J Grid Computing manuscript No.
(will be inserted by the editor)

A Resource Selection System for Cycle Stealing in GPU Grids

Y. Kotani · F. Ino · K. Hagihara

Received: date / Accepted: date

Abstract This paper presents a resource selection system for exploiting graphics processing units (GPUs) as general-purpose computational resources in desktop grid environments. Our system allows grid users to share remote GPUs, which are traditionally dedicated to local users who directly see the display output. The key contribution of the paper is to develop this novel system for non-dedicated environments. We first show criteria for defining idle GPUs from the grid users' point of view. Based on these criteria, our system uses a screensaver approach with some sensors that detect idle resources at a low overhead. The idea for this lower overhead is to avoid GPU intervention during resource monitoring. Detected idle GPUs are then selected according to a matchmaking service, making the system adaptive to the rapid advance of GPU architecture. Though the system itself is not yet interoperable with current desktop grid systems, our idea can be applied to screensaver-based systems such as BOINC. We evaluate the system using Windows PCs with three generations of nVIDIA GPUs. The experimental results show that our system achieves a low overhead of at most 267 ms, minimizing interference to local users while maximizing the performance delivered to grid users. Some case studies are also performed in an office environment to demonstrate the effectiveness of the system in terms of the amount of detected idle time.

Y. Kotani · F. Ino · K. Hagihara
Graduate School of Information Science and Technology, Osaka University,
1-3 Machikaneyama, Toyonaka 560-8531, Japan
Tel.: +81-6-6850-6597
Fax: +81-6-6850-6599
E-mail: ino@ist.osaka-u.ac.jp

Keywords cycle stealing, desktop grid, graphics processing unit, resource monitoring, resource selection

1 Introduction

The graphics processing unit (GPU) [10,25] is a single chip processor designed for acceleration of graphics tasks, such as gaming and rendering applications [1]. Modern GPUs [21, 23] are increasing in computational performance at greater than the rate of the CPU [24]. For example, an nVIDIA GeForce 8800 card achieves a peak performance of 330 GFLOPS for single precision data. Typically, this card is expected to achieve ten times faster performance than optimized CPU implementations [24]. In addition to their attractive performance, GPUs are becoming more flexible in programmability with supporting branching and more data types, such as 32-bit floating point and integers. Consequently, many researchers are trying to apply the GPU to non-graphics problems beyond graphics problems [24]. These activities are called as GPGPU [14], which stands for general-purpose computation on GPUs.

On the other hand, grid technology [11] has emerged as a useful framework for sharing computational resources across multiple organizations. This technology enables us to construct a virtual supercomputer over the Internet. Although there are many types of grid systems depending on their purpose, we use the term grid to refer to a desktop grid [5,6], namely a cycle stealing system that utilizes idle computers at home and the office. In such systems, users can be classified into two groups: (1) *resource owners*, namely local users, who contribute their resources to the grid; and (2) *grid users*, who desire to execute their grid applications on donated resources.

The objective of our work is to run GPGPU applications on desktop grid systems. We think that desktop grids will become a more attractive solution for computational scientists if GPUs are explicitly managed and used as general-purpose resources as well as graphics accelerators. We call such enhancing systems as *GPU grids*, which exploit idle GPUs as well as CPUs at home and the office.

Because GPUs are originally designed to serve their owners who directly see the display output, we must resolve some technical issues to share them with remote grid users, who use them through the Internet. One important issue is resource conflicts between resource owners and grid users. In particular, the following problems must be resolved to develop GPU grids.

P1. The lack of criteria for defining idle resources. We must develop criteria for defining idle GPUs, because such criteria have not been considered from the grid users' point of view. The definition here should be considered from both the owner side and the user side in order to (1) minimize interference to resource owners and (2) maximize the application performance provided to grid users.

P2. The lack of external monitors for the GPU. Current operating systems do not have performance information on the GPU. Furthermore, although modern GPUs have performance counters inside their chips, these internal counters are accessible only from instrumented programs running with an instrumented device driver [7]. Therefore, we need an external monitor to minimize modifications to resource configurations and application code.

P3. The lack of efficient multitasking on the GPU. Although nVIDIA has released GeForce 8800 cards, which support context switching in hardware, preemptive multitasking of GPU applications is not available in Windows XP [26], namely the most popular operating system. Instead, multitasking is cooperatively done by software, which results in lower performance. Thus, existing systems are still not virtualized enough to allow multiple GPU applications to run effectively.

Although problem P3 is critical, it is not easy for non-vendors to give a direct solution to this problem, because the details of GPU architecture are not open to the public. Therefore, assuming that the GPU grid consists of cooperative multitasking systems, we tackle the remaining problems P1 and P2 to select idle GPUs appropriately from grid resources.

To address problem P1, we experimentally define the idle state of the GPU. For problem P2, on the other hand, we develop a resource selection system based on a screensaver approach [19] with low-overhead sensors. The sensors detect idle GPUs according to video random access memory (VRAM) usage and CPU usage on each computer. Once idle GPUs are detected, they are further screened for job execution according to a matchmaking framework [27] and benchmark results. The system performs matchmaking between the user requests and the benchmark results obtained automatically when the screensaver is installed on each of resources. This flexible framework allows users to select appropriate resources they want, making the system adaptive to the rapid advance of GPU architecture. Our system is currently running on Windows systems, which fully support the latest GPUs for entertainment use. Though our system is not yet interoperable with existing grid systems, we think that our idea can be applied to screensaver-based systems such as BOINC [2].

The rest of the paper is organized as follows. In Section 2, we introduce related work. In Section 3, we show an overview of the GPU grid with the definition of idle state. In Section 4, we describe our resource selection system. Section 5 shows some experimental results. Finally, Section 6 concludes the paper.

2 Related Work

To the best of our knowledge, there are two projects [31,33] that utilize GPUs as general-purpose resources in desktop grids. Folding@home [31] is a project aiming at accelerating protein folding simulation on GPUs and CPUs at home and the office. Their system achieves approximately 50 TFLOPS and 200 TFLOPS using 900 GPUs and 200,000 CPUs, respectively. Thus, the system demonstrates that GPUs are effective to increase the performance per node, which contributes to achieve higher total performance with fewer computers. However, GPUs are not explicitly monitored in this system. Therefore, users can run multiple CPU/GPU programs on their machine at the same time, resulting in severe slowdowns [31]. Thus, some resource monitoring and selection framework are needed to resolve such performance issues. The other project is Caravela [33], which realizes stream processing in distributed computing environments. Although it utilizes GPUs as general-purpose resources, the main focus in this project is left on security issues, which are related to CPU grids rather than GPU grids. Grid resources must be dedicated to the system to avoid severe slowdowns, because resource monitoring and selection issues are not addressed in this system.

Some grid projects use the GPU as a graphics accelerator to visualize large-scale data in server grid environments [15,18]. In these environments, resources are dedicated to grid users. Due to this dedication, server grids do not cause resource conflicts between resource owners and grid users. Therefore, resources can be easily managed by a central server that receives jobs from grid users and allocates them to grid resources. Similarly, dedicated clusters of GPUs [9,30] can be categorized into these projects.

There are many projects related to desktop grids. Condor [20] is an earlier system that explores using idle time in networked workstations. This system has a central server that polls every two minutes for available CPUs and jobs waiting. Each workstation has a local scheduler that checks every 30 seconds to see if the running job should be preempted because the owner has resumed using the workstation. Thus, owners are interfered for 30 seconds at the worst case. This interfering time is too long for cooperative multitasking systems, which can significantly drop the frame rate of the display.

BOINC [2] is a middleware of the SETI@home project [29], which demonstrates the practical use of desktop grids. This middleware has a screensaver mode that shows the graphics of running applications. Although this mode is useful to know that resource owners currently do not operate their computers, it is not sufficient to conclude that the GPU is in the idle state. Some additional monitors are needed for the GPU.

NVPerfKit [7] is a monitoring tool that allows us to probe performance counters in the GPU. This tool gives us important performance information such as the ratio of idle time to the total measured time. However, it requires modern nVIDIA GPUs with an instrumented version of device drivers to probe the counters. Therefore, this vendor-specific tool is not a realistic solution to our target system, where various GPUs should be monitored without system and code modifications.

Benchmarking tools provide us effective performance information based on direct execution of representative code. For example, 3DMark06 [12] measures GPU performance using a set of three-dimensional (3-D) graphics applications. On the other hand, gpubench [4] focuses on capturing GPU performance for GPGPU applications. Thus, benchmarking tools might be useful to detect GPUs with higher performance. However, they require a couple of time to finish benchmarking. This benchmarking overhead is critical if benchmarking tools are periodically executed for checking resource availability, because resource owners are interfered for a long time by the tools.

With respect to multitasking of GPU applications, Windows Vista supports preemptive multitasking [26]. As compared with cooperative multitasking, preemptive multitasking provides more stable, reliable performance when multiple applications are executed simultaneously on a computer. Therefore, our definition of idle GPUs might be too rigorous in future preemptive systems, because the definition assumes cooperative multitasking systems. However, we think that this assumption is compatible with future systems, because we only have to relax the definition to collect more appropriate resources for such efficient (preemptive) systems.

In contrast to the related work mentioned above, the key contribution of our work is a solution to problems P1 and P2, which leads to efficient execution of GPGPU applications in non-dedicated grid environments. Furthermore, the key difference to our preliminary paper [19] is the evaluation on the latest GeForce 8800 card, which has an entirely different architecture compared with previous cards. Case studies are also presented to show the effectiveness of the system.

3 GPU Grid

The GPU grid has almost the same structure as existing screensaver-based systems. The only difference is that the GPU grid explicitly manages the GPU as general-purpose resources. We think that this small difference is important to integrate our framework into existing grid systems at low effort. Thus, the GPU grid is a subset of the desktop grid. However, the resource selection system should support GPUless computers as well as GPU equipped computers in order to collect more resources in a coordinated manner.

Due to this characteristic, successful applications for the GPU grid must also be successful in the CPU grid. Therefore, the circle of supported applications is bag-of-tasks applications based on a master-worker model. We think that such applications are also suited to the GPU's parallel architecture, because the GPU is allowed to exploit the parallelism of many independent tasks. Actually, the Folding@home project successfully runs such an application on GPUs. In the following discussion, we assume that a grid job consists of many independent tasks.

3.1 System Overview

Figure 1 shows an overview of the GPU grid, which consists of three main components as follows.

- Grid resources. Grid resources are desktop computers at home and the office connecting to the Internet. Ordinarily, these resources are used by resource owners. However, they are donated for task execution if they are in the idle state. Arbitrary computers can be registered as grid resources regardless of having the programmable GPU or not. Grid resources can be protected by the firewall. Therefore, a "pull" mechanism must be deployed to initiate the communication from the resource side.
- The resource manager. The resource manager takes the responsibility for monitoring and selection of registered resources. It also acts as a job manager, which receives jobs from clients. For the problem of task scheduling, we use a round-robin matchmaker [27] that

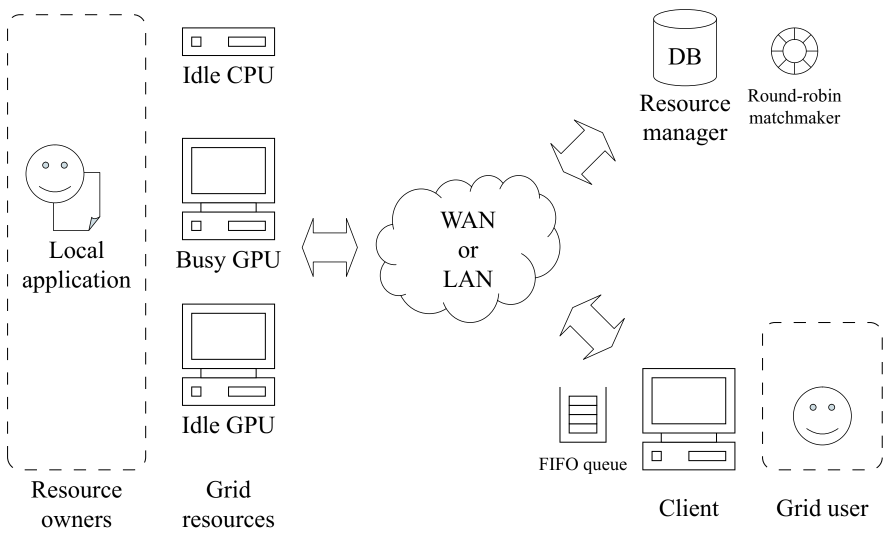


Fig. 1 Overview of the GPU Grid. Grid resources can be protected by the firewall.

equally deals with jobs from different grid users. For each job, the manager performs matchmaking to generate a list of idle resources that could be allocated for execution (see Section 4.2). We accept arbitrary jobs consisting of GPU and CPU applications.

– Clients. Clients are front-end computers for grid users, who want to submit jobs to the grid. Clients can also be registered as grid resources. Each client has a FIFO (first-in first-out) queue to receive jobs from grid users.

Figure 2 presents the interaction between components. Grid jobs are executed in the following steps.

1. Job submission. A job is first submitted to the local queue in the client. The head job of this queue is then submitted to the resource manager. A job here is given by a script file, describing a sequence of program executions. Each execution is regarded an independent task. Jobs are submitted with the application program, the inputs, and a text file describing constraints for resources (Section 4.2).
2. Matchmaking (Section 4.2). The resource manager performs matchmaking to select appropriate resources for task execution. For the monitoring purpose, the selected resources are returned to the client as a list of allocated resources. This matchmaking process is periodically activated for all jobs (each from different users) in a round-robin manner. A period of zero corresponds to an immediate mode that performs matchmaking immediately after receiving a task request from a resource.

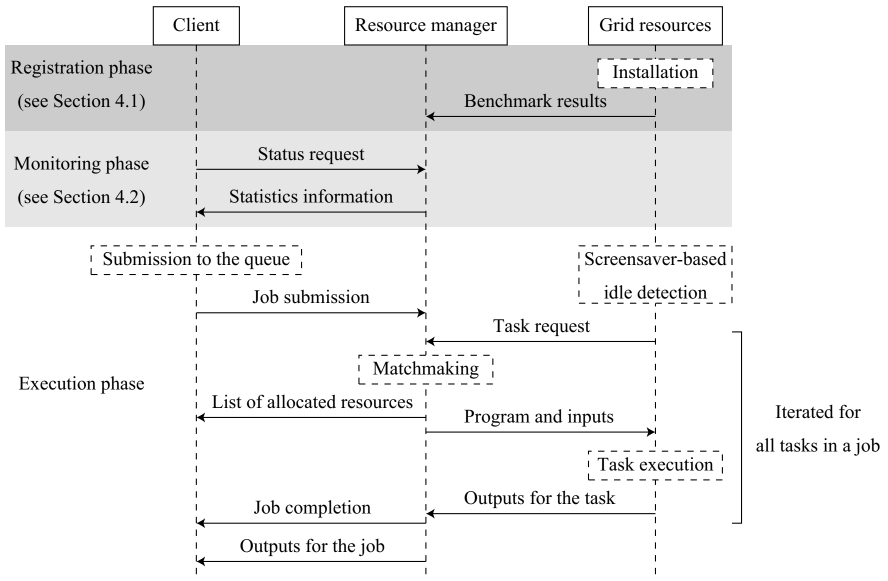


Fig. 2 Interaction between components. The system is based on a "pull" mechanism, which initiates the communication from the resource side.

3. Task execution. Once a resource receives a task from the manager, it downloads the application program and the inputs in order to process the task. After completing the task, it uploads the outputs to the manager and requests the next task. The resource discards the task if it turns to busy.
4. Job completion. After processing all tasks in a job, the resource manager sends a notification to the client. The completed job is then eliminated from the queue to submit the next job to the manager. The outputs can be downloaded from the manager.

In summary, our system is designed based on an integration approach. The pros of this approach are that it allows us to use existing infrastructures, such as grid resources, their owners (communities), and solutions to various important problems, such as user administration, I/O handling, and security issues. Note here that grid resources and their owners can be thought as the core part of grid systems. On the other hand, the cons might be revealed in interactive applications. For example, the graphics pipeline in the GPU is suited to accelerate streaming applications, but such interactive applications will not run effectively in screensaver-based systems.

Table 1 Classification of resource states with owner's activities.

State	CPU	GPU	Owner's activity
1	Idle	Idle	Nothing
2	Busy	Idle	Web browsing, movie seeing, and music listening
3	Idle	Busy	(unrealistic)
4	Busy	Busy	Video gaming

In the following discussion, we use the term *grid application* to denote a program submitted by grid users. We also use the term *local application* to denote a program executed by resource owners on their own resources.

3.2 Definition of Idle Resources

Since a grid resource can have a CPU and a GPU, the resource state can be roughly classified into four groups depending on the state of each processing unit: two groups where both units are idle or busy; and the remaining groups where either one of the units is in the busy state. Table 1 shows this classification with owner's typical activities. In the following, we present the definition of idle resources based on this classification.

As we mentioned in Section 1, the definition must be designed such that it satisfies the following two requirements.

R1. Idle resources must minimize interference to resource owners when they run grid applications.

R2. Idle resources must provide maximum application performance to grid users if they are selected for job execution.

To satisfy the requirements mentioned above, we define an idle resource such that it satisfies all of the following three conditions.

C_1. The resource owner does not interactively operate the resource.

C_2. The GPU does not execute any local application.

C_3. The CPU is idle enough to provide the full performance of the GPU to grid users.

Firstly, condition C_2 excludes states 3 and 4 from the idle state (See Table 1). This condition is essential to satisfy both requirements R1 and R2, because most systems currently do not support preemptive multitasking of GPU applications. If multiple GPU applications simultaneously run on the same resource, grid applications will significantly drop the frame rate of the display output, making resource owners uncomfortable. In addition, local applications will slow down grid applications at the same time. Thus, there can occur significant performance drop both at the owner side and the user side if condition C_2 is not satisfied.

Secondly, due to a similar reason, condition C_1 is needed to minimize interference to resource owners (requirement R1). It excludes state 2 if the resource owner interactively operates the computer through the display output. We also have experimentally confirmed that grid applications suffer from lower performance if the owner gives a window focus to the operating window (see Section 5.2). Therefore, such interactively used computers should be regarded as busy resources also from the grid users' point of view.

Finally, condition C_3 is essential to satisfy requirement R2. GPU applications generally make the CPU usage go to almost 100%, because they usually require CPU intervention during GPU execution. This high usage implies that GPU applications can have a CPU bottleneck. Therefore, CPUs are expected to be idle before job execution. We experimentally show this later in Section 5.2. Note here that condition C_3 might be eliminated in the future, because Windows Graphics Foundation (WGF) 2.0 will enable GPU processing without CPU intervention [3].

4 Resource Selection System

In this section, we describe how our system detects idle resources and how it selects resources for job execution.

4.1 Detection of Idle Resources

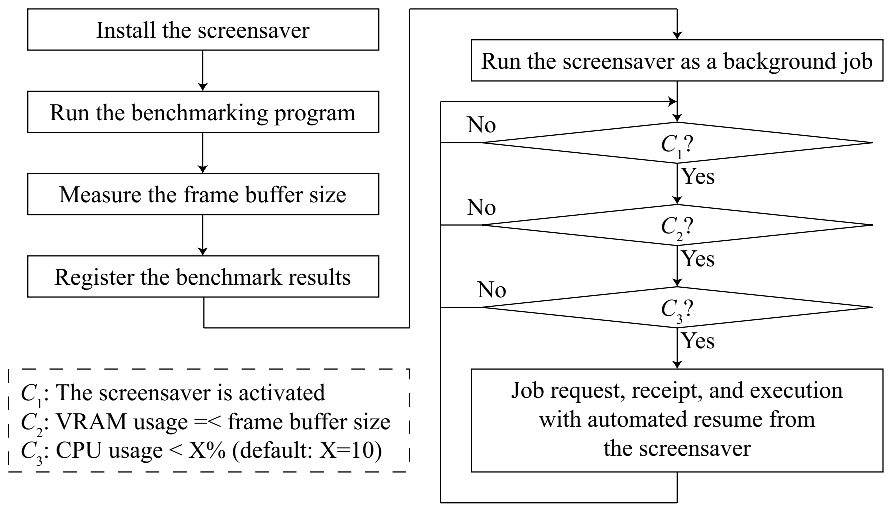
Figure 3 shows the procedure of resource detection. To detect an idle resource that satisfies all conditions C_1, C_2, and C_3, our system checks the resource in the following three steps.

S1. Screensaver check: the screensaver is activated.

S2. VRAM usage check: VRAM usage $\leq$ frame buffer size.

S3. CPU usage check: CPU usage $< X\%$, where X represents a threshold value. The default value is $X = 10$.

Steps S1, S2, and S3 here aim at checking conditions C_1, C_2, and C_3, respectively.



Install the screensaver
Run the benchmarking program
Measure the frame buffer size
Register the benchmark results
Run the screensaver as a background job
No
C_1?
Yes
No
C_2?
Yes
No
C_3?
Yes
Job request, receipt, and execution with automated resume from the screensaver
C_1: The screensaver is activated
C_2: VRAM usage =< frame buffer size
C_3: CPU usage < X% (default: X=10)

Fig. 3 Resource detection procedure processed on grid resources. Steps in the left-hand side are basically performed when the screensaver is installed on the resource. On the other hand, the right-hand side represents the ordinary procedure processed as a background job. The default value of X is experimentally determined as 10.

The first condition C_1 is checked by a screensaver approach that activates the screensaver after five minutes of owner's inactivity. This approach enables us to detect the inactivity at a lower overhead. It also allows owners to rapidly resume their activity, as compared with polling-based methods [20]. Note here that our screensaver avoids updating the display output in order to minimize overhead in cooperative multitasking systems. That is, the frame buffer is updated only when the screensaver is turned on. This intends to keep both the CPU and GPU idle during the screensaver mode, allowing resources to deliver full GPU performance to grid users. The screensaver is implemented using the scrnsave.lib library, which is distributed as a part of Microsoft Visual Studio.

The remaining conditions C_2 and C_3 are checked by sensors at steps S2 and S3, respectively. These sensors are implemented as a part of the ScreenSaverProc function, which is called when the screensaver is activated. Thus, we reduce the monitoring overhead by minimizing the invocation of the sensors.

Checking the VRAM usage is the key idea to evaluate condition C_2 at a low overhead. The idea assumes that the GPU consumes VRAM if it executes any GPU programs. This assumption is valid in the current GPU, which allocates VRAM in advance of execution. Note here that the GPU always consumes VRAM for the frame buffer to refresh the display output. Although the amount of this default consumption is basically determined by the display resolution and its color depth, we have found that it slightly varies depending on hardware and software environments, such as graphics drivers. Therefore, we directly measure the default usage at screensaver installation. Under the assumption mentioned above, we can evaluate condition C_2 by comparing the current VRAM usage to the default usage. If the current value does not equal to the default value, we consider that the GPU is in the busy state. The default value here is also measured at the installation of graphics drivers in addition to the initial installation of the screensaver.

Our VRAM-based monitoring method has two advantages as follows.

- No modification. The VRAM usage can be easily obtained using GetCaps, namely a Direct Draw function [22], which is available in Windows computers. Thus, grid resources do not need any special libraries and hardware except for the screensaver. Furthermore, code modifications are not necessary at the grid users' side.
- Lower overhead. We can know the VRAM usage without GPU intervention because the GetCaps function obtains this information from the device driver. Thus, the VRAM-based method leads to a low-overhead sensor. Note here that GetCaps does not directly give the VRAM usage. This function returns the VRAM capacity and the amount of free space. Therefore, we subtract them to estimate the current usage.

Finally, condition C_3 can be evaluated by accessing CPU usage information provided by the operating system. As same as the VRAM usage, this information does not require GPU intervention. Our implementation calls the PdhCollectQueryData function to access a performance counter in the Windows operating system. We currently use the default value of $X = 10$ according to preliminary experiments (see Section 5.2). However, the threshold value X can be changed by resource owners such that they do not feel interference during job execution.

Although our system is implemented on Windows, it will work on another operating system if it has (1) a screensaver framework that can invoke a program and (2) a device driver and an API function that return the VRAM usage. To the best of our knowledge, such a driver and function are not available on Linux. This might be due to the main market of GPUs. Most of the latest GPUs are installed into Windows PCs to play video games. Similarly, the latest GPUs are designed according to Microsoft's Shader Model, and thus they are always supported in Windows (DirectX [22]) but not fully in Linux (OpenGL [28]).

```
gpu==GeForce 8800 GTX            // GPU model
vram>=512                        // VRAM capacity in MB
os==Windows XP                   // Operating system
cpu_name==Pentium D,Pentium 4    // CPU model
cpu_clock>3.00                   // CPU clock speed in GHz
main_mem>=2048                   // Main memory capacity in MB
gpudriver>=6.14.10.9744          // Driver version
download>900                     // Download performance in MB/s
fpfilltest>8000                  // Drawing performance in Mpixels/s
readback>1000                    // Readback performance in MB/s
```

Fig. 4 A text file describing user's requirement. Our framework performs matchmaking using this file. Resources are specified by attributes, comparison operators, and benchmarking results.

4.2 Selection of Idle Resources

Once idle resources are detected by the screensaver approach, the next issue is the resource selection problem. We resolve this problem by combining two different approaches: a matchmaking approach [27] and a benchmarking approach [4].

The matchmaking approach [27] is responsible for providing a flexible, general framework for resource selection. In this approach, resources are described by attributes such as operating system, main memory capacity, and GPU model. These attributes are then used by grid users to specify the resources they want. This constraint can be written using a formal language that supports operators to specify a group of resources from heterogeneous resources. For example, as shown in Figure 4, the framework enables grid users to select only nVIDIA GeForce 8800 cards with having a fill rate of at least 8 Gpixels/s. We think that this flexible framework is essential to run GPGPU applications in grid environments, because the GPU is still not a matured architecture as compared with the CPU. For example, we have experienced that some applications running on a GPU do not correctly run on different GPUs, due to architectural and driver differences. Therefore, we think that the system must allow users to select appropriate resources they want.

On the other hand, the benchmarking approach [4] takes the responsibility for measuring the effective performance for GPGPU applications. The performance values here are referred by the matchmaking framework for resource selection. The reason why we perform benchmarking is due to the fact that the GPU specification does not always represent the effective performance. Actually, we have found that some middle-range cards outperform high-end cards due to the difference of graphics drivers. Therefore, we run a benchmark program [4] at screensaver installation to obtain the effective performance under the idle state. These benchmark results are then registered at the resource manager to give priorities to detected idle resources. Recall here that benchmarking is performed when installing the screensaver and when updating the graphics driver. Thus, our benchmarking approach avoids long-term interference to resource owners, which we mentioned in Section 2.

The matchmaking framework might not work well if grid users do not have any information on available resources. In this case, some users may put strict constraints to the framework, and thus they might receive a small part of idle resources. Furthermore, they might be enforced to spend time relaxing their constraints to obtain more resources. To deal with this problem, the system provides resource statistics to grid users. The statistics include all attribute values of registered resources and the number of (idle and busy) resources for each value. Though these statistics do not represent precise information on available resources, they will help users in determining appropriate constraints.

5 Experimental Results

In this section, we present some experimental results that evaluate our definition and system using three generations of GPUs. We show that the definition appropriately represents idle resources with satisfying requirements R1 and R2. Furthermore, some case studies are demonstrated to validate the effectiveness of the system in an office environment.

5.1 Setup

Table 2 shows the specification of experimental machines, each with different CPUs and GPUs. We use three desktop computers PC1, PC2, and PC3 running Windows XP. PC1 and PC3 provide the highest and the lowest theoretical performance, respectively. Note here that only the GPU in PC1 supports hardware context switching. It also has an entirely different architecture compared with others. Due to this, we need a different driver for PC1.

For experiments, we use three GPGPU applications: LU decomposition [17], conjugate gradients (CG) [8], and 2-D/3-D rigid registration (RR) [16]. Each application can be briefly summarized as follows.

Table 2 Specification of experimental machines.

Component	PC1	PC2	PC3
CPU	Pentium 4 3.4 GHz	Pentium 4 3.4 GHz	Pentium 4 3.0 GHz
GPU	nVIDIA GeForce 8800 GTX	nVIDIA GeForce 7800 GTX	nVIDIA GeForce 6800 GTO
Core speed (MHz)	575	430	350
Memory speed (MHz)	1800	1200	900
Memory bandwidth (GB/s)	86.4	38.4	28.8
Fill rate (Gpixels/s)	36.8	10.32	4.2
Pipeline engines	128*	24	12
Graphics bus	PCI Express 16X		
Operating system	Windows XP		
Context switch	Hardware	Software	
Driver version	97.44	77.77	

*: 128 stream (scalar) processors.

– LU decomposition of a 2048×2048 matrix. In this implementation, the matrix data is stored as textures in the VRAM. Textures are then repeatedly rendered by the programmable components in the GPU, such as SIMD (single instruction multiple data) and vector processing units. At each rendering step, the CPU determines the working area in textures where the GPU operates.
– CG for solving linear systems with a coefficient matrix of size 64×64. Similar to LU decomposition, this implementation also repeats rendering against textures. However, CG has less CPU workload than LU because the working area is determined by the GPU itself.
– RR for alignment between 2-D images and a 3-D volume. The CPU in this implementation has the lowest workload among the three applications. In contrast, the GPU is loaded more heavily because it operates 3-D data in addition to 2-D images.

The above applications are mainly selected as local applications, which could be graphics applications or GPGPU applications running on a single PC. For GPGPU applications, we adopted numerical applications (LU and CG), because they are typically known as compute-intensive applications. Actually, many GPGPU-related papers has been published in this area [24]. On the other hand, RR is also a GPGPU application but it is similar to graphics applications, because it iteratively renders 3-D objects onto a 2-D screen.

In addition to the GPGPU applications mentioned above, we also use two CPU applications as local applications. One is the PCMark05 benchmark [12], which renders four typical web pages. The other one is the LAME encoder [32], which converts audio files from WAV format to MP3 format. These applications are used to investigate the degree of interference at the owner's side.

5.2 Evaluating Definition of Idle Resources

To validate the definition of idle resources, we investigate the performance of local and grid applications both on idle resources and busy resources. The performance is presented by application throughput, namely the number of executions per second. During performance measurement, each of local and grid applications is executed in an infinite loop to obtain accurate throughputs.

According to the definition in Section 3.2, busy resources can be represented by one of negative conditions $\overline{C_1}$, $\overline{C_2}$, and $\overline{C_3}$. For each condition, we investigate the application performance as follows.

$\overline{C_1}$: The resource owner interactively operates the resource. We run the PCMark05 benchmark [12] and a grid application at the same time. During this simultaneous execution, we measure the throughput of local and grid applications to assess this condition in terms of requirements R1 and R2. Because PCMark05 renders various web pages, this experiment measures the interference to owners assuming that they are browsing web pages during job execution.

$\overline{C_2}$: The GPU executes local applications. We simultaneously run two GPGPU applications as local and grid applications. We then measure the throughput of grid applications. This experiment mainly intends to assess how grid applications are interfered by local applications.

$\overline{C_3}$: The CPU is not idle enough to provide the full performance of the GPU to grid users. We measure the throughput of grid applications with varying CPU usages from 0% to 100%. We also measure the throughput while running the MP3 encoder [32], which intensively uses CPU resources.

Figure 5 shows the throughput of grid applications with different CPU usages, giving results under condition $\overline{C_3}$. For all applications, we can see that the throughput basically

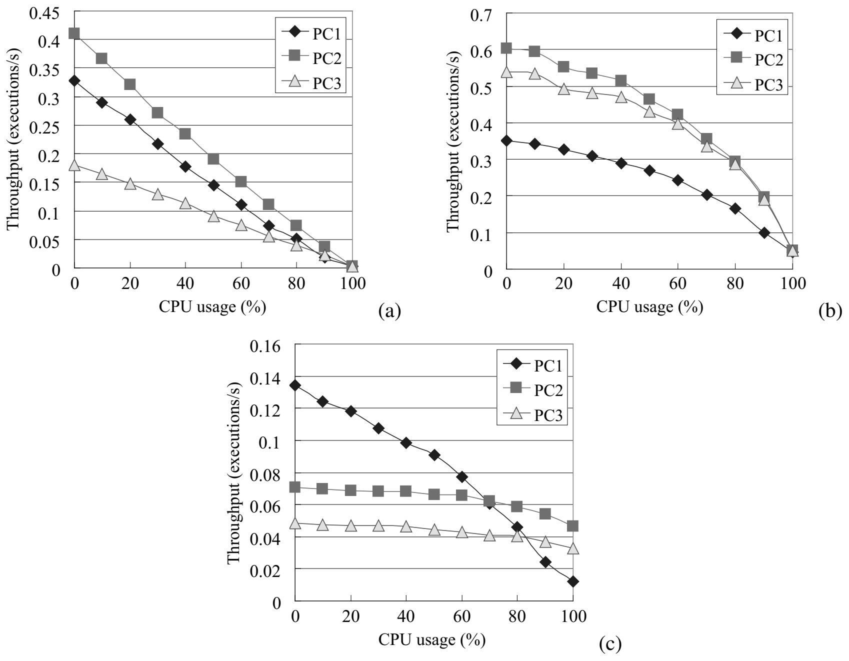


Fig. 5 Measured throughput of grid applications with different CPU usages. Three GPGPU applications are executed as grid applications: (a) LU decomposition, (b) conjugate gradients (CG), and (c) rigid registration (RR). Throughput is presented in executions per second.

decreases as the CPU usage increases. One remarkable point here is that the CPU usage significantly affects the application performance if the application has a CPU bottleneck. For example, LU linearly drops the performance, whereas RR slowly decreases the performance on PC2 and PC3. This is due to the difference of workload characteristics inherent in GPGPU applications. That is, LU requires more CPU intervention during execution, because it frequently switches textures as compared with RR. It also frequently transfers more data between the CPU and the GPU. Therefore, LU linearly drops the performance as compared with RR running on PC2 and PC3. Note here that PC1 shows different performance behavior in Figure 5(c). In particular, the latest GeForce 8800 card becomes slower than older cards when the CPU usage is greater than 70%. We could not clearly identify the reason for this behavior, but one possible answer could be given by the underlying graphics driver. As compared with older cards, this card has an entirely different architecture. For example, it employs unified shaders, scalar processing units, and so on. Therefore, the graphics driver might still not be matured well enough to run programs efficiently in severe situation. Ac-

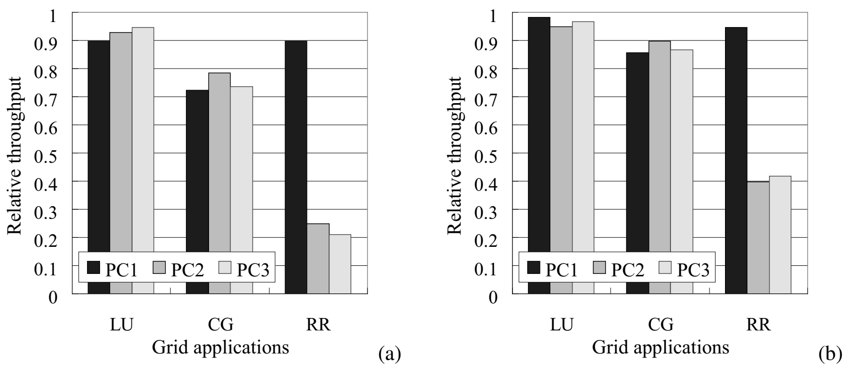


Fig. 6 Relative throughput provided to local applications: (a) a web browser (PCMark05) and (b) an MP3 encoder (LAME). Applications in the horizontal axis represent grid applications running as a background job. The relative throughput is normalized to the original throughput measured without running any grid applications. Higher values indicate smaller interference to resource owners.

cording to these results, we think that idle resources are allowed to have some CPU tasks if the tasks do not intensively use the CPU. Thus, we have determined that idle resources can have a CPU usage of at most X (step S3). The default value of $X = 10$ is mainly determined according to the amount of detected idle time, as shown later in Section 5.4.

The other interesting point in Figure 5 is that PC1 with the latest GeForce 8800 card fails to achieve the highest results for LU and CG, due to the latency of the GPU pipeline. We find that the GeForce 8800 card has the longest latency though it has the highest bandwidth in the three graphics cards. This performance characteristic increases the GPU time in LU and CG, which iteratively render small region. In such applications, the entire performance can be determined by the latency rather than the bandwidth. Note here that the performance is also affected by the CPU, as we mentioned before. Similar results are observed in another implementation [13] of LU decomposition. Thus, high-end cards do not always give the highest performance for GPGPU applications if the pipeline latency determines the entire performance. Our flexible system is also motivated by this result.

Figure 6 shows the relative throughput P_2/P_1 of two local applications: (a) the PC-Mark05 benchmark and (b) the LAME encoder. P_1 here represents the original throughput measured without running grid applications while P_2 represents the throughput lowered by simultaneous execution of local and grid applications. Therefore, a relative performance of 1 means that the grid application does not slow down the local application. As shown in Figure 6, we observe approximately 20% slowdowns in most cases. This lower interference is mainly due to the window focus given to the local application. In these cases, PCMark05 and LAME is processed with a higher priority than grid applications. Therefore, the slowdowns are not so critical in these cases. However, we find that PC2 and PC3 suffer for more than 50% performance drop if RR is executed as a grid application. For example, the rendering performance of web pages is decreased from approximately 2 to 0.5 pages/s. Recall here that RR shows different behavior on these two machines in Figure 5. That is, RR has a GPU bottleneck even if the local application intensively uses the CPU at the same time. Therefore, although the window focus is given to local applications, the GPU is loaded enough to slow down local applications that wait for the GPU to update the display output (frame buffer). Thus, resources in condition $\overline{C_1}$ should not be used for job execution, because grid applications can cause significant interference to resources owners. To avoid this interference, we need step S1.

Figure 7 now presents the degree of slowdowns from the grid users' point of view. It shows the relative throughput P_4/P_3 of grid applications, where P_3 represents the original performance on dedicated machines and P_4 represents the lowered performance under simultaneous execution. In this figure, we can see that PM and EC significantly reduce the performance of grid applications. That is, if the PCMark05 benchmark or the LAME encoder is executed as a local application, only 20% throughput is achieved in most cases. However, as shown in Figure 7(c), PC2 and PC3 again show different behavior if RR is executed as a grid application. The slowdown is reduced to less than 50%, because RR is robust to intensive use of the CPU, as we presented in Figure 5(c). In summary, grid applications significantly drop their performance if owners run CPU applications on their computers with the window focus.

In Figure 7, we also can see results obtained by running two GPGPU applications at the same time. Figure 7(a) indicates that the performance of LU is reduced to appropriately 20% if the resource owner executes any GPGPU applications. This is also due to the window focus, which lowers the priority of grid applications. Actually, the slowdown of RR in Figure 7(c) is not so serious because RR is robust to lower prioritization. Another important point is the effect of preemptive multitasking in PC1. Although this effect is not clearly shown in Figures 6 and 7, we found that the preemptive mechanism realizes true multitasking of local and grid applications. In contrast, PC2 and PC3 serially run local and grid applications. That is, in such cooperative multitasking systems, grid applications are kept waiting until the completion of local applications. This serialization problem is critical if local applica-

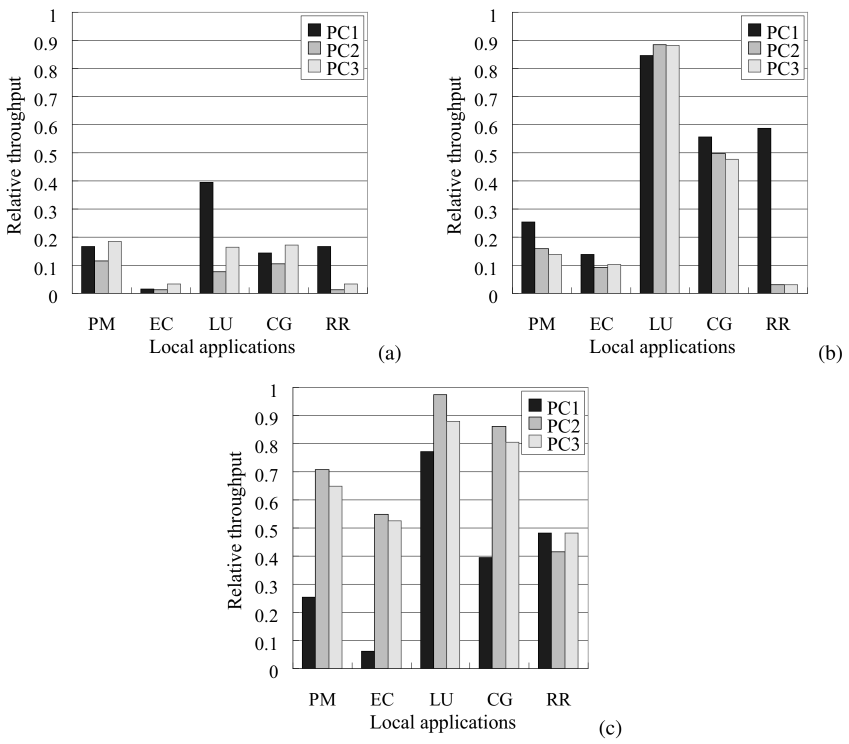


Fig. 7 Relative throughput provided to grid applications: (a) LU, (b) CG, and (c) RR. Applications in the horizontal axis represent local applications running at the same time. PM and EC represent the PCMark05 benchmark and the LAME encoder, respectively. Higher values indicate that higher performance is delivered to grid users.

tions need long time for completion. Note here that this problem is also critical for resource monitoring systems if the systems require GPU intervention during monitoring. Thus, we think that condition C_2 is needed to define idle resources.

In summary, we think that the definition is reasonable in terms of minimizing interference to resource owners while maximizing the application performance provided to grid users.

5.3 Evaluating Overhead of Resource Selection

We now evaluate the monitoring overhead of our system. Table 3 shows the execution time of local applications, explaining how these applications are perturbated by the resource monitoring overhead. In experiments, we use LU, CG, and RR as local applications. For each

Table 3 Perturbation time measured using three local applications. T_1 and T_2 represent the execution time of the local application without resource monitoring and that with monitoring, respectively. $T_2 - T_1$ represents the perturbation time caused by monitoring. Times are presented in seconds.

Local	PC1 (s)			PC2 (s)			PC3 (s)		
application	T_1	T_2	$T_2 - T_1$	T_1	T_2	$T_2 - T_1$	T_1	T_2	$T_2 - T_1$
LU	3.049	3.256	0.207	2.441	2.540	0.099	5.558	5.670	0.112
CG	2.850	2.960	0.110	1.670	1.780	0.110	1.859	2.075	0.216
RR	7.446	7.648	0.202	14.159	14.470	0.311	18.590	18.790	0.200

PC, we first measured the original time T_1 with disabling resource monitoring, and then obtained time T_2 with enabling monitoring. Therefore, the perturbation time $T_2 - T_1$ explains how long the monitoring system perturbates local applications. To measure time T_2, we iteratively ran a local application using a script with an infinite loop, and then invoked the screensaver program using our system. Each of the execution time in Table 3 is an average value from 20 executions.

We observe the highest perturbation time of 311 ms when executing RR on PC2. However, this time is short enough as compared with the entire execution time T_1. This perturbation time is mainly due to the monitoring overhead of 267 ms: 194 ms for activating the screensaver at step S1; 2 ms for checking the VRAM usage at step S2; and 71 ms for accessing the CPU usage at step S3. The breakdown here is directly measured by instrumenting the screensaver and sensor programs. Although the screensaver activation takes 194 ms, it does not cause significant interference to resource owners, because the activation guarantees the owner's inactivity. Furthermore, our monitoring system minimizes interference to GPU applications, because it is implemented as a small CPU program. Thus, the perturbation time is much smaller than the results in Figure 7, presenting the effectiveness of our monitoring system.

One concern is the interference to GPGPU applications that do not require interaction with owners, because such states cannot be screened at step S1. However, this is not so critical because the screensaver updates the frame buffer only at activation. Therefore, the GPU is fully served to local applications after the activation. The remaining time for checking the VRAM and CPU usages is also performed in short time, because it is fully processed at the CPU side. Thus, local applications are perturbated only when the screensaver is activated. Therefore, as shown in Table 3, our system achieves a low-overhead monitoring with minimum interference.

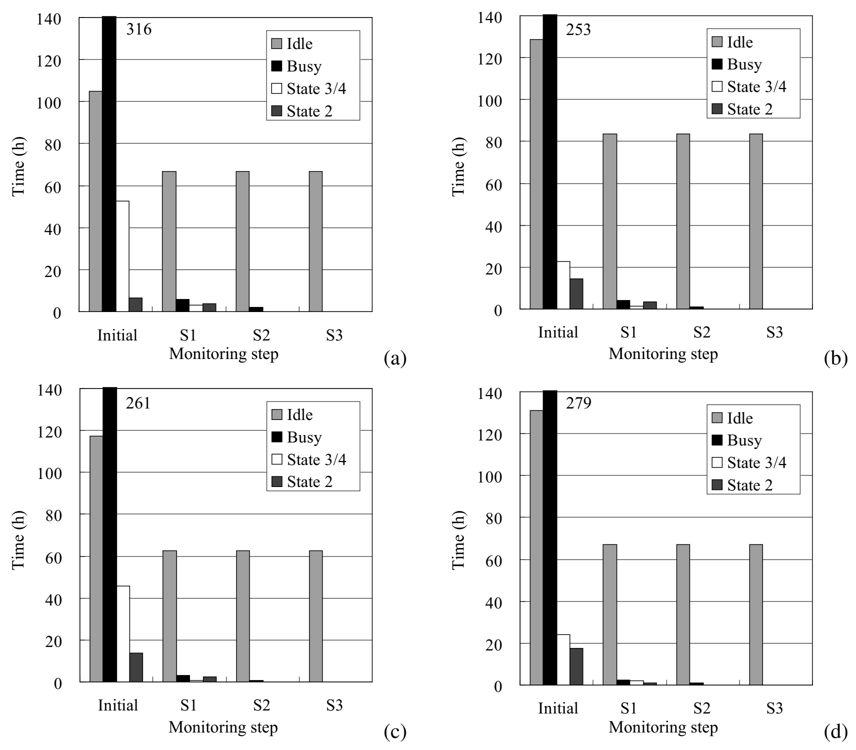


Fig. 8 Classification of resource states at each of monitoring steps S1, S2, and S3. Each subfigure represents the classification for an owner. Time is presented in hours. Classification is done according to conditions C_1, C_2, and C_3.

5.4 Case Study

Finally, we show the amount of idle time detected by the proposed system. We have installed the screensaver on four desktop computers in our laboratory to monitor them for 60 days. Their owners are four graduated students performing research on GPGPU applications. Note here that computers are powered off at night and over a weekend. Such offline time is not included in idle time in the following results.

Figure 8 shows the classification of resource states at each of monitoring steps. It explains how the system screens out busy states 2, 3, and 4 in the four computers (see Table 1). To present an accurate breakdown of states, this classification is done by additional sensors that monitor the resources at one minute intervals. In this figure, we can see that the proposed system detects idle time of approximately 60 hours out of 120 hours. Thus, the system detects 51–65% of the total idle time. The remaining 35–49% of idle time could not

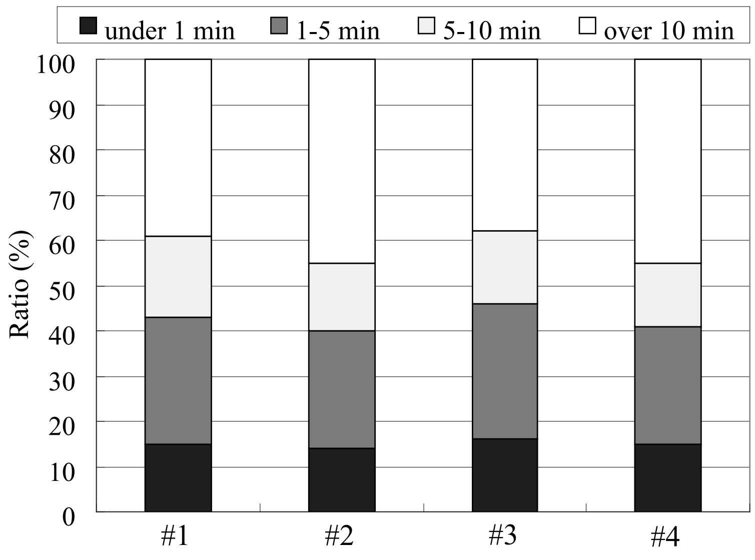


Fig. 9 Breakdown of idle time with four different owners. The breakdown is done according to the length of idle time.

be detected due to the timeout of five minutes needed before screensaver activation. That is, the system failed to detect short idle period of less than five minutes. Therefore, reducing this timeout will increase the amount of detected time but also increase interference to resource owners.

We also can see that step S1 screens out most of the busy states. For example, the busy time is reduced from 316 hours to 6 hours at step S1 in Figure 8(a). This means that 98% of busy time is effectively detected by the screensaver. The remaining 2% of busy time is also detected by the succeeding steps S2 and S3. Thus, although the screensaver is useful to detect idle resources, it is not sufficient to avoid resource conflicts between users and owners. Our system resolves this problem by processing steps S2 and S3 in addition to step S1.

Figure 9 shows a breakdown of the idle time, explaining how long the resources keep the idle state. This figure indicates that once a resource becomes idle, it possibly keeps the idle state for more than ten minutes. Actually, the average length of the idle state ranges from 12 to 20 minutes, as shown in Table 4. According to this table, the computers used in this study become idle five times a day, each with more than ten-minute length. Although this result depends on owners, this information is useful to determine the granularity of a grid task.

Table 4 Number of screensaver activations and average length of activations. Average time is presented in minutes.

Owner	Number of activations	Average length (m)
#1	340	13
#2	305	17
#3	344	12
#4	338	20

On the other hand, GPGPU applications generally complete their execution in less than ten minutes due to the limitation on VRAM capacity. Therefore, we think that this average idle time is long enough to construct a grid task.

Finally, we discuss on the default value of threshold X. We further classified resource states in terms of CPU usage. We then found that 97% of resource states have CPU usage of at most 10% after step S2. Therefore, even if we increase the threshold to 20%, the amount of idle time increases only by 1%. Thus, we think that the default threshold of $X = 10$ is reasonable in terms of minimizing interference to resource owners while maximizing application performance provided to grid users.

6 Conclusion

We have presented a resource selection system for the GPU grid, which aims at running GPGPU applications in desktop grid environments. We also have shown criteria for defining idle resources in the GPU grid. Our system is designed for cooperative multitasking systems, but it also works on preemptive multitasking systems. In order to detect idle resources, the system employs a screensaver approach with low-overhead monitors, which are designed to detect idle GPUs without GPU intervention. After this detection, the system selects the resources for job execution by performing matchmaking between the user requirements and the benchmark results. This matchmaking mechanism allows users to select appropriate resources they want, making the system flexible enough to the rapid advance of GPU architecture.

The experimental results show that the definition is reasonable with minimizing interference to resource owners while maximizing the application performance provided to grid users. We also find that the system achieves a low overhead of at most 267 ms, which is short enough as compared to the execution time of local applications. Actually, we observe that the perturbation time is less than 311 ms. The case study demonstrates that our screensaver-based system effectively detects idle resources in an office environment, which possibly keep the idle state for more than ten minutes.

One future work is to develop a mechanism that automates the generation of resource requirements used for matchmaking. This mechanism will help users in selecting appropriate resources, freeing them from understanding which resources provide higher performance to their applications. We are also planning on integrating our system with existing screensaver-based systems to perform evaluation in a distributed environment. We think that much higher performance is still left in the home because nVIDIA has shipped over 40 million G80 cards[1].

Acknowledgements This work was partly supported by JSPS Grant-in-Aid for Scientific Research for Scientific Research (B)(2)(18300009) and for Young Researchers (19700061).

References

1. Akenine-Möller, T., Haines, E. (eds.): Real-Time Rendering, second edn. Morgan Kaufmann, San Mateo, CA (2002)
2. Anderson, D.P.: BOINC: A system for public-resource computing and storage. In: Proc. 5th IEEE/ACM Int'l Conf. Grid Computing (GRID'04), pp. 4–10 (2004)
3. Blythe, D.: Windows graphics overview. In: Windows Hardware Engineering Conf. (WinHEC'05) (2005). `http://www.microsoft.com/whdc/winhec/Pres05.mspx`
4. Buck, I., Fatahalian, K., Hanrahan, P.: GPUBench: Evaluating GPU performance for numerical and scientific applications. In: Proc. 1st ACM Workshop General-Purpose Computing on Graphics Processors (GP^2'04), p. C–20 (2004)
5. Cappello, F., Djilali, S., Fedak, G., Herault, T., Magniette, F., Néri, V., Lodygensky, O.: Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid. Future Generation Computer Systems **21**(3), 417–437 (2005)
6. Chien, A., Calder, B., Elbert, S., Bhatia, K.: Entropia: architecture and performance of an enterprise desktop grid system. J. Parallel and Distributed Computing **63**(5), 597–610 (2003)
7. nVIDIA Corporation: NVPerfKit 2.1 User Guide (2006). `http://developer.nvidia.com/object/nvperfkit_home.html`
8. Corrigan, A.: Implementation of conjugate gradients (CG) on programmable graphics hardware (GPU) (2005). `http://www.cs.stevens.edu/~quynh/student-work/acorrigan_gpu.htm`
9. Fan, Z., Qiu, F., Kaufman, A., Yoakum-Stover, S.: GPU cluster for high performance computing. In: Proc. Int'l Conf. High Performance Computing, Networking and Storage (SC'04) (2004)

[1] Keynote talk by Sanford Russell in Research and Industrial Collaboration Conference 2007.

10. Fernando, R. (ed.): GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Addison-Wesley, Reading, MA (2004)
11. Foster, I., Kesselman, C. (eds.): The Grid: Blueprint of a New Computing Infrastructure. Morgan Kaufmann, San Mateo, CA (1998)
12. Futuremark Corporation: Products (2006). `http://www.futuremark.com/products/3dmark06/`
13. Galoppo, N., Govindaraju, N.K., Henson, M., Manocha, D.: LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In: Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'05) (2005). 12 pages (CD-ROM)
14. GPGPU: General-Purpose Computation Using Graphics Hardware (2007). `http://www.gpgpu.org/`
15. Grimstead, I.J., Avis, N.J., Walker, D.W.: Automatic distribution of rendering workloads in a grid enabled collaborative visualization environment. In: Proc. Int'l Conf. High Performance Computing, Networking and Storage (SC'04) (2004). 10 pages (CD-ROM)
16. Ino, F., Gomita, J., Kawasaki, Y., Hagihara, K.: A GPGPU approach for accelerating 2-D/3-D rigid registration of medical images. In: Proc. 4th Int'l Symp. Parallel and Distributed Processing and Applications (ISPA'06), pp. 769–780 (2006)
17. Ino, F., Matsui, M., Hagihara, K.: Performance study of LU decomposition on the programmable GPU. In: Proc. 12th Int'l Conf. High Performance Computing (HiPC'05), pp. 83–94 (2005)
18. Jankun-Kelly, T., Kreylos, O., Ma, K.L., Hamann, B., Joy, K.I., Shalf, J., Bethel, E.W.: Deploying web-based visual exploration tools on the grid. IEEE Computer Graphics and Applications **23**(2), 40–50 (2003)
19. Kotani, Y., Ino, F., Hagihara, K.: A resource selection method for cycle stealing in the GPU grid. In: Proc. 4th Int'l Symp. Parallel and Distributed Processing and Applications Workshops (ISPA'06 Workshops), pp. 939–950 (2006)
20. Litzkow, M.J., Livny, M., Mutka, M.W.: Condor - a hunter of idle workstations. In: Proc. 8th Int'l Conf. Distributed Computing Systems (ICDCS'88), pp. 104–111 (1988)
21. Luebke, D., Humphreys, G.: How GPUs work. Computer **40**(2), 96–100 (2007)
22. Microsoft Corporation: DirectX (2007). `http://www.microsoft.com/directx/`
23. Montrym, J., Moreton, H.: The GeForce 6800. IEEE Micro **25**(2), 41–51 (2005)
24. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum **26**(1), 80–113 (2007)
25. Pharr, M., Fernando, R. (eds.): GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley, Reading, MA (2005)
26. Pronovost, S., Moreton, H., Kelley, T.: Windows display driver model (WDDM) v2 and beyond. In: Windows Hardware Engineering Conf. (WinHEC'06) (2006). `http://www.microsoft.com/whdc/winhec/Pres06.mspx`
27. Raman, R., Livny, M., Solomon, M.: Matchmaking: An extensible framework for distributed resource management. Cluster Computing **2**(2), 129–138 (1999)
28. Shreiner, D., Woo, M., Neider, J., Davis, T.: OpenGL Programming Guide, fifth edn. Addison-Wesley, Reading, MA (2005)

29. Sullivan, W.T., Werthimer, D., Bowyer, S., Cobb, J., Gedye, D., Anderson, D.: A new major SETI project based on project serendip data and 100,000 personal computers. In: Proc. 5th Int'l Conf. Bioastronomy, p. 729 (1997)

30. Takizawa, H., Kobayashi, H.: Hierarchical parallel processing of large scale data clustering on a PC cluster with GPU co-processing. The J. Supercomputing **36**(3), 219–234 (2006)

31. The Folding@Home Project: Folding@home distributed computing (2007). `http://folding.stanford.edu/`

32. The LAME Project: LAME MP3 Encoder (2007). `http://lame.sourceforge.net/`

33. Yamagiwa, S., Sousa, L.: Design and implementation of a stream-based distributed computing platform using graphics processing units. In: Proc. 4th Int'l Conf. Computing Frontiers (CF'07), pp. 197–204 (2007)