks, we use the Phoronix Test Suite [24] that executes the OpenGL 3-D games, *OpenArena*, *World of Padman*, *Urban Terror*, and *Unreal Tournament 2004 (UT2004)*, in the demo mode based on the test profile, producing various GPU-intensive workloads. We also use *MPlayer* as a periodic workload. In addition, the Gallium3D *Engine* demo program is used as a regularly-behaved workload, and the Gallium3D *Clearspd* demo program that exploits a GPU command *bomb* is used as a misbehaving workload. Furthermore, we use *SPECviewperf 11* [28] to evaluate the throughput of different GPU scheduling models. The screen resolution is set to 1280×1024. The scheduling parameters are loaded from the pre-

configured TimeGraph specification file. The maximum number of records in the history table for GPU execution cost prediction is set to 100.

6.1 Prioritization and Isolation

We first evaluate the prioritization and isolation properties achieved by TimeGraph. As described in Section 3, TimeGraph automatically assigns priorities. CPU `nice` priorities are always effective, while GPU priorities are effective only when TimeGraph is activated. The priority level is aligned between the GPU and CPU. We use the PRT policy for the X server to prevent it from affecting primary applications, but it is scheduled by the highest GPU/CPU priority, since it should still be responsive to blit the rendered frames to the screen.

Coarse-grained Performance: Figure 7 shows the performance of the 3-D games, while the Engine widget is concurrently sharing the GPU. We use the HT policy for the 3-D games, while the Engine widget is assigned the PRT policy under TimeGraph. As shown in Figure 7, TimeGraph improves the performance of the 3-D games by about 11% for OpenArena, 27% for World of Padman, 22% for Urban Terror, and 2% for UT2004, with GPU priority support. Further performance isolation is obtained by GPU reservation, capping the GPU resource usage of the Engine widget. Our experiment assigns the Engine widget a reserve of $2.5ms$ every $25ms$ to retain GPU resource usage at 10%. As compared to the case without GPU reservation support, the performance of the 3-D games is improved by $2 \sim 21\%$ under PE reservation, and by $4 \sim 36\%$ under AE reservation. Thus, the AE policy provides better performance for the 3-D games at the expense of more conservative scheduling of the Engine widget with prediction.

Figure 8 presents the results from a setup similar to the above experiments, where the Clearspd bomb generates heavily-competing workload instead of the Engine widget. The performance benefit resulting from assigning higher GPU priorities to the games under the HT policy is clearer in this setup. Even without GPU reservation support, TimeGraph enables the 3-D games to run about $3 \sim 6$ times faster than the vanilla Nouveau driver, though they still face a performance loss of about $24 \sim 52\%$ as compared to the previous setup where the Engine widget contends with the 3-D games. Regulating the GPU resource usage of the Clearspd bomb through GPU reservation limits this performance loss to be within 3%. Particularly, the AE policy yields improvements of up to 5% over the PE policy.

Extreme Workloads: In order to evaluate the capabilities of TimeGraph in the face of extreme workloads, we execute the 3-D games with five instances of the Clearspd bomb. In this case, the cap of each individual reserve is correspondingly decreased to $0.5ms$ every $25ms$ so that the total cap of the five Clearspd-bomb tasks is aligned with $2.5ms$ every $25ms$. As here are multiple Clearspd-bomb tasks, we evaluate an additional setup where a single PE reserve of $2.5ms$ every $25ms$ runs with the Shared reservation mode. As shown in Figure 9, the 3-D games are nearly unresponsive without TimeGraph support due to the scaled-up GPU workload, whereas TimeGraph can isolate the performance of the 3-D games even under such an extreme circumstance. In fact, the performance impact is reduced to $7 \sim 20\%$ by using GPU priorities, and leveraging GPU reservation re-

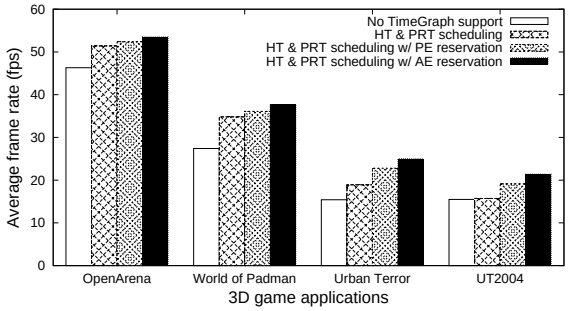


Figure 7: Performance of the 3-D games competing with a single instance of the Engine widget.

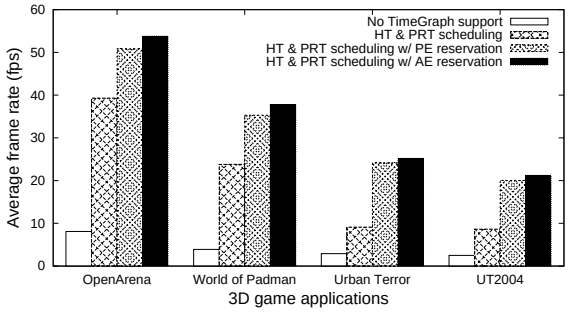


Figure 8: Performance of the 3-D games competing with a single instance of the Clearspd bomb.

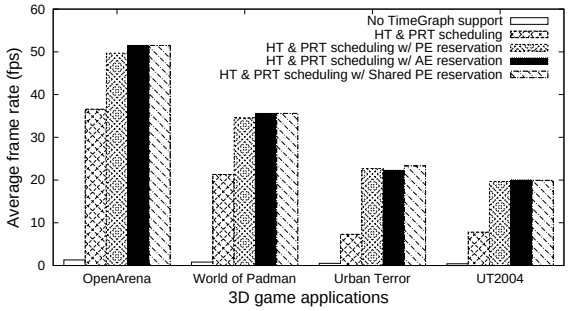


Figure 9: Performance of the 3-D games competing with five instances of the Clearspd bomb.

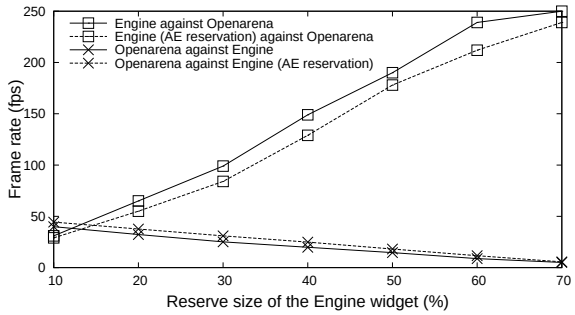


Figure 10: Performance regulation by GPU reservation for the 3-D game and the 3-D widget.

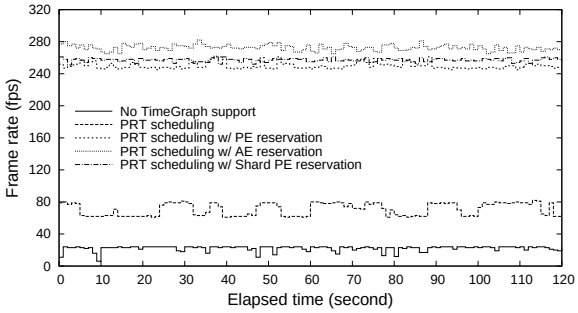
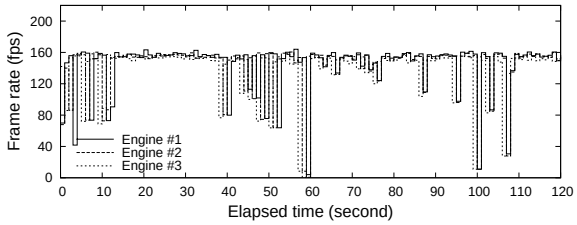
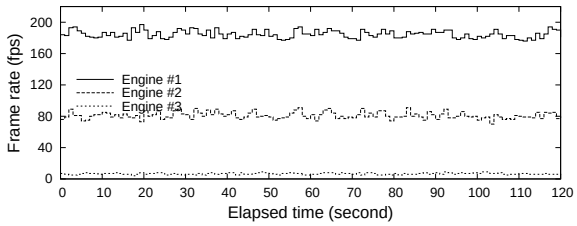


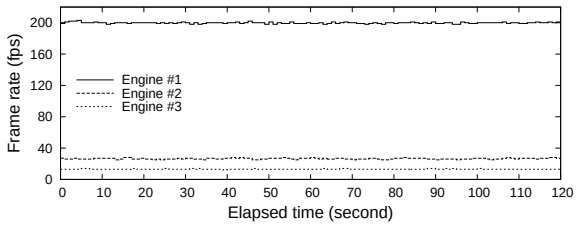
Figure 11: Performance of the Engine widget competing with five instances of the Clearspd bomb.



(a) No TimeGraph support.



(b) PRT scheduling.



(c) PRT scheduling and PE reservation.

Figure 12: Interference among three widget instances.

sults in nearly no performance loss, similar to results in Figure 8. The Shared reservation mode also provides slightly better performance with PE reserves.

Performance Regulation: We next demonstrate the effectiveness of TimeGraph in regulating the frame-rate for each task by changing the size of GPU reserve. Figure 10 shows the performance of the OpenArena game and the Engine widget contending with each other. The solid lines indicate a setup where the PE policy is assigned for both the applications, while the dotted lines indicate a setup where the AE policy is assigned for the Engine widget instead. GPU reservation is configured so that the total GPU resource usage of the two applications is capped at 90%, and the remaining 10% is available for the X server. Assigning the AE policy for the Engine widget slightly improves the performance of the OpenArena game, while it brings a performance penalty for the Engine widget itself due to the overhead for prediction of GPU execution costs. In either case, however, TimeGraph successfully regulates the frame-rate in accordance with the size of GPU reserve. In this experiment, we conclude that it is desirable to assign a GPU reserve for the OpenArena game with $C/T = 60 \sim 80\%$ and that for the Engine widget with $C/T = 10 \sim 30\%$, given that this configuration provides both the applications with an acceptable frame-rate over 25 fps.

Fine-grained Performance: The 3-D games demonstrate highly variable frame-rate workloads, while 3-D widgets often exhibit nearly constant frame-rates. In order to study the behavior of TimeGraph on both these two categories of applications, we look at the variability of frame-rate with time for the Engine widget contending with five instances of the Clearspd bomb, as shown in Figure 11. The total GPU resource usage of the Clearspd-bomb tasks is capped at $2.5ms$ every $25ms$ through GPU reservation, and a higher priority is given to the Engine widget. These results show that GPU reservation can provide stable frame-rates on a time for the Engine widget. Since the Engine widget is not as GPU-intensive as the 3-D games, it is affected more by the Clearspd bomb making the GPU overloaded, when GPU reservation is not applied. The benefits of GPU reservation are therefore more clearly observed.

Interference Issues: We now evaluate the interference among regularly-behaved concurrent 3-D widgets. Figure 12 (a) shows a chaotic behavior arising from executing three instances of the Engine widget concurrently, with different CPU priorities but without TimeGraph support. Although the Engine widget by itself is a very regular workload, when competing with more instances of itself, the GPU resource usage exhibits high variabil-

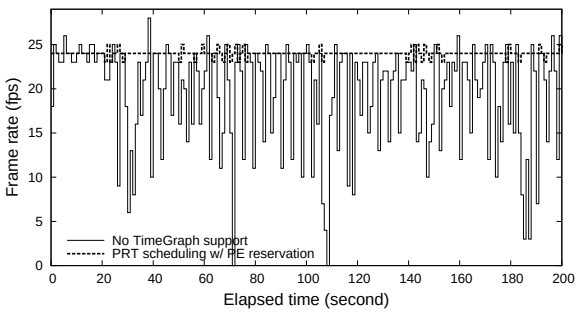


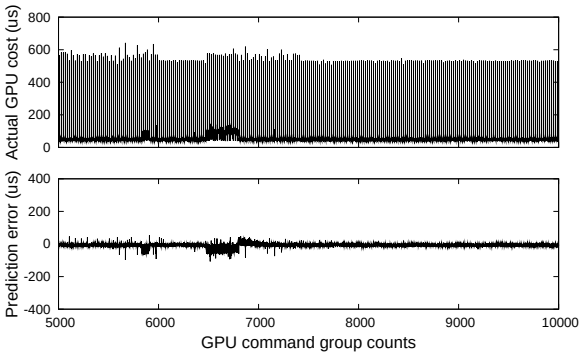
Figure 13: Performance of MPlayer competing with five instances of the Clearspd bomb.

ity and unpredictability. Figure 12 (b) illustrates the improved behavior under TimeGraph using the PRT policy, where we assign the high, the medium, and the low GPU priorities for *Engine #1*, *Engine #2*, and *Engine #3* respectively, using the user-space tool presented in Section 3. TimeGraph successfully provides predictable response times for the three tasks according based on their priorities. Further performance isolation can be achieved by GPU reservation, exploiting different sizes of GPU reserves: (i) $15ms$ every $25ms$ to *Engine #1*, (ii) $5ms$ every $50ms$ to *Engine #2*, and (iii) $5ms$ every $100ms$ to *Engine #3*, as shown in Figure 12 (c). The PE policy is used here. Since the Engine widget has a non-trivial dependence on the CPU, the absolute performance is lower than expected for smaller reserves.

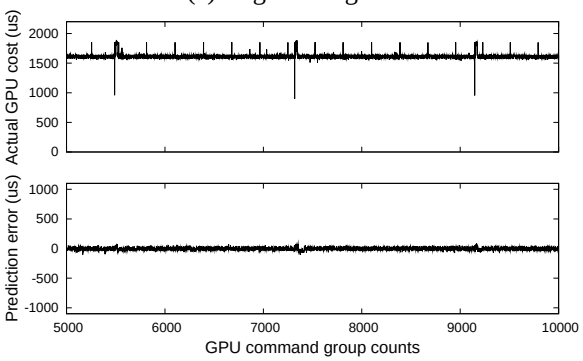
Periodic Behavior: For evaluating the impact on applications with periodic activity, we execute MPlayer in the foreground when five instances of the Clearspd bomb contend for the GPU. We use an H264-compressed video, with a frame size of 1920×800 and a frame rate of 24 fps, which uses x-video acceleration on the GPU. As shown in Figure 13, the video playback experience is significantly disturbed without TimeGraph support. When TimeGraph assigns a PE reserve of $10ms$ every $40ms$ for MPlayer, and a PE reserve of $5ms$ every $40ms$ for the Clearspd bomb tasks in the Shared reservation mode, the playback experience is significantly improved. It closely follows the ideal frame-rate of 24 fps for video playback. This illustrates the benefits of TimeGraph for interactivity, where performance isolation plays a vital role in determining user experience.

6.2 GPU Execution Cost Prediction

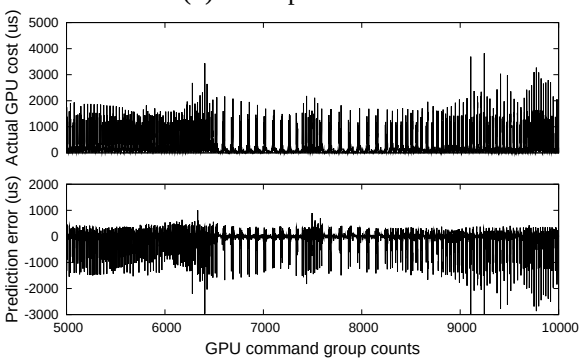
We now evaluate the history-based prediction of GPU execution costs for realizing GPU reservation with the AE policy. The effectiveness of AE reservation relies highly on GPU execution cost prediction. Hence, it is important to identify the types of applications for which we



(a) Engine widget.



(b) Clearspd bomb.



(c) OpenArena game.

Figure 14: Errors for GPU execution cost prediction.

can predict GPU execution costs more precisely. Figure 14 shows both actual GPU execution costs and prediction errors for the 3-D graphics applications used in earlier experiments: Engine, Clearspd, and OpenArena. Since these applications issue a very large number of GPU command groups, we focus on a snapshot between the 5000th and the 10000th GPU command groups of the applications. In the following discussion, positive errors denote pessimistic predictions. Figure 14 (a) shows that the history-based prediction approach employed by TimeGraph performs within a 15% error margin on the Engine widget that uses a reasonable number of methods. For the Clearspd bomb that issues a very limited set of methods, while producing extreme workloads, Time-

Graph can predict GPU execution costs within a 7% error margin as shown in Figure 14 (b). On the other hand, the results of GPU execution cost prediction under OpenArena, provided in Figure 14 (c), show that only about 65% of the observed GPU command groups have the predicted GPU execution costs within a 20% error margin. Such unpredictability arises from the inherently dynamic nature of complex computer graphics like abrupt scene changes. The actual penalty of misprediction is, however, suffered only once per reserve period, and is hence not expected to be significant for reserves with long periods.

In addition to the presented method, we have also explored static approaches using pre-configured values for predicted costs. Our experiments show that such static approaches perform worse than the presented dynamic approach, largely due to the dynamically changing and non-stationary nature of application workloads.

GPU execution cost prediction plays a vital role in real-time setups, where it is unacceptable for low-priority tasks to even cause the slightest interference to high-priority tasks. As the above experimental results show that our prediction approach tends to fail for complex interactive applications like OpenArena. However, we expect the structure of real-time applications to be less dynamic and more regular like the Engine and Clearspd tasks. GPU reservation with the AE policy for complex applications like OpenArena would require support from the application program itself, since their behavior is not easily predictable from historic execution results. Otherwise, the PE policy is desired for low overhead.

6.3 Overhead and Throughput

In order to quantify the performance overhead imposed by TimeGraph, we measure the standalone performance of the 3-D game benchmarks. Figure 15 shows that assigning the HT policy for both the games and the X server incurs about 4% performance overhead for the games. This small overhead is attributed to the fact that TimeGraph is still invoked upon every arrival and completion of GPU command group. It is interesting to see that assigning the PRT policy for the X server increases the overhead for the games up to about 10%, even though the games use the HT policy. As the X server is used to blit the rendered frames to the screen, it can lower the frame-rate of the game, if it is blocked by the game itself. On the other hand, assigning the PRT policy for both the X server and the game adds a non-trivial overhead of about $17 \sim 28\%$ largely due to queuing and dispatching all GPU command groups. This overhead is introduced by queuing delays and scheduling overheads, as TimeGraph needs to be invoked for submission of each GPU command group. We however conjecture that such overhead cost is inevitable for GPU scheduling at the device-driver level to shield important GPU applications from performance interference. If there is no TimeGraph support, the performance of high-priority GPU applications could significantly decrease in the presence of competing GPU workloads (as shown in Figure 1), which affects the system performance more than the maximum scheduling overhead of 28% introduced by TimeGraph. As a consequence, TimeGraph is evidently beneficial in real-time multi-tasking environments.

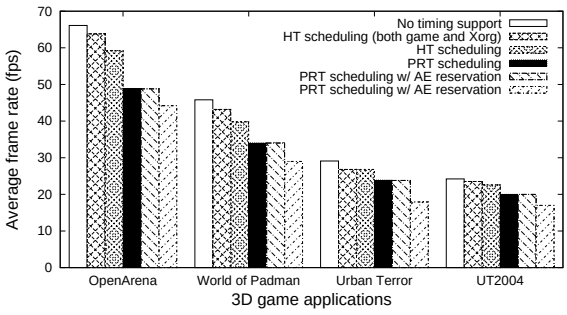


Figure 15: Performance overheads of TimeGraph.

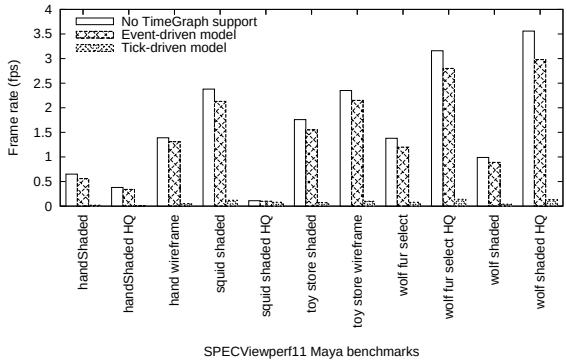


Figure 16: Throughput of event-driven and tick-driven schedulers in TimeGraph.

Finally, we compare the throughput of two conceivable GPU scheduling models: (i) the event-driven scheduler adopted in TimeGraph, and (ii) the tick-driven scheduler that was presented in the previous work, called GERM [1, 5]. The TimeGraph event-driven scheduler is invoked only when GPU command groups arrive and complete, while the GERM tick-driven scheduler is invoked at every tick, configured to be $1ms$ (Linux *jiffies*). Figure 16 shows the results of SPECviewperf benchmarking with the *Maya* viewset created for 3-D computer graphics, where the PRT policy is used for GPU scheduling. Surprisingly, the TimeGraph event-driven scheduler obtains about $15 \sim 30$ times better scores than the tick-driven scheduler for most test cases. According to our analysis, this difference arises from the fact that many

GPU command groups can arrive in a very short interval. The GERM tick-driven scheduler reads a particular GPU register every tick to verify if the current GPU command group has completed. Suppose that there are 30 GPU command groups with a total execution time of less than $1ms$. The tick-driven scheduler takes at least $30ms$ to compete these GPU command groups because the GPU register must be read 10 times, while the event-driven scheduler could complete them in $1ms$ as GPU-to-CPU interrupts are used. Hence, non-trivial overheads are imposed on the tick-driven scheduler.

7 Related Work

GPU Scheduling: The Graphics Engine Resource Manager (GERM) [1, 5] aims for GPU multi-tasking support similar to TimeGraph. The resource management concepts of TimeGraph and GERM are, however, fundamentally different. TimeGraph focuses on prioritization and isolation among competing GPU applications, while fairness is a primary concern for GERM. Since fair resource allocation cannot shield particular important tasks from interference in the face of extreme workloads, as reported in [31], TimeGraph addresses this problem for GPU applications through priority and reservation support. Approaches to synchronize the GPU with the CPU are also different between TimeGraph and GERM. TimeGraph is based on an event-driven model that uses GPU-to-CPU interrupts, whereas GERM adopts a tick-driven model that polls a particular GPU register. As demonstrated in Section 6.3, the tick-driven model can become unresponsive when many GPU commands arrive in a short interval, which could likely happen for graphics and compute-intensive workloads, while TimeGraph is responsive even in such cases. Hence, TimeGraph is more suitable for real-time applications. In addition, TimeGraph can *predict* GPU execution costs a priori, taking into account both methods and data sizes, while GERM *estimates* them posteriorly, using only data sizes. Since GPU execution costs are very dependent not only on data sizes but also on methods, we claim that TimeGraph computes GPU execution costs more precisely. However, additional computation overheads are required for prediction. TimeGraph therefore provides light-weight reservation with the PE policy without prediction to address this trade-off. Furthermore, TimeGraph falls inside the device driver, while GERM is spread across the device driver and user-space library. Hence, GERM could require major modifications for different runtime frameworks, e.g., OpenGL, OpenCL, CUDA, and HMPP.

The Windows Display Driver Model (WDDM) [25] is a GPU driver architecture for the Microsoft Windows. While it is proprietary, GPU priorities seem to be supported in our experience, but are not explicitly exposed to the user space as a first-class primitive. Apparently, there is no GPU reservation support. In fact, since NVIDIA shares more than 90% of code between Linux and Windows [23]. Therefore, it eventually suffers from the performance interference as demonstrated in Figure 1.

VMGL [11] supports virtualization in the OpenGL APIs for graphics applications running inside a Virtual Machine (VM). It passes graphics requests from guest OSes to a VMM host, but GPU resource management is left to the underlying device driver. The GPU-accelerated Virtual Machine (GViM) [8] virtualizes the GPU at the level of abstraction for GPGPU applications, such as the CUDA APIs. However, since the solution is 'above' the device driver layer, GPU resource management is coarse-grained and functionally limited. VMware's Virtual GPU [3] enables GPU virtualization at the I/O level. Hence, it operates faster and its usage is not limited to GPGPU applications. However, multi-tasking support with prioritization, isolation, or fairness is not supported. TimeGraph could coordinate with these GPU virtualization systems to provide predictable response times and isolation.

CPU Scheduling: TimeGraph shares the concept of priority and reservation, which has been well-studied by the real-time systems community [13, 26], but there is a fundamental difference from these traditional studies in that TimeGraph is designed to address an arbitrarily-arriving non-preemptive GPU execution model, whereas the real-time systems community has often considered a periodic preemptive CPU execution model. Several bandwidth-preserving approaches [12, 29, 30] for an arbitrarily-arriving model exist, but a non-preemptive model has not been much studied yet. The concept of priority and reservation has also been considered in the operating systems literature [4, 9, 10, 17, 20, 31]. Specifically, batch scheduling has a similar constraint to GPU scheduling in that non-preemptive regions disturb predictable responsiveness [27]. These previous work are, however, mainly focused on synchronous on-chip CPU architectures, whereas TimeGraph addresses those scheduling problems for asynchronous on-board GPU architectures where explicit synchronization between the GPU and CPU is required.

Disk Scheduling: Disk devices have similarity to GPUs in that they operate with non-preemptive regions off the chip. Disk scheduling for real-time and interactive systems [2, 16] therefore considered priority and reservation support for non-preemptive operation. However, the GPU is typically a coprocessor independent of the CPU, which has its own set of execution contexts, registers, and memory devices, while the disk is more dedicated to I/O. Hence, TimeGraph uses completely different mechanisms to realize prioritization and isolation than these previous work. In addition, TimeGraph needs to ad-

dress the trade-off between predictable response times and throughput since synchronizing the GPU and CPU incurs overhead, while disk I/O is originally synchronous with read and write operation.

8 Conclusions

This paper has presented TimeGraph, a GPU scheduler to support real-time multi-tasking environments. We developed the event-driven model to schedule GPU commands in a responsive manner. This model allowed us to propose two GPU scheduling policies, Predictable Response Time (PRT) and High Throughput (HT), which address the trade-off between response times and throughput. We also proposed two GPU reservation policies, Posterior Enforcement (PE) and the Apriori Enforcement (AE), which present an essential design knob for choosing the level of isolation and throughput. Our detailed evaluation demonstrated that TimeGraph can protect important GPU applications even in the face of extreme GPU workloads, while providing high-throughput, in real-time multi-tasking environments. TimeGraph is open-source software, and may be downloaded from our website at `http://rtml.ece.cmu.edu/projects/timegraph/`.

In future work, we will elaborate coordination of GPU and CPU resource management schemes to further consolidate prioritization and isolation capabilities for the entire system. We are also interested in coordination of video memory and system memory management schemes. Exploration of other models for GPU scheduling is another interesting direction of future work. For instance, modifying the current API to introduce non-blocking interfaces could improve throughput at the expense of modifications to legacy applications. Scheduling overhead and blocking time may also be reduced by implementing an real-time *satellite kernel* [18] on microcontrollers present in modern GPUs. Finally, we will tackle the problem of mapping application-level specifications, such as frame-rates, into priority and reservation properties at the operating-system level.

References

[1] BAUTIN, M., DWARAKINATH, A., AND CHIUEH, T. Graphics Engine Resource Management. In *Proc. MMCN* (2008).

[2] DIMITRIJEVIC, Z., RANGAWAMI, R., AND CHANG, E. Design and Implementation of Semi-preemptible IO. In *Proc. USENIX FAST* (2003).

[3] DOWTY, M., AND SUGEMAN, J. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 73–82.

[4] DUDA, K., AND CHERITON, D. Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler. In *Proc. ACM SOSP* (1999), pp. 261–276.

[5] DWARAKINATH, A. A Fair-Share Scheduler for the Graphics Processing Unit. Master's thesis, Stony Brook University, 2008.

[6] FAITH, R. *The Direct Rendering Manager: Kernel Support for the Direct Rendering Infrastructure*. Precision Insight, Inc., 1999.

[7] FREEDESKTOP. Nouveau Open-Source Driver. `http://nouveau.freedesktop.org/`.

[8] GUPTA, V., GAVRILOVSKA, A., TOLIA, N., AND TALWAR, V. GViM: GPU-accelerated Virtual Machines. In *Proc. ACM HPCVirt* (2009), pp. 17–24.

[9] JONES, M., ROSU, D., AND ROSU, M.-C. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proc. ACM SOSP* (1997), pp. 198–211.

[10] KRASIC, C., SAUBHASIK, M., AND GOEL, A. Fair and Timely Scheduling via Cooperative Polling. In *Proc. ACM EuroSys* (2009), pp. 103–116.

[11] LAGAR-CAVILLA, H., TOLIA, N., SATYANARAYANAN, M., AND DE LARA, E. VMM-Independent Graphics Acceleration. In *Proc. ACM VEE* (2007), pp. 33–43.

[12] LEHOCZKY, J., SHA, L., AND STROSNIDER, J. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *Proc. IEEE RTSS* (1987), pp. 261–270.

[13] LIU, C., AND LAYLAND, J. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM 20* (1973), 46–61.

[14] MARTIN, K., FAITH, R., OWEN, J., AND AKIN, A. *Direct Rendering Infrastructure, Low-Level Design Document*. Precision Insight, Inc., 1999.

[15] MESA3D. Gallium3D. `http://www.mesa3d.org/`.

[16] MOLANO, A., JUWA, K., AND RAJKUMAR, R. Real-Time Filesystems. Guaranteeing Timing Constraints for Disk Accesses in RT-Mach. In *Proc. IEEE RTSS* (1997), pp. 155–165.

[17] NIEH, J., AND LAM, M. SMART: A Processor Scheduler for Multimedia Applications. In *Proc. ACM SOSP* (1995).

[18] NIGHTINGALE, E., HODSON, O., MCLLORY, R., HAWBLITZEL, C., AND HUNT, G. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proc. ACM SOSP* (2009).

[19] NVIDIA CORPORATION. Proprietary Driver. `http://www.nvidia.com/page/drivers.html`.

[20] OIKAWA, S., AND RAJKUMAR, R. Portable RT: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior. In *Proc. IEEE RTAS* (1999), pp. 111–120.

[21] PATHSCALE INC. ENZO. `http://www.pathscale.com/`.

[22] PATHSCALE INC. PSCNV. `https://github.com/pathscale/pscnv`.

[23] PHORONIX. NVIDIA Developer Talks Openly About Linux Support. `http://www.phoronix.com/scan.php?page=article&item=nvidia_qa_linux&num=2`.

[24] PHORONIX. Phoronix Test Suite. `http://www.phoronix-test-suite.com/`.

[25] PRONOVOST, S., MORETON, H., AND KELLEY, T. Windows Display Driver Model (WDDM v2) And Beyond. In *Windows Hardware Engineering Conference* (2006).

[26] RAJKUMAR, R., LEE, C., LEHOCZKY, J., AND SIEWIOREK, D. A Resource Allocation Model for QoS Management. In *Proc. IEEE RTSS* (1997), pp. 298–307.

[27] ROUSSOS, K., BITAR, N., AND ENGLISH, R. Deterministic Batch Scheduling Without Static Partitioning. In *Proc. JSSPP* (1999), pp. 220–235.

[28] SPEC. SPECviewperf. `http://www.spec.org/gwpg/gpc.static/vp11info.html`.

[29] SPRUNT, B., LEHOCZKY, J., AND SHA, L. Exploiting Unused Periodic Time for Aperiodic Service using the Extended Priority Exchange Algorithm. In *Proc. IEEE RTSS* (1988), pp. 251–258.

[30] SPURI, M., AND BUTTAZO, G. Efficient Aperiodic Service under Earliest Deadline Scheduling. In *Proc. IEEE RTSS* (1994), pp. 2–11.

[31] YANG, T., LIU, T., BERGER, E., KAPLAN, S., AND MOSS, J.-B. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *Proc. USENIX OSDI* (2008), pp. 73–86.