Arbitration of Legacy VGA Interface for Multiple Video Cards

Tiago Vignatti[1], **Paulo R. Zanoni**[1], **Ander C. de Oliveira**[1], **Luis C. E. de Bona**[1], **Fabiano Silva**[1]

[1] Universidade Federal do Paraná
Centro de Computação Científica e Software Livre
Caixa Postal 19.081 – CEP 81.531-980 – Curitiba – PR – Brasil

{vignatti, paulo, ander, bona, fabiano}@c3sl.ufpr.br

Abstract. *When multiple applications use the legacy Video Graphics Array (VGA) interface to communicate with different graphics devices attached to the same machine, problems inherited by the old Industry Standard Architecture (ISA) system happen. If the accesses to these graphics devices are not correctly controlled, an access made using that interface might be decoded by the wrong graphics card. Since the Peripheral Component Interconnect (PCI) and the Accelerated Graphics Port (AGP) standards are compatible with ISA, this problem remains in most modern devices. This paper presents a solution independent of the application to this problem. It consists of a conceptual architecture, which defines an arbiter that manages all the VGA accesses and an user space interface used by the applications. An implementation using the Linux Kernel and the X server is also presented. The results of the experiments show that there is no significant performance loss.*

1. Introduction

Most machines today have only one graphics device, which is very suitable for the majority of users activities. This setup is highly used and well supported since it is essential for all desktops. Machines with more than one video card[1] are not that common and do not have the same support. When multiple applications use the legacy VGA interface to communicate with different graphics devices attached to the same machine, problems inherited by the old ISA system happen [Shanley and Anderson 1999]. If the accesses to these graphics devices are not correctly controlled, an access made using that interface might be decoded by the wrong graphics card. Since the PCI and AGP standards are compatible with ISA, this problem remains in most modern devices.

The application used to control the graphics and to display bitmap windows in many operating systems is the X server through the *X Window System* [Angebranndt et al. 2004, Scheifler and Gettys 1997]. In the late 1990s a module inside the open-source implementation of the X server has been added to control multiple graphics devices, which solved most of the problems concerning multiple video cards [X.Org Foundation]. However, the problems remain for setups where there are multiple applications trying to communicate concurrently with different devices.

[1] In this paper "video card", "graphics device" and "VGA device" are different terminologies to denote the same.

One relevant example of a setup that needs a correct VGA interface arbitration is the multiseat model of computation. A multiseat (or multiterminal) is a computer with multiple video cards, keyboards and mice, that allows multiple users to run independent sessions. Ideally, there should be one instance of the X server for each user. Since this is not possible due to the legacy VGA interface, other solutions that use only one instance of the X server are being deployed, facing some problems such as the lack of video acceleration and great overhead because they use nested[2] X servers [Oliveira et al. 2006, Oliveira et al. 2008, Zanoni et al. 2008]. Multiseat has a huge importance in terms of social responsibility and cost reduction. For instance, the Paraná Digital project [Castilho et al. 2006] is bringing about 11000 multiseats (totalizing 44000 terminals) to public schools in Brazil. Unfortunately all of these terminals are relying on that approach using one instance of the X server to controls all devices cards, thus under-using the hardware.

This paper introduces the *VGA Arbiter*, an arbiter that controls all the accesses that use the legacy VGA interface on the video cards of a single machine. It also presents an user space interface to communicate with the arbiter. An implementation of this conceptual architecture including a module inside the X server is shown, which is based on the work started in 2005 by Benjamin Herrenschmidt [Herrenschimdt a]. Although the current implementation of the arbiter is a Linux Kernel module, it can be easily implemented in other operating systems or even as an operating system independent user space application.

In the remainder of this paper, Section 2 explains in details the problems inherent to the PCI bus specification and the current solution deployed by the X server. Section 3 shows the conceptual architecture of the VGA Arbiter and Section 4 shows details of the three pieces of its implementation. Experiments were evaluated and shown in Section 5. Finally, Section 6 presents the considerations and new directions of this research.

2. PCI Issues

The peripheral devices have registers that are used to control them. These registers are the interface between the device and the CPU. They are mapped to memory or IO, depending on the computer architecture, so everything written in those memory or IO positions is actually written in the registers of a device [Corbet et al. 2005].

On the PCI system, the addresses of the registers can be dynamically assigned, so they can be mapped to any memory or IO position, allowing the use of many PCI devices without memory overlapping (as long as the implementation is correct). This assignment is done using a register called Base Address Register (BAR) [Shanley and Anderson 1999]. On the ISA system, this is different. The ISA devices are mapped to fixed physical addresses, not dynamically assigned. Some of these addresses correspond to legacy VGA registers, used by video cards to implement the VGA specification mostly used to do basic operations such as the initialization of the cards [Lo et al. 2005].

The PCI system is compatible with the ISA system, so its devices implement the

[2]A nested X server uses as input and output devices another X server. Therefore the nested X server delegates all the hardware access to the responsability of the underlying X server.

ISA features. For this reason, the devices have to decode those physical addresses defined by the ISA specification, including the video cards that decode the legacy VGA registers.

The PCI and ISA specifications also define that the devices have registers that can be used to control whether they will decode all the IO and memory accesses on them or not. There are also registers in the PCI bridges that are used to control the legacy VGA decoding.

For instance, in a system with multiple video cards that decode the legacy VGA interface mapped to fixed physical addresses, when an access is made to one of those addresses, more than one card will decode it. If there is only one application accessing these cards, it can simply use the explained registers to enable the VGA decoding on only the desired card and disable it on all the others. This is exactly what the X server does so far to allow the control of multiple video cards.

The idea implemented in the X server is referred to as Resource Access Control (RAC) [X.Org Foundation]. With this solution, if multiple graphics devices are needed and they can share the same X server, they get controlled by RAC inside this application and all the accesses are decoded by the right cards, so problems do not happen. On the other way, when more than one application needs to access the legacy VGA interface, an entity that has control over all the accesses is needed. The hole of this entity can be seen as a resource broker so that applications can communicate to it and access the bus only when the desired card will decode its access, otherwise cards will decode access that are not meant to be decoded and will get an inconsistent state. The next section presents a possible design of such entity.

3. Conceptual Architecture

The VGA arbiter is a semaphore that coordinates the accesses to the graphics devices made through the legacy VGA interface. Every application or Kernel code that uses this interface to access a video card must use the arbiter. To simplify and abstract the way the applications access it, an user space interface was proposed. Figure 1 shows a diagram of this architecture.

3.1. The Arbiter

The arbiter is the entity responsible for changing registers of the graphics devices and the PCI bridges to guarantee that when an access using the VGA legacy interface is made, only the desired device will decode it. When an application needs to use this interface, it must first ask the arbiter, specifying which video card is being used and what kind of resources it decodes, then the arbiter will set up the proper registers in the motherboard and give the access to the application, allowing it to safely communicate with the video card. When the application finishes the access, it must give it back to the arbiter, so that other applications will also be able to use the legacy VGA interface. In short words, the arbiter works like a semaphore, allowing one application at a time.

If some video card does not use that interface, or can stop using it at any time, the application that controls it can tell it to stop decoding the VGA accesses, and then tell the arbiter that it does not decode it. This way, it will be removed out of arbitration, which might generate some performance increase, depending on the implementation.

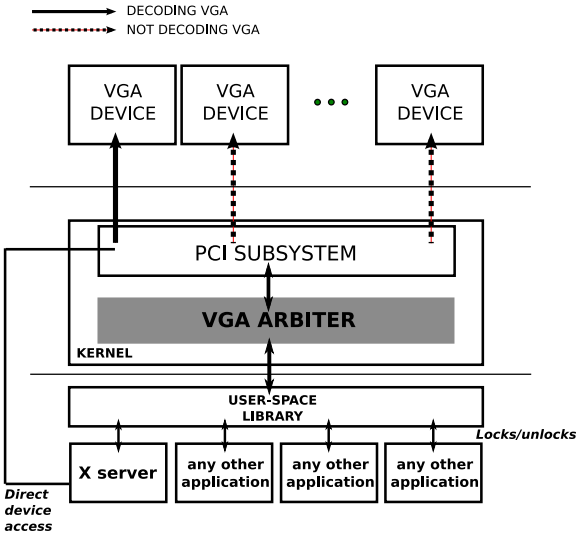

Figure 1. The *VGA Arbiter* coordinates the legacy VGA instructions.

3.2. User Space Interface

Every application and Kernel code that access devices using the legacy VGA interface will have to be changed to use the arbiter. They also have to be implemented correctly to only acquire the locks when needed and give them back to the arbiter as soon as they finish communicating with the device. If a single application starts communicating with the graphics devices without the arbiter, the problems explained will still happen.

4. Implementation

This section explains the details of the implementation of the VGA arbiter, its user space library, and an implementation of a client that uses it, the X server.

4.1. The Arbiter

On the current Linux implementation (Kernel version 2.6.23.12), both the Kernel code and some applications make use of the legacy VGA interface. Since all the accesses made using it must be managed by the arbiter, implementing it as an application would require the Kernel code to access the application before using the interface. Considering this, a built-in Kernel implementation seems reasonable. Despite this fact, the current implementation is not a Linux Kernel built-in, but a Linux Kernel module. It does not allow the Kernel code to use it in an easy way, but it is simpler and faster to build and test, since it does not require a whole Kernel compilation and a machine reboot every time the code is changed. Also, it is still a proof of concept, so when it becomes more stable, moving it to inside the Kernel will be simple.

The advantage of making an user space application instead of a Kernel module is that it could be system independent. It could share a lot of the operating system dependent code implemented inside the X server, which would make its implementation easier. The problem is that, as mentioned, it would require communication between it and the clients and also the Kernel, which is less efficient.

The current implementation provides a character device node for the user space applications, and still no interface for the code inside the Kernel. It is already defined, but

will only be provided when the VGA Arbiter becomes part of it. The current interface for the character device and for the Kernel is based on the implementation proposed in 2005 by Benjamin Herrenschmidt [Herrenschimdt a] after a discussion at the X.org mailing list [Herrenschimdt b]. The implementation is also an improvement on this original code.

To use the arbiter through the device node, an application has to: (i) open the device node, (ii) tell which PCI device it is using, (iii) tell what kind of resources it decodes (legacy memory, legacy IO, normal memory, normal IO) and (iv) start calling the lock and unlock functions. There are two kinds of locking functions, one that is blocking and other that is not, which returns a different value on failure. When it closes the file descriptor, the Kernel automatically unlocks everything that was locked by the client. If an application wants to use multiple video cards, it has to open the node more than once (although this is transparent if it is using the library, and not the device node directly). The actual messages sent through the file descriptor are very simple strings. The Kernel code just keeps comparing strings to find out what it needs to do.

Considering the current interfaces, there is a big field for improvements. The device node protocol could use smaller strings, which would reduce the time the module stays comparing them (although this would make it less user friendly). Or there could be used even a faster method than a device node.

4.2. User Space Library

The library is very simple and can operate multiple file descriptors transparently, so that its clients will be able to use multiple video cards. There is not too much space for optimization, given its simplicity.

4.3. X Server

The X server implementation is just a proof of concept. Since neither the interface of the module nor the user space library are stable, there is no point in putting efforts in implementing applications that will have to change as the interfaces change. However, a proof that the module works was needed, and the X server will be one of the main clients of the VGA arbiter in most Linux desktops, so an implementation was created.

The current RAC code is a wrapper in the video drivers that executes some code before and after any video driver function is called. The X server implementation that uses the VGA Arbiter uses exactly the same idea: it wraps the video drivers and calls lock and unlock around every video driver function. Also, the locking and unlocking functions are called in a few other parts of the code, to allow the server initialization and state change (between *setup* and *operating*).

Since this implementation is independent from the video drivers, it does not know what kind of resources the video cards decode, so it has to assume that every card decodes legacy IO and memory. It does not even know whether any specific driver function call will use the legacy VGA interface or not. Changing this could significantly improve the overall performance, because the locking and unlocking functions would be called less frequently, allowing better performance for all the applications.

The ideal implementation would be inside the drivers. They would directly call the library that provides the access to arbiter, so these calls would be made only when the

access to the legacy VGA interface is really needed, reducing the time each card would hold the lock and the amount of times the lock would be acquired. Also, the driver will tell what kind of resources it decodes, so it'll be able to stay out of the arbitration if it is not decoding legacy VGA.

Other relevant problem is to keep backwards compatibility with systems that does not have the VGA arbiter. The X server should use the VGA Arbiter if the system has one, or the RAC if the system does not.

Even with the performance problems detected, some tests were done and the results are presented on the next section. They showed that the implemented parts work, and also the performance impact of the current implementation. Of course, performance losses were expected.

5. Results

Two sets of experiments were conducted to evaluate the trade offs involving the VGA arbiter. In the first set (Section 5.1), a single-head X server – with one video card and one display – was stressed, one time using the arbiter and another time not using it. In the second set (Section 5.2), the difference between a multi-head X server – with more than one display, each one assigned to a video card – using the RAC and another using the arbiter were evaluated.

The experimental setup has been performed using a 2.26GHz Celeron processor and two nVidia GeForce FX 5500 PCI video cards with 256MB of video RAM (similar results were obtained using two Radeon RV100 QY PCI video cards with 128MB of video RAM). The operating system was Linux, with 2.6.23.12 Kernel version. The evaluations were primarily concerned with the performance issues added by inserting the VGA arbiter in the system. Part of the results have been produced with x11perf [McCormack et al. 2007], which is a tool used for measuring the X server performance through repeated sequences of primitive operations.

Besides these two sets of experiments, the reader has to keep in mind that the arbitration implemented here also adds the functionality to use various clients of the arbiter at the same time (e.g. to deploy a multiseat), which is a feature not present in the parts of the experiments that do not use the arbiter.

5.1. The Burden of the Arbiter

The first experiment compares a single-head X server using the arbiter and another single-head X server not using it. A counter has been inserted inside X to increment every time a lock and an unlock is performed for each of four chosen tasks. Besides the counter, an application has been built to measure how much time is spent just to perform a given number of locks and unlocks without any operation. This shows the amount of time spent inside the Kernel for a set of lock/unlock calls. This way, with the number of locks/unlocks of a desired task and the amount of time spent inside kernel we can deduce the overhead of the arbiter.

The first row of Table 1 shows the number of locks/unlocks needed when filling solid rectangles successive times (`x11perf -rect500`), which is a very intensive operation. This task took about 34 seconds to be completed and it locked/unlocked 308206

Procedure	# inside kernel
“x11perf -rect500” which lasts about 34 secs	308206
“mplayer -vo x11” playing a video of 30 secs	1542
one instance of X server and xclock	1360
one instance of X server	68

Table 1. The number of locks/unlocks performed by a single-head X server using the arbiter.

times on average (all the experiments in this paper were performed ten times and the average result was picked). Using the the application that measures the amount of time a specific number of locks/unlocks take, it was obtained that the time spent to lock/unlock 308206 times is about 0.8048845 seconds. So it can be concluded that for a very intensive operation that lasts 34 seconds, 0.8048845 seconds have been spent in the arbitration. In other words, only 2.35 % of overhead happened.

Other kinds of operations which use less frequently the VGA interface achieved better results. For instance, `mplayer -vo x11` with a video playing for 30 seconds, spent only 0.004428 seconds inside the arbitration, i.e., 0.01476 % which is almost insignificant. The third and fourth rows of the Table 1 are useful as a base of comparison showing how many times lock/unlock was performed by very simple tasks.

The intent of the second experiment is to show how an application, which is a the real use case, will burden the X server when the VGA arbiter is active. A simple game (Kobo Deluxe) that does a lot of rendering operations was used to evaluate this case[Olofson]. The game itself shows the amount of frames per second (FPS) it outputs. The X server without the arbiter showed an average of 315.53 FPS against 311.90 FPS by the X server using the arbiter. On other words, the first X server outperforms the second in about 1.16 %.

Therefore, this first set of experiments show that the burden of the VGA arbiter is insignificant in the majority of the operations. Thus, the overhead of the arbiter in systems that do not need it – like a single-head X – is too little if compared to the functionality it offers – the possibility to run other applications that use the legacy VGA Interface.

5.2. RAC $\times$ VGA Arbiter

In the second set of experiments, the performance difference of a multi-head environment was evaluated, i.e, an X server using the RAC is compared to another one using the arbitration implemented in this paper. The experiments consisted of running two applications at the same time in each X server, one at each screen. This was useful because it stressed the semaphore of the arbiter, creating race conditions between the screens.

The first experiment, a common operation to fill solid rectangles repeatedly (`x11perf -rect500`) was started in each screen simultaneously. The X server using RAC obtained 3395 rectangles per second on screen one and 3400 on screen two. On the other hand, the VGA arbiter obtained 3385 and 3400 rectangles per second at screen one and two respectively. This leads to the conclusion that the performance overhead of the arbiter is comparable to the overhead of the RAC module, which was expected, since the arbiter implementation is based on the RAC implementation.

As in the first set of experiments, the second experiment showed a close to real

usage of the VGA interface arbitration with the Kobo Deluxe game. The X server using the RAC module shows an average of 162.86 FPS on screen one against 163.91 FPS using the VGA arbiter. On screen two, RAC shows 172.27 FPS and VGA arbiter 172.96 FPS. This is all summarized in Figure 2. The conclusion follows identical as the first experiment in this set: no significant loss of performance was observed by using the arbiter instead RAC.

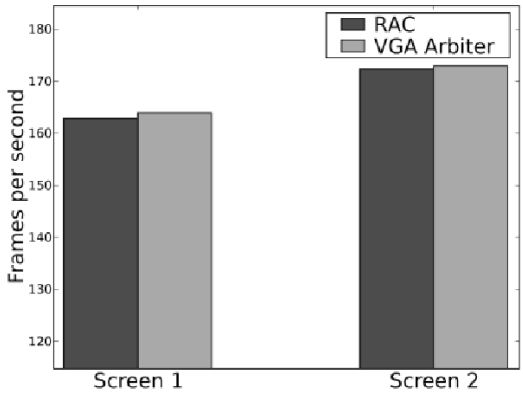

Figure 2. Frames per second difference between RAC and the VGA Arbiter using Kobo Deluxe game.

6. Considerations and Future Work

This paper presented a solution to the problem associated with the legacy VGA interface. The solution implements an arbitration scheme that can be used by any application, allowing different clients to use the legacy VGA interface concurrently (e.g. to open multiple instances of the X server, as in a multiseat). It consists of a Kernel module and an user space interface. An implementation has been deployed using Linux, a library to communicate with it and a module of the X server. Although this is based on Linux, its ideas can be easily used to implement Kernel modules or applications on other operating systems.

Although performance was really not the goal, the experiments showed that even with the current solution, the performance loss is small. Thus, as a first future work, optimizations can be implemented to improve the results even further.

Another future work is to make the interrupt request (IRQ) use the VGA arbiter correctly. All the work done did not address it. This must be done carefully since the system can enter an inconsistent state if, for instance, the card generates an interrupt when the arbiter has disabled memory decoding on it. The arbiter would need to either forbid video cards to use interrupts if they are set to decode legacy space or have a software driver callback for disabling IRQ emission on a given card when it is disabled by the arbiter.

For portability purposes, an user space application with functionality comparable to VGA arbiter could be built, so those platforms that do not provide an arbiter implementation inside the kernel could rely on it.

Acknowledgments

The authors would like to thank Benjamin Herrenschmidt for the first efforts in providing an application independent solution for the presented problem. They would also like to thank the X.Org community for all the help and critics provided to improve the current work.

Availability

The Kernel module source code is licensed with GPL, and both the library and X implementation are licensed with a BSD-like license called MIT license. The source code can be downloaded at [Vignatti et al.].

References

Angebranndt, S., Drewry, R., Karlton, P., Newman, T., Scheifler, B., Packard, K., Wiggins, D., and Gettys, J. (2004). Definition of the Porting Layer for X v11 Sample Server. *X Consortium Inc and X.Org Foundation*.

Castilho, M. A., Sunye, M. S., Weingaerter, D., Bona, L. C. E., Silva, F., Direne, A. I., Garcia, L. S., and Guedes, A. L. P. (2006). *Making government policies for education possible by means of Open Source technology: a successful case*, volume 1. Hershey - USA: Idea Group Publishing, 1st edition.

Corbet, J., Rubini, A., and Kroah-Hartman, G. (2005). *Linux Device Drivers*. O'Reilly, 3rd edition.

Herrenschimdt, B. VGA arbiter implementation proposal. `http://lists.freedesktop.org/archives/xorg/2005-March/006745.html`.

Herrenschimdt, B. VGA arbitration: API proposal. `http://lists.freedesktop.org/archives/xorg/2005-March/006663.html`.

Lo, L.-T., Watson, G. R., and Minnich, R. G. (2005). FreeVGA: architecture independent video graphics initialization for linuxBIOS. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 50–50, Berkeley, CA, USA. USENIX Association.

McCormack, J., Karlton, P., Angebranndt, S., Kent, C., Packard, K., and Gill, G. (2007). x11perf - x11 server performance test program.

Oliveira, A. C., Neto, A. H., Zanoni, P. R., and Vignatti, T. (2008). Xat - X Address Translation. `http://www.c3sl.ufpr.br/multiseat/xat/`.

Oliveira, A. C., Vignatti, T., Weigaertner, D., Silva, F., Castilho, M., and Sunye, M. (2006). Um modelo de computação multiusuário baseado em computadores pessoais. *VII Workshop sobre Software Livre*, pages 135–140.

Olofson, D. Kobo Deluxe Official Home. `http://olofson.net/kobodl/`.

Scheifler, R. W. and Gettys, J. (1997). *X Window System: core and extension protocols*. Digital Equipment Corp., Acton, MA, USA.

Shanley, T. and Anderson, D. (1999). *PCI System Architecture*. Addison-Wesley, 4th edition.

Vignatti, T., Zanoni, P. R., and Herrenschmidt, B. X.Org Foundation - VGA Arbiter Source Code. `http://www.x.org/wiki/VgaArbiter`.

X.Org Foundation. RAC.Notes - Resource Access Control system. `http://cgit.freedesktop.org/xorg/xserver/tree/hw/xfree86/doc/devel/RAC.Notes`.

Zanoni, P., Vignatti, T., Oliveira, A. C., , Silva, F., and de Bona, L. C. E. (2008). O Estado Atual dos Multiterminais. *IX Workshop sobre Software Livre*.