Stony Brook University

A Fair-Share Scheduler for the Graphics Processing Unit

A THESIS PRESENTED

BY

ASHOK DWARAKINATH

TO

THE GRADUATE SCHOOL

IN THE PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

STONY BROOK UNIVERSITY

August 2008

Stony Brook University

The Graduate School

Ashok Dwarakinath

We, the thesis committee for the above candidate for the
Master of Science degree,
hereby recommend acceptance of this thesis.

Professor Tzi-cker Chiueh, Thesis Advisor
Computer Science Department

Professor Jennifer L.Wong
Computer Science Department

This thesis is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

A Fair-Share Scheduler for the Graphics Processing Unit

by

Ashok Dwarakinath

Master of Science

in

Computer Science

Stony Brook University

2008

The latest Graphics Processing Units (GPU) have more transistors than modern multi-core CPUs. GPUs have also evolved into general purpose stream processors capable of complex floating point computation. Programming language frameworks/environments which extend common programming languages like C are available to exploit this computation power. As a result, GPUs are not only being used for better 3D rendering, they are also being used to solve general purpose computation tasks like performing scientific calculations. With this new CPU vs. GPU power equation; there emerges a need to manage the GPU resource efficiently.

In this thesis, we look at the issue of fair allocation of the GPU resource among competing GPU applications. Graphics device drivers in modern operating systems adopt a first come first serve approach at allocating the GPU. With this approach, one of the applications can monopolize the GPU. We propose an alternate approach at allocating the GPU based on the deficit round robin algorithm which ensures nearly equal GPU times to competing applications. To implement this algorithm, we make changes to the graphics device driver subsystem, to have per-process queues for graphics commands. The algorithm also requires an estimation of GPU command execution times. A measurement based approach is used to maintain average execution times for various GPU command types. This allows estimation of future command execution times based on command type.

To

My Family

Contents

List of Tables

List of Figures

Acknowledgements

I wish to thank Prof. Tzi-cker Chiueh for giving me an opportunity to work on this project and for his guidance and support throughout the project. I am grateful to Prof. Jennifer Wong for reviewing my thesis and providing valuable comments for improvement. I would like to thank Mikhail Bautin, PhD Candidate at Stony Brook University for introducing me to this project and providing help in technical matters. I would also like to thank my ECSL lab-mates for creating a fun work place and for all the interesting discussions at work. And finally, I would like to thank my family, for their encouragement and support in all my endeavors.

Chapter 1

Introduction

GPUs have been growing in power to cater to the demand created by a multi-billion dollar computer gaming industry. Modern GPUs now have more transistors than recent multi-core CPUs [1]. Table 1 shows a comparison of an NVIDIA GeForce GTX 280 GPU with Intel Core 2 Extreme Quad-Core QX 9650. These are the frontline processors in terms of processing power in mid-2008. The higher computation power of the GeForce GPU is not just economics; it is also because adding more transistors to a GPU is more attractive because of their special needs. (More arithmetic operations per word preferred)

	GeForce GTX 280	**Intel Core 2 QX 9650**
Transistor Count	1.4 Billion	820 Million
FLOPS	933 G	< 100 G
Peak Memory Bandwidth	141 GB/s	6.6 GB/s

Table 1: CPU v/s GPU in terms of computation power

In addition to their raw power, GPUs are designed to be highly parallel processors. This is so that the vertices or pixels can be processed in parallel. The processors run shader programs on the vertices or pixels. In a more generic sense, GPUs operate on data records in parallel by running program fragments or kernels with the records as input. Thus any computation that operates on a large stream of records with similar operation being performed on each record is a good candidate to be ported to the GPU. This gives rise to an entire class of non-graphics applications like matrix computation, image processing and Protein folding calculations that can be ported to the GPU. This is commonly referred to as GPGPU [2]. (General Purpose computation on GPUs) But until recently, porting an application to the GPU was difficult because of lack of any programming frameworks and control flow limitations in kernels. (No if-then-else) This is not true any more with the advent of development frameworks like CUDA, BrookGPU and CTM. Stanford University's Folding@home project [3] which performs distributed protein folding calculations on ATI graphics cards and the SETI@home project which also has a GPU client are some real world examples of GPUs being used for non-graphics applications.

As GPUs start being widely used for non-graphics applications, and considering their immense power, it is inevitable that multiple such applications will be executed simultaneously and hence will compete for the GPU resource on a system. Thus there is a need for a scheduler that manages allocation of the GPU.

In current operating systems, the GPU resource is managed by the graphics device driver. The application programs link to a graphics library implementing the OpenGL or DirectX API, which in turn calls functions in a user level driver component. This user level driver converts OpenGL or DirectX commands into commands that are specific to the graphics card. The user level driver then issues a system call to the kernel level graphics driver to DMA the commands to the GPU. The user level driver also ensures that the application executes against the correct graphics state (view port specifications, lighting parameters etc…). This is usually done by obtaining a lock on the GPU before sending graphics commands. The process of obtaining the lock ensures restoration of the correct graphics state before the commands are sent. This is similar to process context switch on the CPU where the process state is restored before it starts executing on the CPU. Unlike CPU scheduling, where the context switch is transparent to the application, GPU scheduling requires the user level driver which is part of the application to explicitly restore the state whenever necessary and release locks on time. This opens up the possibility of clients holding the lock for too long and starving other clients.

To ensure fairness among multiple applications using the GPU, we explore a new design for the graphics device driver. We have developed a prototype called GERM (Graphics Engine Resource Manager) with per-process command queues in the kernel and a scheduler that uses an algorithm called deficit round robin to ensure fairness among competing applications. The rest of this thesis explores the design, implementation and evaluation of this prototype.

Chapter 2

Related Work

Windows Display Driver Model [4] (WDDM) introduced with windows vista seeks to solve the same problems as GERM. In Windows XP, the display driver model has the same problems described in the previous chapter. The GPU scheduling is first come first serve, which can essentially lead to starvation for some clients. WDDM v1.0 which is a basic version of the driver model and is implemented in windows vista has per-process command queues in the kernel, but the policy used for scheduling is not known. WDDM v2.0 is an advanced version of the driver model. It supports mid-command buffer pre-emption and per-process page tables with demand paging of video memory pages. This would require a new generation of GPUs. GPU has to support multiple contexts and interrupt the CPU on completion of a command group. Each context includes a ring buffer for commands and a page table for video memory. WDDM v2.1 is more advanced; it supports mid-pixel pre-emption. This is the ultimate level of device isolation to processes using the GPU. No process will ever starve. But these are tough problems to solve. (Consider the case of a shader that executes for a really long time) While WDDM has the same goals as GERM, it is targeted at windows operating systems. The GERM prototype has been implemented on Linux, is open source, and as such can benefit from contributions by the open source community.

Compute Unified Device Architecture [5] (CUDA) is a C language framework from NVIDIA that eases development of programs for the GPU. It provides simple extensions to the C language and an SDK to develop programs for NVIDIA GPUs. CUDA also requires a driver component to be installed in the system running CUDA programs. So, the CUDA software stack at the lowest level essentially consists of a driver which interacts with the GPU. The CUDA library runtime which sits on top of the CUDA driver translates commands from the CUDA program and sends them to the driver. CUDA programs are C programs with additional syntax to specify kernels. (functions executed by threads) A compiler front-end called nvcc is provided to compile CUDA-specific extensions. Nvcc compiles the CUDA specific code into object files called cubins. These cubins are loaded by the CUDA runtime and sent through the driver to the GPU. CUDA provides a multi-threaded programming model with facility to synchronize threads. The programmer can specify multiple threads that execute the same kernel in parallel. Threads that execute on the same processor core form a thread group. The thread group has access to shared memory on the core. (similar to L1 cache in CPU) Each thread also has local memory. There can be a maximum of 512 threads in a thread group. Each thread group runs independently on different processor cores. The job of a programmer is to partition a problem into multiple sub-problems each of which can be run independently

either in parallel or serial fashion. Each such sub-problem can be assigned to a thread group. Then the data in the sub-problem can be processed in parallel using threads in the thread group. CUDA is currently supported for GPUs based on NVIDIA's Tesla architecture.

ATI Stream SDK [6] is the successor to ATI Close To Metal (CTM) technology. It provides low-level access to the registers and memory on the GPU. The software stack at the lowest level consists of a device driver that provides a forward compatible hardware abstraction layer (Compute Abstraction Layer - CAL) to programs. This abstraction layer works very much like a virtual machine. It accepts intermediate code called AMD IL and generates GPU specific code that can be executed by the target device. The CAL API is used by the runtime library to send pre-compiled kernel routines to the GPU. The Stream SDK also provides a compiler called brcc to compile ATI-specific extensions to the C language. This is really a meta-compiler since the output of this is a C++ source file containing compiled kernels. (embedded in the source) The programmer has to write kernels that execute in parallel on data arrays called streams.

Both CUDA and ATI stream computing initiatives are a departure from the device driver architecture prevalent for graphics programs. They provide access to the computing power of the GPU without any need for the programmer to interact with features of the GPU like lighting parameters or viewport specifications that only graphics programs need. As such the current CUDA and CTM infrastructure doesn't concern itself with multiple programs using the GPU at the same time and hence it is not clear what policy they adopt to schedule commands from multiple programs.

SGI's multi-rendering X windows [7] support is based on modifying the X server to have multiple rendering threads in its address space. An OpenGL client first sends commands to X server, which then assigns these commands to a rendering thread. The rendering thread in turn parses these commands and sends them to the GPU. Once the rendering completes, the X server returns to the client and is ready to accept more commands. This approach obviously involves some latency as the clients need to send commands to X server. Fairness issues have not been explored in this implementation.

Chapter 3

Direct Rendering Infrastructure

Direct Rendering Infrastructure [8] (DRI) is a framework that allows OpenGL clients to directly access the graphics hardware without the need to communicate with the X server. The purpose of DRI is to provide hardware acceleration for 3D rendering. Direct rendering uses the graphics hardware to perform geometry calculations and is therefore fast. In contrast, In-direct rendering involves communicating with the X server, where the calculations are done in software and hence slower. OpenGL is a specification of graphics primitives from SGI. Mesa 3D [9] is an open source implementation of OpenGL developed primarily for Linux and FreeBSD systems running X. Mesa also provides user-level graphics drivers that interact with kernel drivers from the DRI project that enable OpenGL programs to use hardware acceleration. Our prototype design and implementation is based on Mesa 3D and DRI code and seeks to overcome problems with their existing design. Hence a detailed discussion of the two follows in subsequent sections.

3.1 Overview

Figure 1 gives an overview of the direct rendering infrastructure. An OpenGL program consists of the application code linked with the Mesa OpenGL library (libGL.so). The program is also linked with a GPU specific user level driver. (For ex: r200_dri.so). The OpenGL library uses the user level driver to send GPU specific commands to the kernel level driver. The kernel driver module sends the commands to the GPU by writing the commands to a ring buffer that is accessible by the GPU.

3.2 User level driver

Having part of the driver code in user level serves two purposes:

- It is easier to debug and maintain user level code.
- Security: User code is un-privileged and bugs here will not crash the system.

The GPU user level driver accumulates commands from the application and sends them to the kernel level driver through an IOCTL system call interface. In the current Linux DRI implementation the IOCTL interface is provided by libdrm, which provides a wrapper around the ioctl system calls to the kernel. The driver also maintains the graphics

state of the application and ensures that the commands are executed against the correct state. Figure 2 shows the major components of the user level driver and its interaction with the kernel level driver.

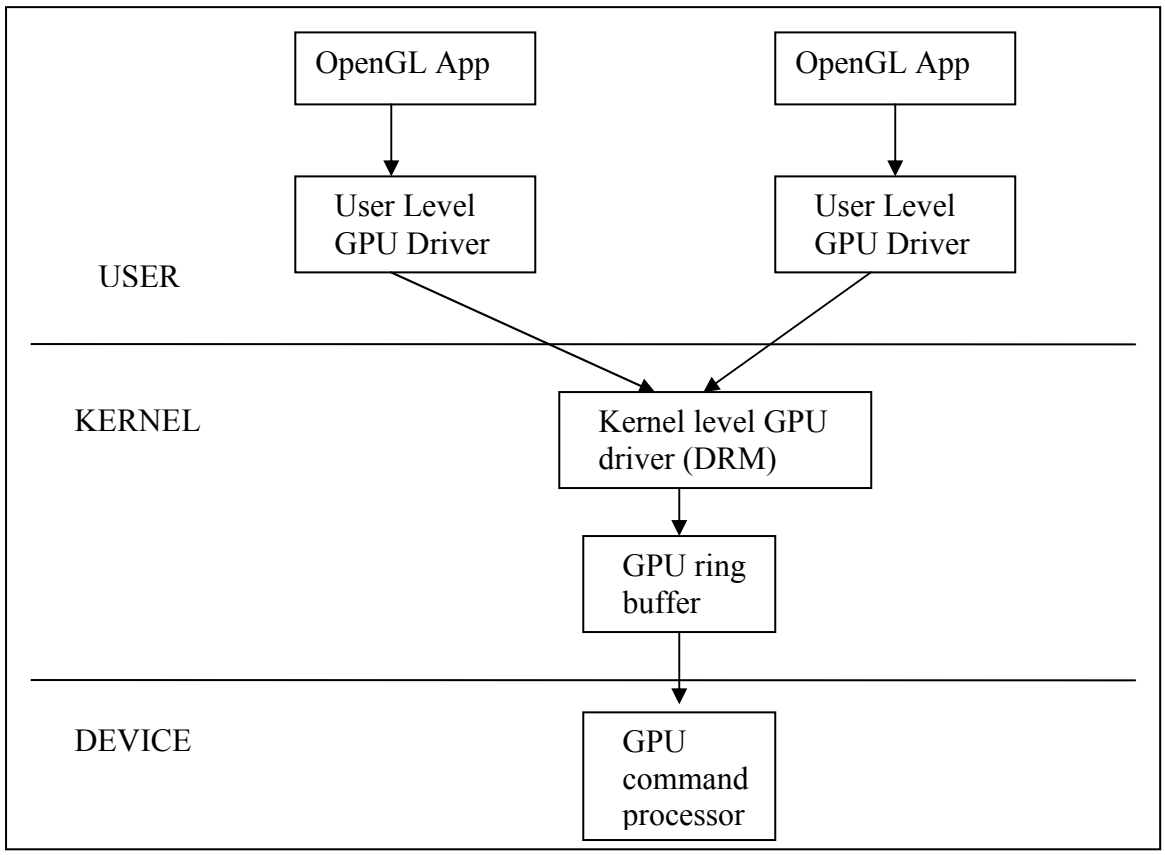

Figure 1 : High level view of DRI

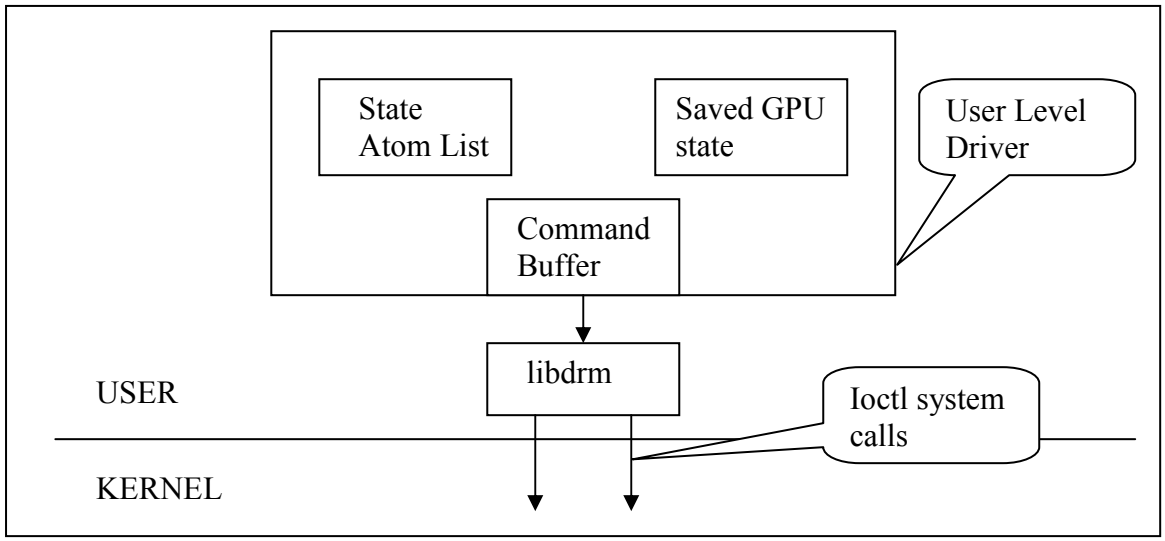

Figure 2 : User Level GPU Driver

The user level driver contains the following main data structures:

- ***Command Buffer*:** This is an 8 KB array that holds GPU commands before it is flushed to the GPU. Buffering ensures lesser calls into the kernel and allows command groups such as state restoration commands, glArrayElement commands to be atomically sent to the GPU.
- ***State Atom List:*** This is a list of state atoms (command sequences) that represents the state information that was last sent to the GPU. This basically consists of information such as transformation matrices, lighting parameters etc...
- ***Saved GPU state:*** This is a copy of the above information for internal book-keeping. This structure maintains the latest GPU state information that needs to be sent on the next buffer flush.

The user level driver also ensures that only one client is sending commands to the GPU at a time. This is done using the locking primitives implemented with the help of the kernel level driver. The user level driver sends an ioctl call to the kernel driver requesting a lock on the hardware. The system call returns once the client has the lock. If the client detects that the lock was previously held by another client, then the state information will be sent again on the next buffer flush. This is the equivalent of a context switch in the GPU.

The set of commands sent between a lock acquisition and a lock release is called a command batch. This is different from a command group, which is a set of commands that need to be sent atomically to the GPU.

The user level driver also manages data structures that provide the application with a view of video memory layout – the regions in video memory currently occupied or free. Each client maintains this information separately and keeps it up to date with the help of the kernel level driver. A detailed discussion of this is beyond the scope of this thesis.

3.3 Kernel Level Driver

Kernel level driver in DRI is called Direct Rendering Manager [10] (DRM). This consists of a top level common module (drm.ko in linux 2.6) and a driver specific module. (Ex: radeon.ko for ATI radeon cards) Figure 3 shows the main components of the kernel level GPU driver. The top level module (drm.ko) provides the code that exposes the graphics card as a char device. (implements the file ops necessary and creates a device node at /dev/drm). drm.ko also contains a handler for the ioctl call to /dev/drm through which the user level driver interacts. This handler in turn invokes the function in the GPU specific module that actually handles the system call. If the call requires sending commands to the GPU, the handler in the GPU specific driver would write the commands now formatted as command packets (with header and payload) to the ring buffer that is accessible by the GPU through DMA.

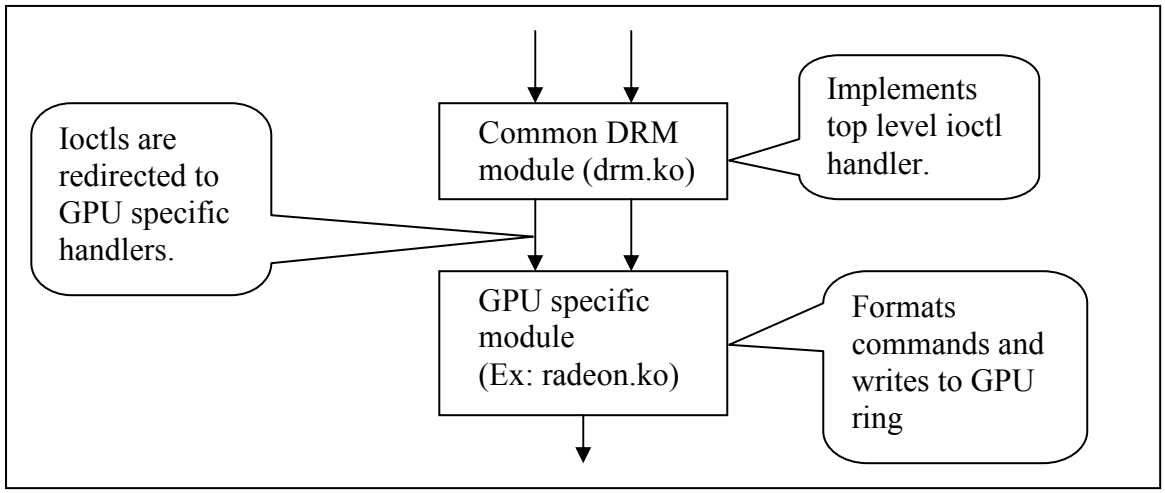

Figure 3: Kernel level GPU Driver

3.4 GPU Ring Buffer

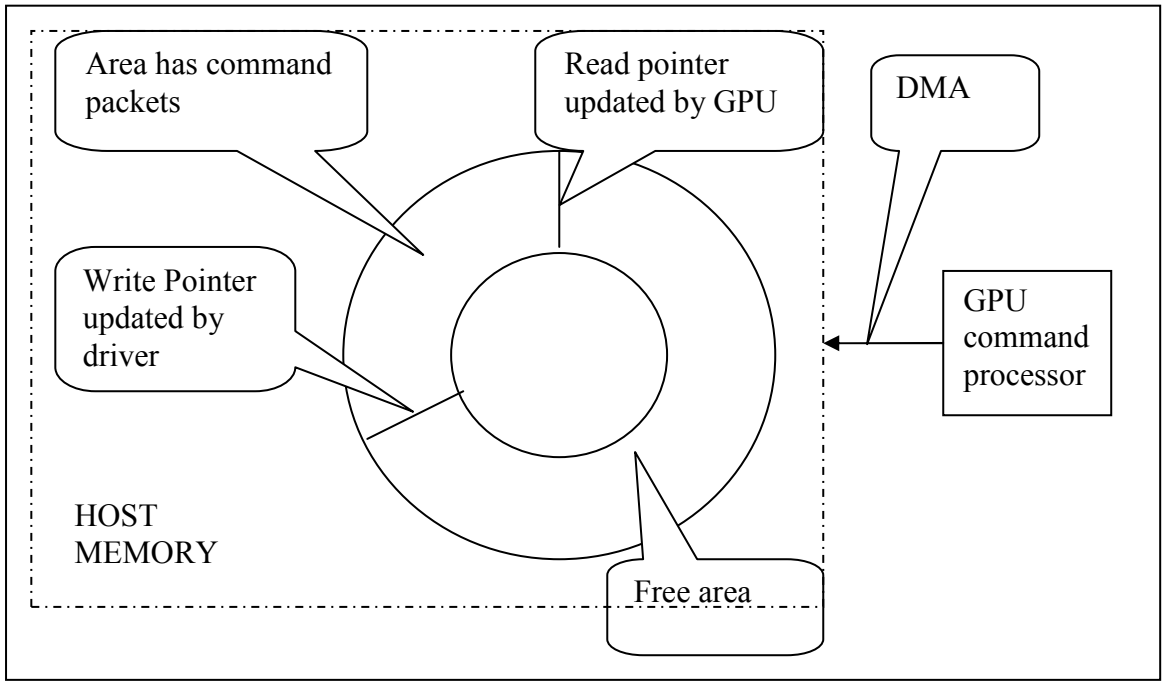

Figure 4: Ring buffer schematic

A circular buffer is usually used as the primary data-structure in producer-consumer setups. For communication between the GPU and the host CPU, an area of memory termed ring buffer is used. Figure 4 shows the main components of the ring buffer, and gives an indication on how the buffer is used. The buffer is written in the anti-clockwise direction in the figure above. There is a read pointer which is updated only by

the GPU. It indicates the position in the buffer at which the next read will take place. The write pointer is updated by the driver in the host after it writes command packets to the buffer. Note that there are two copies of read/write pointers, one in host memory and the other in GPU registers. The process of reading and writing to the ring buffer ensures that both copies are in-sync. Writing to the ring is through BEGIN_RING, ADVANCE_RING and COMMIT_RING macros in DRM code. BEGIN_RING macro ensures there is adequate buffer space. ADVANCE_RING macro advances the write pointer. OUT_RING(x) macro writes the byte x into the buffer. COMMIT_RING macro updates the write pointer register of the GPU and reads the read pointer value from the GPU register to synchronize read/write pointer values and ensure correct posting of data to the ring.

3.5 Command Flow

Figure 5 shows a command flow diagram with only the important buffers (data containers in general) in the user level driver and kernel driver that store data provided by an OpenGL application and how that data reaches the GPU. The diagram illustrates how vertex data is transferred to the GPU. Texture data transfer would be similar to this.

Vertex data consists of geometric co-ordinates, along with the color values (RGB) for that vertex. An OpenGL program typically calls the glVertex3f OpenGL function to specify this information. The Mesa 3D implementation of this function, calls a user level driver function to store this vertex data into a DMA buffer mapped into the address space of the process. The user level driver also maintains a vertex buffer that contains a pointer to this DMA buffer and maintains a vertex count variable as well as other meta-data about the vertices. Once a primitive (triangle, polygon) is output to the vertex buffer, a command packet containing meta-data about the vertices in this primitive is flushed to the user level command buffer mentioned in section 3.2. This meta-data contains the packet type and the start address and length of vertex data in the DMA buffer. Once this command buffer is full, it is sent through the IOCTL interface to the kernel level driver, which writes the commands to the GPU ring buffer. The GPU then processes the command and obtains the vertex data through DMA.

Figure 6 shows a timing diagram with various components of DRI, and how they interact to push information from the OpenGL App to the GPU. A glVertex3f OpenGL call gets translated to a user level driver-specific call, xx_Vertex3f. At this stage, the vertex data is merely buffered in the user driver as discussed previously. When the command buffer gets full or due to an explicit flush call, the command data is sent to the kernel level driver through an IOCTL call. Before the data is sent, a lock is obtained on the GPU, by a system call to the kernel. If another process previously held the lock, state data is sent to the GPU. Access to the ring buffer is through macros discussed in section 3.4.

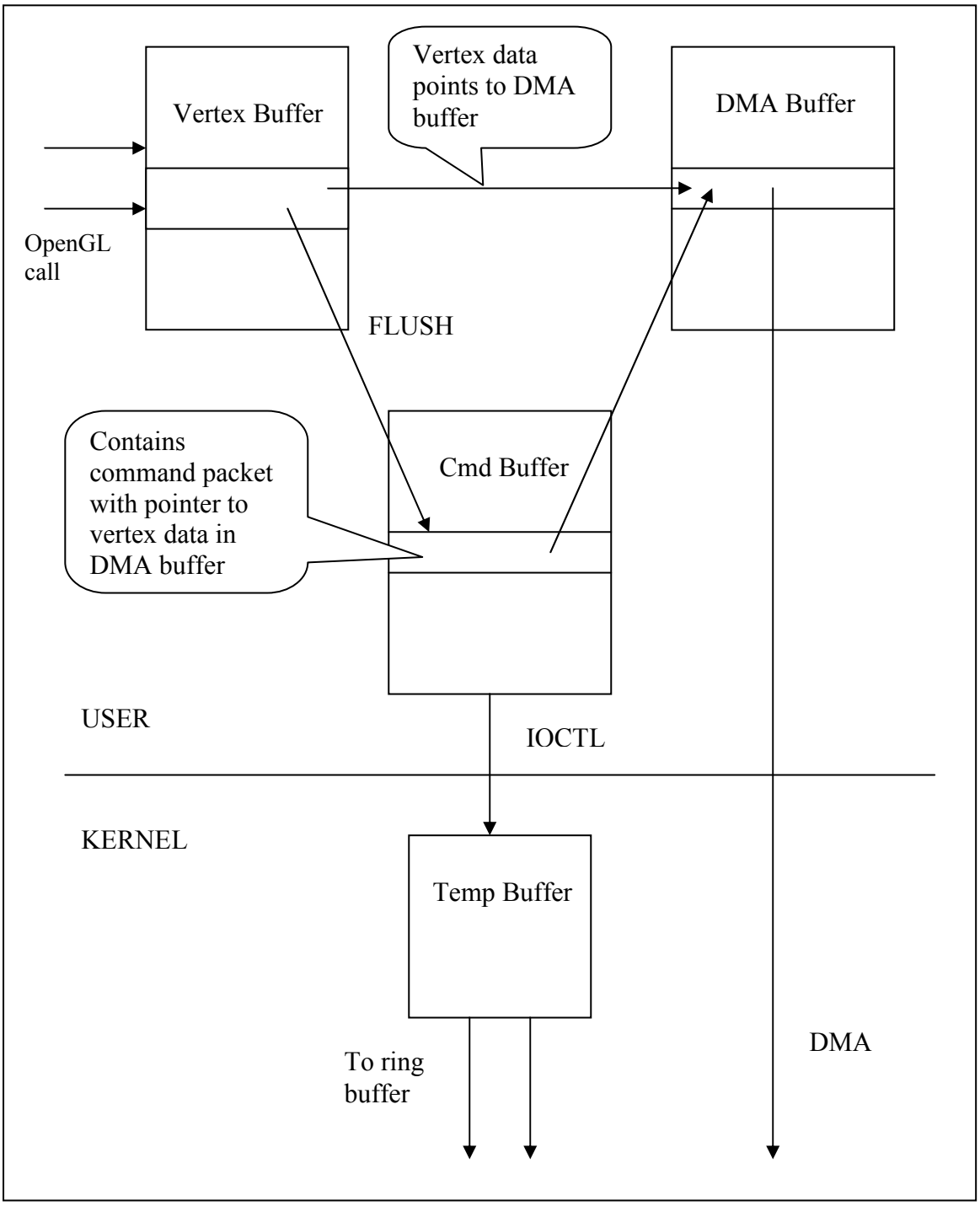

Figure 5: Command flow diagram showing the buffers containing vertex data

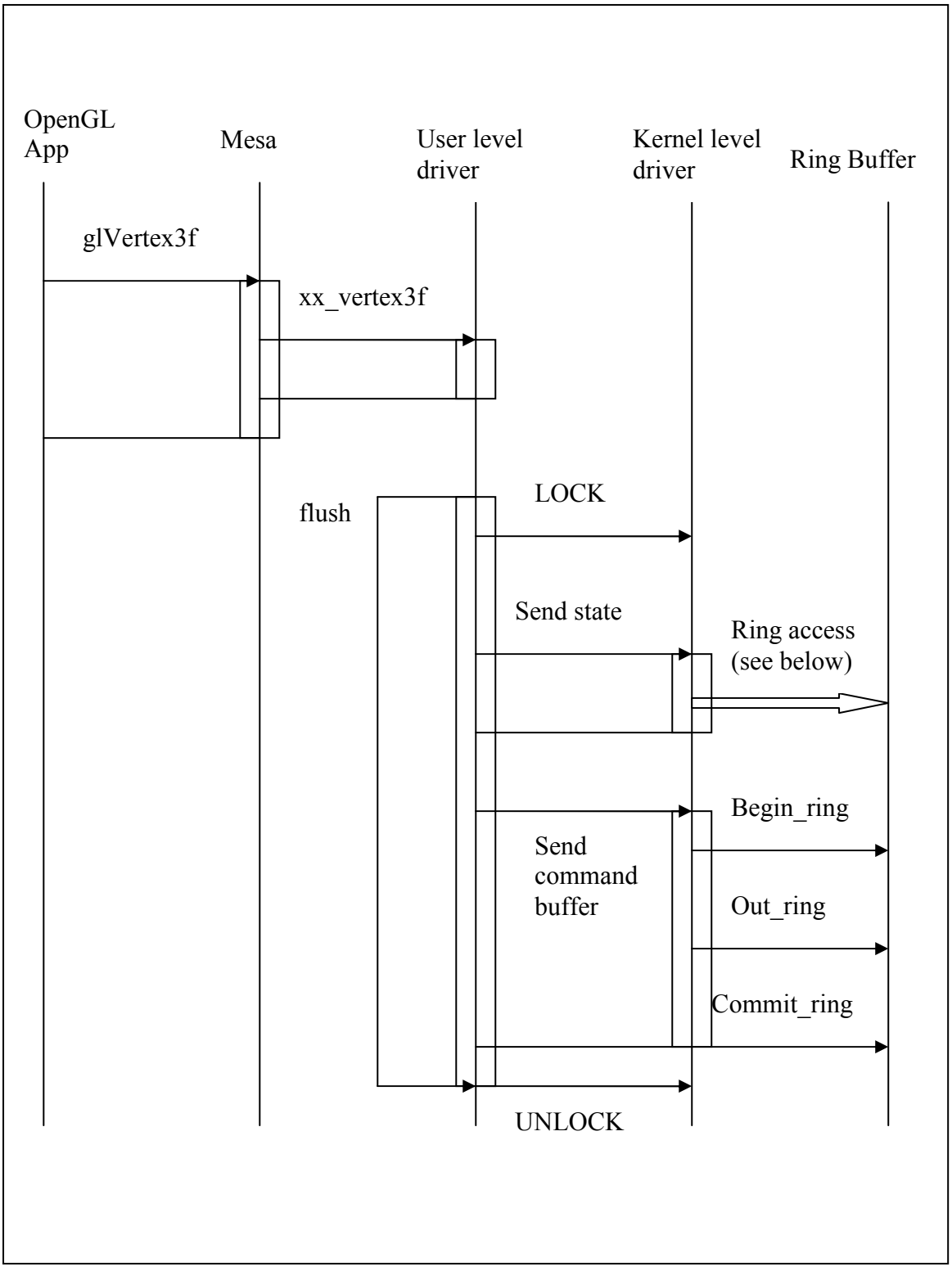

Figure 6: Timing diagram showing flow of data from OpenGL App to ring buffer

3.6 Limitations of DRI

As shown in the timing diagram in the previous section, the user level driver ensures exclusive access of the GPU to a process by obtaining a lock on the GPU. There are two issues to note here:

a) The lock acquisition and release is entirely done at the user level. A malicious client can hold the lock for any amount of time and prevent other processes from accessing the GPU.
b) The amount of GPU time consumed is directly proportional to the CPU time allotted to the process. A process which consumes more CPU time, but needs less GPU resources still gets more GPU time than another process which needs more GPU resources, but less CPU time.
c) The command batch that is sent atomically to the GPU can take an arbitrary amount of time to execute.

Point (a) above can lead to starvation of clients. Point (b) and (c) can lead to un-fair allocation of the GPU to competing processes. The next chapter discusses the design of GERM, which seeks to address the above limitations.

Chapter 4

Design

4.1 Design Goals

In the previous chapter we looked at DRI design and its limitations. We have modified DRI to overcome these limitations. In particular, we have modified the graphics driver with the following high-level goals in mind:

- To provide equal GPU time to a set of competing GPU applications with similar requirements.
- To prevent starvation for applications with low GPU requirements.

The following sub-sections look at the various design issues and the solutions considered in achieving the above goals. The chapter ends with a section on the detailed design of the prototype, including the changes to kernel level and user level drivers.

4.2 Overview

It is obvious that in order to achieve the stated goals we need to schedule commands at the command group level as against the command batch level in DRI. With the current (lock---command flush---unlock) model in DRI, this is not possible. The locking mechanism has to be disabled to pre-empt a process at command group level. Since, in the current DRI design, graphics state maintenance is closely linked to locking, we need to develop a different mechanism to restore the graphics state of a process whenever we schedule command groups belonging to a process. We also need a scheduling policy that enforces our notion of fairness. Since the notion of fairness is closely linked to the GPU time consumed by a process, we also need a way to measure how long a command group takes to execute on the GPU.

The issues summarized above can be formally stated as follows:

a) Fine-grained command scheduling – This involves scheduling commands at the command group level. To implement this, we need a way to buffer commands till they can be sent to the GPU. Note that this was not necessary in DRI, as the commands were sent to the GPU in a single system call from the user level driver. Closely linked to the issue of fine-grained scheduling is the problem of maintaining state. Where to maintain state information, how to update it, and how to ensure that the command groups are always executed against the correct state are some of the other issues.

b) Scheduling policy – We need a scheduler that can send command groups to the GPU. The scheduler needs to implement an algorithm that ensures that all processes get to send commands eventually and that they send roughly commands of the same cost each time.
c) Measuring GPU execution time – We need a notion of GPU execution time. The problem here is that until recently, GPUs didn't interrupt the CPU once a command group was processed.
d) GPU command cost estimation – We will see when we discuss scheduling policy that GPU command cost estimation is an important issue to consider in our design. GPU command cost estimation is the ability to predict the GPU execution time of a command group based on its contents. We need to develop some kind of heuristics to solve this problem.

The following sub-sections discuss each of the above issues in detail.

4.3 Fine-grained command scheduling

Figure 7 illustrates fine-grained command scheduling. Command group is the unit of commands sent atomically to GPU. A scheduler sends the command groups on behalf of a process. Note that such a scheduler is best implemented in the kernel, since it needs access to the GPU ring buffer and has a global role to play in the graphics system. After a command group is sent, the scheduler may switch context, to send command groups belonging to a different process. To be able to do this, the command groups need to be stored in per-process command queues.

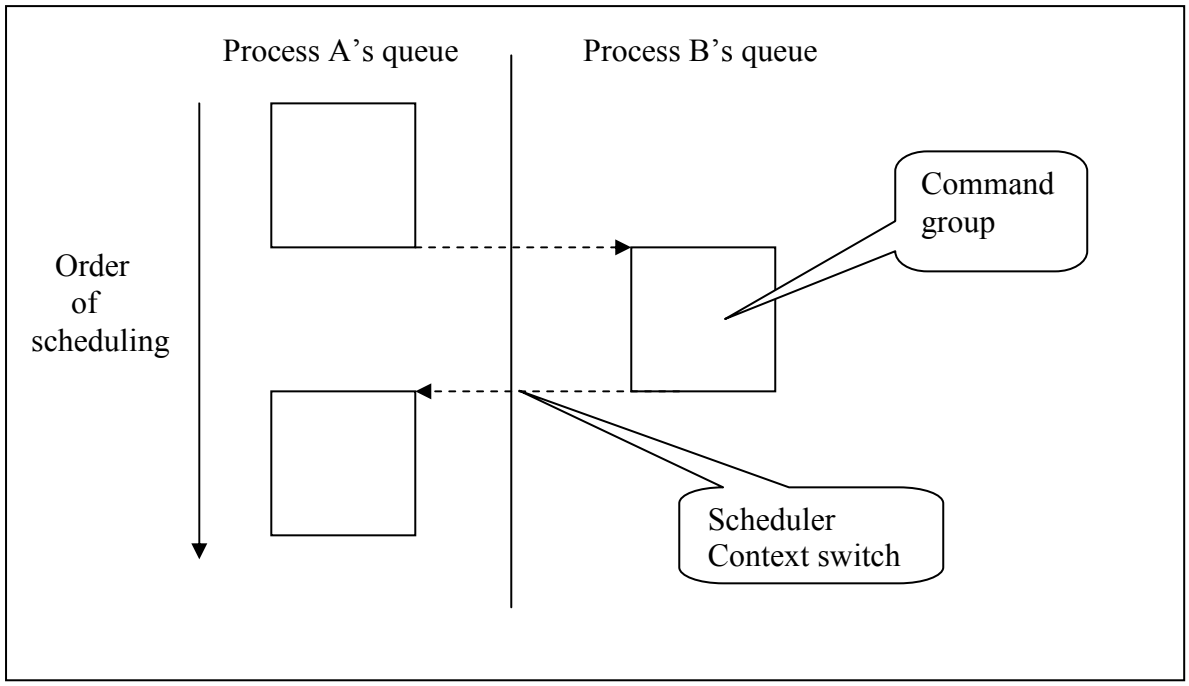

Figure 7: Fine grained command scheduling

4.3.1 Command Queues

Command groups need to be buffered before the scheduler can send them to the GPU. This is done in per-process command queues. There are various ways to implement command queues:

a) One way is to maintain the buffers in the kernel and memory-map them to the user program. The user program can insert commands in this buffer through the user level driver. This is a zero-copy approach to send command groups to the GPU. The disadvantage is that this requires a complete re-design of DRI user level and kernel drivers. The kernel level driver in DRI writes to the ring buffer in a format that the GPU understands. This needs to be now done in the user level driver before inserting commands to the command queue to maintain the zero copy advantage. The absolute lack of documentation in the open source drivers make such a huge change difficult to implement.
b) Another way is to maintain buffers in the kernel and modify user level driver to insert commands into these buffers instead of directly writing to the ring buffer. This is much easier to implement, as the access to the GPU ring is through well-defined macros. Changing these macros to insert commands into the command queues is easy to implement. The disadvantage of this approach is that there is an extra copy involved. The command groups are first copied into the command queues, before the scheduler writes them to the ring buffer.

We have chosen option (b) because it is easier to implement correctly. GPGPU programs generally upload a single kernel which is run multiple times on input data. As a result it is unlikely that an extra copy while inserting commands into the command queue will be a bottleneck for such programs.

4.3.2 Context switch (state restoration)

User level driver locking needs to be removed to implement fine-grained scheduling. With this change, the scheduler needs to make sure that the command group is executed against the correct state. This requirement suggests that the state of each process needs to be maintained by the scheduler. The state can also be updated by the OpenGL App during the course of execution. In DRI, this state data is sent to the GPU in the same way as other commands. This needs to be changed since the scheduler is the one that sends data to the GPU. (And it needs to have the most recent state of the App, in case of a context switch). The user level driver and kernel driver are modified to adhere to a protocol in the data they exchange. This is to recognize the type of command (normal data or state change command) in the command group. In the kernel level driver, a state change command results in change of the internal state data maintained by the scheduler for the application. This state change is also sent to the GPU by the scheduler before

succeeding commands in the command stream are executed. The exact format of commands and command groups is discussed in section 4.7.3.

4.4 Scheduling Policy

GERM scheduling policy is based on an algorithm called Deficit Round Robin [11] which was proposed for fairness in packet switching with multiple input flows (queues). This algorithm requires O(1) work per packet. The main idea of the algorithm is to assign a quantum (say number of bytes) to each input flow. This is done every scheduling round of the round robin. If an input flow uses less than a quantum in the current round, then it gets to send more data in the next round. (deficit of the current round plus the quantum for the next round). Similarly, if it consumes more than a quantum in the current round (say, packet size is more than the quantum assigned), then it gets to send less data in the next round of scheduling. This works well with variable-sized packets and thus reduces packet processing time. (no fragmentation required) Note that it is easy to implement a priority-based scheme on top of quantum-based scheduling. If one of the input flows has higher priority, then it can be assigned a higher quantum each round.

Since this is a round robin algorithm, processes will not starve. The scheduler visits the command queue of every process in each scheduling round. As long as a process has commands to be sent, it will get to use its quantum.

Figure 8 shows how this algorithm is adapted for GPU scheduling. The GERM scheduler maintains per-process queues for commands. The commands are in containers called command groups or packets, which is a unit of commands that needs to be sent to the GPU atomically. The quantum in this case is GPU time. Unlike data packet size, we don't really know how much GPU time a given command group will take to execute. It may not even be proportional to command group size. One way to know the time is to refer to GPU specifications – to find the cycle count for various commands executed by a particular GPU. While building the command packets in the kernel, we can tag packets with their GPU cost from the above data. This will be fairly accurate. Unfortunately, this data is not available to the general development community. The other alternative is to estimate command group execution time by using heuristics and a measurement-based approach. We can maintain running averages of command group execution times and use that to estimate the cost of command groups in the command queues. This scheme is discussed in the next sub-section.

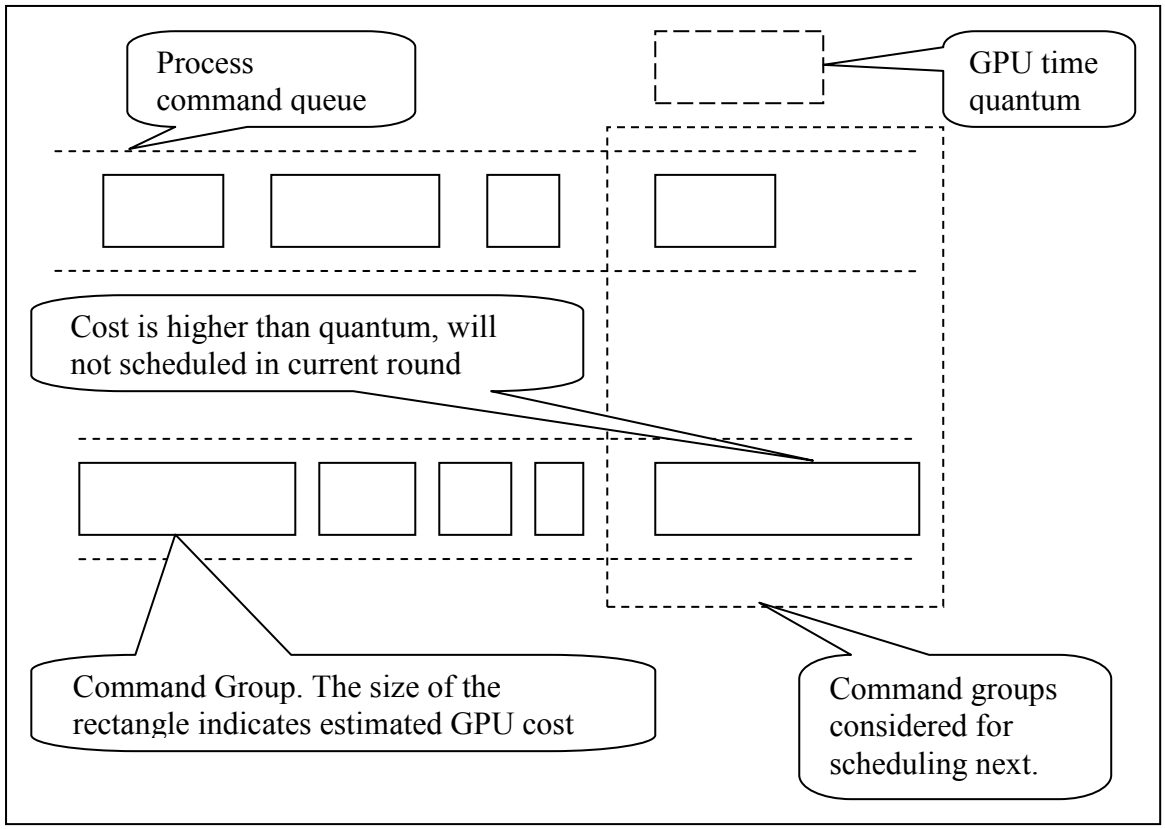

Figure 8: Scheduler Design

4.5 GPU commands cost estimation

We estimate the cost of GPU commands using *heuristics* on the contents of the command group. We examine the contents of the command group and look at the following characteristics of the command group in the order they are mentioned below:

- ***Presence of Texture Dispatch Command:*** Texture dispatch command results in transfer of textures (image data) from the host memory to the GPU. The command group containing this command should consume high GPU time.
- ***Presence of Buffer Swap Command:*** In case of double buffering, a buffer swap command switches the back buffer (which contains the latest pixel data) with the front buffer (which would be used to render on screen). Double buffering results in better frame rates as one buffer can be used to display on screen while the other buffer is being drawn into. We have found through measurements that this operation is costly and needs to be accounted separately.
- ***Number of Vertices:*** Vertices are processed in the geometry stage of the graphics pipeline to calculate visibility, color and projection of the 3D scene. Besides, vertex shaders are also run on the vertices if specified. It is safe to consider that the GPU time consumption of a command group is proportional to the number of vertices in the command group.

- ***Number of Bytes in command group:*** If none of the above commands are found, we consider the cost of a GPU command group to be proportional to the size of the command group.

We use a measurement based approach to quantize the costs mentioned above. The scheduler maintains the following *averages or coefficients* to estimate the cost of GPU command groups.

- ***Time to upload 1 byte of texture:*** With each texture dispatch command we find out the size of the texture that is associated with that command and by measuring the time taken by this command to execute, we can calculate the average time it takes to upload a byte of texture to the GPU. When a command group contains a texture upload command, we just multiply the size of the associated texture with this average to get the estimated GPU time for the command group.
- ***Average time for a buffer swap:*** We use this average to estimate the cost of a command group containing a buffer swap command.
- ***Average time per vertex:*** We use this average to estimate the cost of a command group with a given number of vertices.
- ***Average time per command byte:*** We use this average to estimate the cost of a command group that does not contain any of the above commands.

Note that these averages are used in the order they are mentioned above. So, the command group is checked for vertex data only if it does not contain texture dispatch commands and buffer swap commands. Central to the measurement-based approach described above is the aspect of measuring GPU execution time. This is discussed in the next section.

4.6 Measuring GPU command execution time

Most GPUs don't interrupt the CPU once a command group is processed by the GPU. So, it is difficult to measure the exact time that a command group took to execute. Recent GPUs can be setup to interrupt the CPU once a command group is processed. This is by using something called fence objects where a command to write a scratch register is injected into the command stream. This command when executed causes an interrupt to the CPU. But the GPU on which our implementation is based doesn't have this feature. So, we have used a polling based approach.

After each command group is written to the ring buffer, we inject a command that increments a scratch register which we call the timing register on the GPU. This acts like a counter for the number of command groups executed on the GPU. We poll the value of this register regularly to find the number of command groups executed since the last time the register value was read. We also measure the CPU time when the timing register is read. With these measurements it is possible to calculate the GPU time consumed per command group. It is important that we poll regularly to ensure accurate measurement. We discuss how this is done in the next section.

4.7 Detailed design

4.7.1 Changes to User Level Driver

The following changes are required in the user level driver:

- ***Removal of calls requesting hardware locks*** – The locking API in the user level driver consists of LOCK_HARDWARE and UNLOCK_HARDWARE macros. These macros are no longer required, since context switching and atomicity of command groups dispatch is handled by the scheduler present in the kernel level driver in GERM. These macros have been modified to do nothing in the driver code.
- ***Add command group heuristics information*** – Command group cost estimation requires information on the contents of a command group like the number of vertices in the command group. This information is available only to the user level driver. The driver has been updated to communicate this information to the kernel level driver.
- ***Tag state information*** – Hardware state is restored by the scheduler during a context switch. So, when the state atoms are emitted by the user level driver into its command buffer, it needs to be tagged so that the kernel level driver can identify this information in the command stream and update the state information it maintains.
- ***Tag DMA Buffer Discard*** – DMA Buffers are used to dispatch vertex lists. These buffers are re-used when the GPU has processed the current buffers. This is done by incrementing an age register on the GPU. This has to be done by the scheduler now. So, these buffer discard commands are tagged in the command stream.

4.7.2 Changes to Kernel Level Driver

Figure 9 describes the major components in the kernel level driver and their interaction. The components are described in detail below.

- ***Per process command queues*** – This is a linked list of 96KB buffers. Buffers are allocated and freed as commands are queued and processed.
- ***Change ioctl handlers*** - The ioctl handlers in DRM write commands to the ring buffer using BEGIN_RING, OUT_RING, ADVANCE_RING and COMMIT_RING macros. These macros have been modified to write to the command queues instead.
- ***Scheduler thread*** – The scheduler that implements the deficit round robin algorithm is a separate kernel thread. This thread visits the active list of processes in a round robin fashion and dispatches the commands in their command queues to the ring buffer.

- ***Timing thread*** – Our scheduling algorithm relies on accurate estimation of GPU command cost. So, we need to tune the cost metric coefficients we maintain (average time per vertex, average time per command byte, etc…) in real time. This tuning is done by the timing thread, which wakes up every jiffy to poll the timing register on the GPU and calculate the time consumed per command group. This data is then used to update the cost metric coefficients.
- ***Work Chunk queue*** – Note that for the timing thread to update the cost metric coefficients, it has to know the characteristics of the command group. (if the cost of that command group was estimated with cost metric coefficients associated with texture dispatch or buffer swap or vertex or command byte) So, we have another queue linked from the task structure that contains meta-data about the command groups sent. This meta-data is contained in a work chunk structure, an instance of which is en-queued into the work chunk queue when a command group is sent to the GPU, and de-queued when the timing thread finds out that the corresponding command group has been processed by the GPU. (this is fairly straight-forward as the GPU is a FIFO engine)

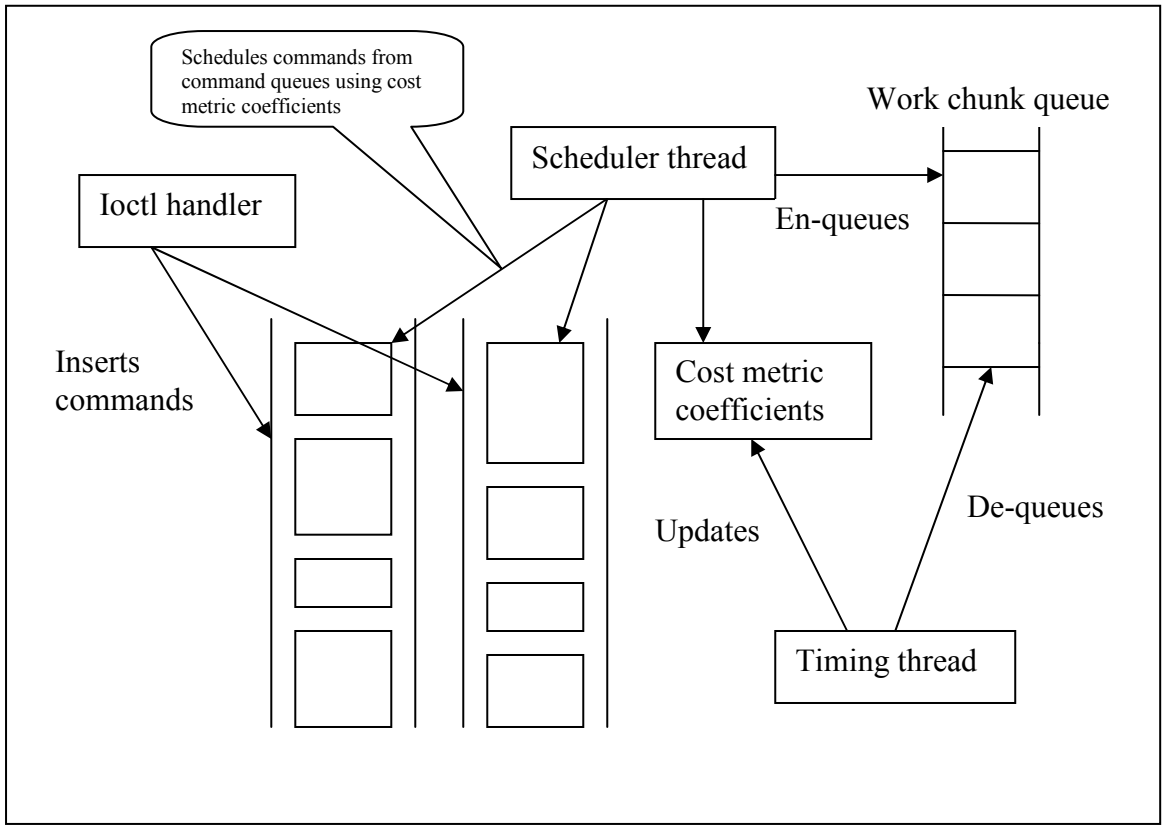

Figure 9: GERM component diagram

4.7.3 Command group format

Figure 10 shows the format in which command groups are stored in the command queues. A command group consists of a set of command items with a header storing meta-data about the command group. This meta-data consists of cost information like number of vertices in the command group; number of command bytes, number of commands in command group, number of buffer swap commands, and size of texture data if a texture dispatch command is present in command group. Note that this command group format is internal to GERM and only the command data gets sent to the GPU.

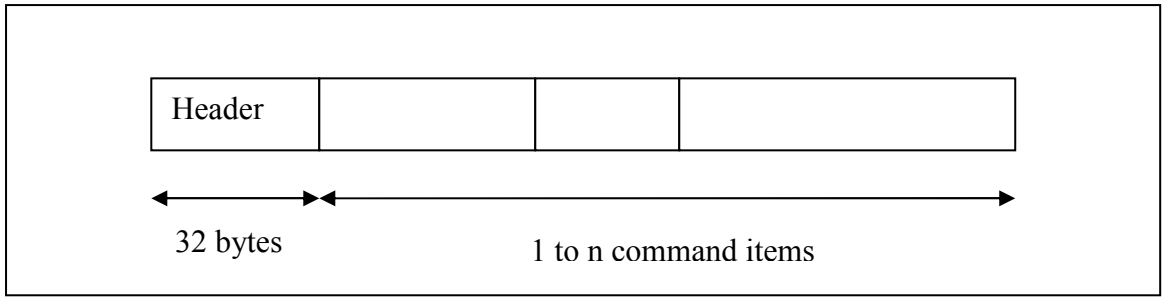

Figure 10: GERM command group format

4.7.4 Command Format

Figure 11 shows the format of a command item in the command group. A command item can be just command data, or a state atom, or a buffer swap command. These commands are distinguished by a special 2 byte mask (SC_MASK) in the beginning of the command item. Note again that this command format is internal to GERM, only the command data is sent to the GPU.

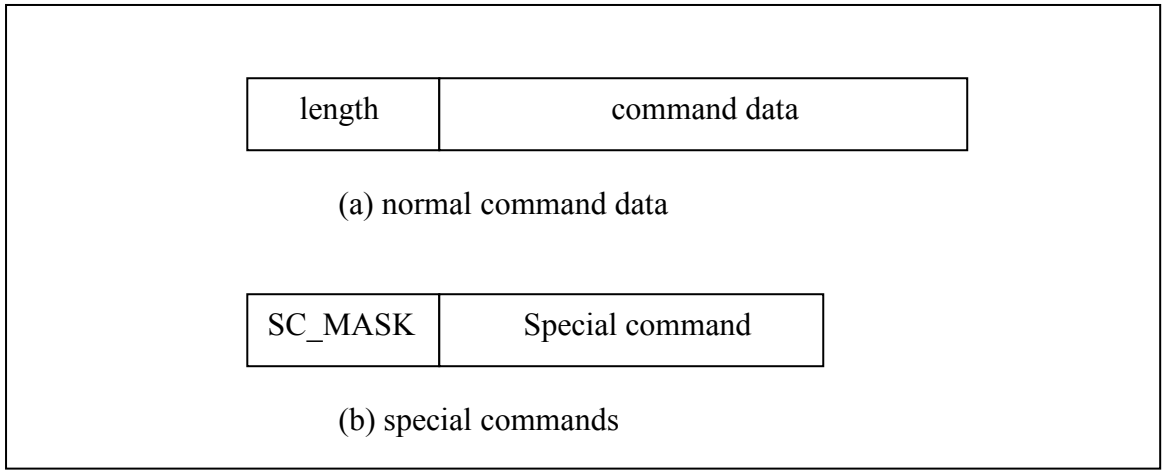

Figure 11: GERM command item format

Chapter 5

Implementation

GERM is currently implemented for ATI Radeon r200 GPU [12]. This GPU was chosen because of the availability of open source drivers. Software requirements for the prototype are as follows:

- Mesa 3D version 6.5.3
- X server version 6.9
- DRM version 1.28
- Linux Kernel 2.6.x

Changes are required in the user level r200 driver and the kernel level driver. (DRM, radeon) The prototype is provided as a patch to the standard driver software.

5.1 Main data structures

The following are main data structures used in the prototype:

- ***GERM device structure*** – This structure has been added to the DRM code to maintain GERM specific information. This information includes the list of tasks currently accessing the GPU (active list), pointers to the task structures of the scheduler thread and timing thread, the current hardware state of the GPU and locks to protect these members. This device structure is linked from the private info of the DRM device structure used in DRM code.
- ***GERM task structure*** – This structure represents a task accessing the GPU. This structure contains the list of buffers used to store commands emitted by the task, heuristic information for command cost estimation (average time per vertex), hardware state of the task and locks to protect these members.
- ***GERM command buffer*** – This structure contains an array of 96K bytes to store the command data. A linked list of these buffers is accessible from the task structure.

5.2 Scheduler thread

```
loop until shutdown
      current_task = next task in active list
      Dequeue current_task and insert at end of active list
      current_task.time_left += GERM_QUANTUM
      command_cost = cost of next command group
      loop while command_cost < current_task.time_left
            schedule command group to be sent to GPU
            create new work chunk instance
            fill the work chunk with command group meta-data
            insert into work chunk queue
            command_cost = cost of next command group
            current_task.time_left -= command_cost
      end loop
end loop
```

Figure 12: Scheduler thread main loop

Figure 12 shows the pseudo-code for the main loop in the scheduler thread. The active list is a list of GERM task structures. The time_left variable in the task structure indicates the GPU time available for the task. Locks are used to avoid simultaneous access of the task structure members by the timing thread. (Not shown in the figure for clarity) GERM_QUANTUM is the quantum value in the deficit round robin algorithm. In our implementation this has been chosen to be 0.0005 * (CPU clock speed). This is considering the cost of command transfer to the GPU which is more of the order of a memory access.

5.3 Timing thread

The timing thread wakes up every jiffy and probes the GPU to find out the number of command groups that have been processed since the last probe. Figure 13 shows the main loop in the timing thread. The number of command groups executed is found by reading the timing register and taking the difference of that value and the last probe value. If there are command groups that have been processed, we measure the time consumed per command group and then use the work chunk list to find the meta-data

about each of the command groups processed. The meta-data includes estimated time of the command group, the task associated with command group and the cost metric used for estimation. The deficit round robin algorithm assumes that the cost value of the command group is known. Since we only have an estimate while scheduling, we need to correct the actual time left for a task once we know the correct cost value. This is done by updating the time_left variable of the task. The cost metric coefficients are then updated.

```
loop until shutdown
      sleep for 1 jiffy
      current_probe_value = read timing register
      num_cmd_grps = current_probe_value – last_probe_value
      if (num_cmd_grps > 0)
            current_time = system time
            time_consumed = current_time – last_probe_time
            time_per_cmd_grp
                  = time_consumed / num_cmd_grps
            loop while num_cmd_grps > 0
                  wc = dequeue next work chunk instance
                  task = task associated with wc
                  task.time_left = task.time_left +
                                   wc.estimated_time –
                                   time_per_cmd_grp
                  update task cost metrics using wc
                  and time_per_cmd_grp
                  num_cmd_grps -= 1
            end loop
      end if
      last_probe_value = current_probe_value
      last_probe_time = current_time
end loop
```

Figure 13: Timing thread main loop

5.4 *Portability*

GERM is currently tested to work on ATI r200 GPU. The current device driver code organization requires us to modify GPU-specific user level driver and kernel level driver. But it should be easy to port major portion of GERM which includes the scheduler and timing thread to any other GPU. There are some aspects which include dealing with GPU state that would require careful review when porting to another GPU.

Chapter 6

Evaluation

6.1 Methodology

Running multiple graphics applications is not the norm currently. As GPGPU applications get more popular there is a good reason for multiple such programs to be run simultaneously. GPGPU applications essentially have a part that runs on the CPU (network access, synchronization between threads) and a part that runs on the GPU (core computation). It makes sense to run multiple of these together. (when one program is running code on CPU, another can use the GPU) GPGPU applications are just starting to make their presence felt and it is difficult to find representative applications for testing. It is also tough to find an application that runs on all GPUs, as the development frameworks are targeted for a specific GPU model. In our evaluation we have used a set of graphics applications instead. These graphics applications provide a good enough instruction mix to test our command estimation and scheduling. We have written our own GPGPU programs (matrix multiplication and string comparison) to evaluate our prototype.

One of the goals of GERM is to ensure that applications with similar requirements get similar share of GPU time. One of the ways of measuring fairness is to look at Frames per second (fps) of an application. If an application renders at F fps when running alone, then the time to render a single frame is 1/F seconds. If this application then runs at f fps when running along with other applications, then the GPU time consumed by this application is f/F seconds. One could then look at the maximum variation in GPU times consumed by a set of applications to measure fairness. Lesser variation implies better fairness. A more accurate measure of GPU time consumed can be obtained from within GERM. Note that we need to measure command group execution times in our scheduling algorithm. By summing up these command group execution times, we can obtain the overall GPU time consumption of a process. To obtain these times, the execution time statistics of the set of active applications are exported through the /proc interface. These statistics are then used to determine fairness.

GERM is built on top of DRI. This was done to avoid major changes in the graphics driver architecture. But this adds an overhead to applications running on GERM. To measure this overhead, we run multiple instances of the same application on DRI and run the same instances on GERM. We observe the change in frame rate (fps) in these experiments. The difference in frame rate can be considered as the overhead associated with GERM.

It would be interesting to see the accuracy of command group estimation heuristics used in GERM. This is important because the effectiveness of the deficit round

robin algorithm depends on accurate estimation. For each type of heuristic, we print the estimated time of a command group based on that heuristic and the actual time consumed to get an idea of its accuracy. The following sections discuss each of the above evaluations in detail.

6.2 Fairness

Fairness is the main goal of GERM. We define fairness in the following way – if t[i] is the GPU time consumed by process "i" in a set of n competing processes, t_{max} and t_{min} are maximum and minimum GPU times consumed by processes in the application mix, then the following formula gives a measure of fairness (Utime) for the application mix -

$$Utime = (t_{max} - t_{min}) / \sum t[i]$$

Note that a higher value of Utime indicates lower actual fairness. The statistics on the GPU time consumed by the current set of processes is available through a /proc interface exposed by GERM. The experiments to measure fairness were conducted over a 2 to 5 second period. The applications used for experiments are described in the next sub-section.

6.2.1 Test Applications

Mesa provides demo programs along with the OpenGL library. We have used these applications in our evaluation. But there aren't any GPGPU programs that run on ATI r200 GPU. So, we have written a string comparison program and a matrix multiplication program using the ATI_fragment_shader extension [13]. This extension provides assembly-like instructions to add, subtract and multiply 8 bit numbers. The fragment shader cannot access arbitrary memory locations in the GPU. It can only access the texture data mapped to the current pixel being processed. For simplicity, we map textures such that there is a one-to-one mapping between texture elements and pixels in the drawing area. So, to represent a 100x100 matrix, we have a 100x100 texture array mapped to 100x100 pixel drawing canvas. The fragment program code is executed for each pixel in the screen area.

To add two 100x100 matrices, the matrices are represented in two textures of size 100x100. These textures are mapped onto a 100x100 vertex grid with a view-port of size 100x100 pixels. The fragment program has a single instruction to add two numbers. The output matrix can be read by the program using a glReadPixels call.

Multiplying two matrices is more complicated. Matrix multiplication involves multiplication and addition of numbers in the rows and columns of the matrix. Given the one-to-one mapping of texture data and pixels on screen and other limitations (only 6 texture units, 16 total instructions in the fragment program, no arbitrary memory reads) multiplying 2 matrices requires multiple passes on the input data and intermediate matrices.

6.2.2 Test Results

Table 2 shows the test results of the fairness experiments. The difference in GPU time consumed by any 2 processes in the application mix is consistently below 5% which meets our goals. Correct cost estimation is the key to ensuring fairness in our implementation. The slightly higher Utime values for matrix multiplication programs indicate that for some command groups our estimation was not accurate. These are limitations of a measurement approach to cost estimation. More accurate estimation would require parsing the GPU commands, which would make the implementation highly GPU-specific.

Application Mix	Fairness (Utime)
Gears, Train	0.12%
Quake 3, Train	0.02%
Gloss, Gears, Train	0.15%
3xString Comparison	0.97%
2xMatrixMul128×128	0.55%
MatrixMul128×128, String Comparison	1.53%
MatrixMul128×128, MatrixMul256×256	2.40%

Table 2: Fairness values of different application mixes under GERM

6.3 Overhead measurements

The overhead associated with GERM is from two sources:

- **Context switching**: Fine-grained scheduling results in more number of context switches. Each context switch requires the state information to be sent to the GPU and the GPU execution pipeline to be drained completely of existing commands.
- **Command queues and scheduling**: There is an extra copy operation associated with storing commands in command queues before they are scheduled to be sent to the GPU. This results in an overhead for applications which send a lot of commands to the GPU.

6.3.1 Context switching overhead

To measure context switching overhead we used a program that draws 100 triangles per frame. We modified GERM kernel code to induce N context switches per frame. (a single context switch time is too small to measure) Table 3 shows the measured frames per second of the application with different number of context switches per frame. With this data we can calculate the context switch time to be around 3 micro-seconds. With a scheduling quantum of 1ms, there would be roughly 1000 context switches per

second with a context switching overhead of 3ms. The overhead of context switches is thus less than 0.3% (3ms per second).

Number of Context Switches/Frame	FPS	Frame Time(1/FPS)
0	518	0.0019
500	466	0.0021
1000	269	0.0037
2000	143	0.0069
3000	92	0.0108
4000	74	0.0135

Table 3: Impact of context switches on observed frame rate.

6.3.2 Scheduling Overhead

To measure scheduler overhead, we ran multiple instances of the same application and measured the frames per second (fps) under DRI and under GERM. The difference in fps gives the overhead of the scheduler. (We saw in the previous section that the context switch overhead is negligible.) Table 4 shows the results for graphics applications and Table 5 shows the same for GPGPU applications. From the results we see that for more CPU intensive graphics applications like Quake 3, the overhead is higher. This can be attributed to the extra copy associated with having the commands in the command queues.

Application	Triangles per frame	Textures	Frames per second				
				1x	2x	3x	4x
Underwater	228	232	DRI	280	140	95	71
			GERM	256	131	89	68
			Overhead	8.6%	6.4%	6.3%	4.2%
Gloss (Mesa demo)	566	17	DRI	196	98	66	49
			GERM	175	87	57	44
			Overhead	10.7%	11.2%	13.6%	10.2%
GearTrain (Mesa demo)	2800	0	DRI	77.8	39		
			GERM	66.7	32.7		
			Overhead	14.3%	16.2%		
Quake 3	6000	2226	DRI	63.1	30.7	20	
			GERM	49.8	24.9	8.7	
			Overhead	21.1%	18.9%	56.5%	

Table 4: Overhead of scheduler (graphics applications).

Application	Size of Instance	Elapsed time (CPU Mcycles)			
			1x	2x	3x
String Comparison	12.2KB	DRI	3.54	7.39	11.61
		GERM	3.57	8.04	12.04
		Overhead	0.85%	8.79%	3.70%
Matrix Multiplication	128x128	DRI	11.96	24.43	35.47
		GERM	12.36	25.5	37.63
		Overhead	3.34%	4.38%	6.09%

Table 5: Overhead of scheduler (non-graphics applications).

6.4 Accuracy of command group estimation

To measure the accuracy of estimation heuristics, we print the command group estimated time and the actual time consumed for each type of heuristic. The heuristics related to textures, buffer swaps and vertices are accurate within 0.1% of the actual time consumed. The heuristic related to command bytes size is not that accurate, >20% of actual time in some cases. This provides a strong case for parsing command groups to develop additional heuristics. But this requires a better knowledge of the GPU commands and architecture. Unfortunately, this information is not available to the general development community currently.

Chapter 7

Conclusion and Future Work

In this thesis, we address the issue of fairness among applications using the GPU. To solve this problem we adapt the deficit round robin algorithm to schedule commands to the GPU. As part of our prototype, we create per-process command queues in the kernel and implement fine-grained scheduling by doing away with hardware locks and have the scheduler handle context switches. We also develop heuristics for estimation of GPU command costs. The content of a command group determines its cost. We use a measurement based approach to determine the coefficients or average cost of a command group for a specific type of command. We then use these coefficients to determine the cost of future command groups. Estimation allows us to implement the deficit round robin algorithm, and ensure fairness among multiple applications.

The current GERM prototype implemented for ATI r200 GPU ensures GPU time variance of less than 5% among competing applications. The overhead of the scheduler is less than 10% for less CPU intensive graphics applications and GPGPU applications. For high-end graphics applications, the overhead is higher because of the additional copy involved in maintaining per-process command queues in the kernel.

Going forward, the GERM prototype can be ported to newer GPUs. These GPUs might provide better feedback (interrupts) to the CPU on completion of command groups thus resulting in a more accurate time measurement of command group execution. ATI recently released an open-source driver [14] for the R500 GPUs and also made the GPU specification public [15]. Though the driver is still very basic, this is an important step in opening up GPU architectures, which would result in better graphics drivers on Linux.

In hindsight, the extra copy associated with maintaining command queues could become a bottleneck if GERM is used for running multiple graphics applications (games). This extra copy can be eliminated by a complete re-design of the user level and kernel level drivers with focus on command buffer management.

One aspect we have not explored in this thesis is that of texture memory management. As we have seen in command group estimation, texture dispatch is costly. There could be a situation where sharing textures between programs is useful. For example, we have two instances of the same program running and they access the same texture. Currently, if there is not enough video memory, existing textures need to be swapped out of video memory and new textures brought in. This is a waste of memory bandwidth if the two textures are the same. Schemes can be developed to recognize this situation and provide ways to share textures when possible.

Bibliography

[1] "General Purpose Computation on Graphics Hardware" http://www.gpgpu.org/s2007/slides/01-introduction.pdf, 2007

[2] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," Computer Graphics Forum 26, March 2007.

[3] "Folding@home project", http://folding.stanford.edu/

[4] 3. B. Langley, "Windows 'Longhorn' Display Driver Model - Details And Requirements," in WinHEC, 2004.

[5] "Compute Unified Device Architecture", http://en.wikipedia.org/wiki/CUDA

[6] "ATI stream SDK", http://ati.amd.com/technology/streamcomputing/sdkdwnld.html

[7] M. J.Kilgard, S. Hui, A. A. Leinwand, and D. Spalding, X Server Multi-rendering for OpenGL and PEX. Silicon Graphics, Inc, June 1996.

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

[9] "Mesa 3D project", http://mesa3d.org/

[10] R. E. Faith, The Direct Rendering Manager: Kernel Support for the Direct Rendering Infrastructure. Precision Insight, Inc., May 1999.

[11] M. Shreedar and G. Varghese, "Efficient Fair Queuing Using Deficit Round Robin," IEEE/ACM Transactions on Networking, 1996.

[12] "ATI R200 GPU", http://www.gpureview.com/ati-r200-chip-28.html

[13] "ATI fragment shader." http://oss.sgi.com/projects/ogl-sample/registry/ATI/fragment_shader.txt, August 2002.

[14] "RadeonHD driver", http://gitweb.freedesktop.org/?p=xorg/driver/xf86-video-radeonhd

[15] "Radeon R5xx Acceleration", http://www.x.org/docs/AMD/R5xx_Acceleration_v1.3.pdf